TELECOM Nancy 2ème année
Janvier 2017 - Mai 2017

Projet de compilation (Module PCL).

L'objectif de ce projet est d'écrire un compilateur du langage LOOC (`Langage Orienté Objet pour la Compilation`)*, un mini-langage à objets, très simplifié. Ce compilateur produira en sortie du code assembleur* `microPIUP/ASM`[1]*, code assembleur que vous avez étudié dans le module PFSI de première année.*

1 Réalisation du projet.

Vous travaillerez par groupes de 4 élèves, et en aucun cas seul ou à deux. Si vous êtes amenés à former un trinôme, 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 pour les étapes d'analyse lexicale et syntaxique. Vous génèrerez ensuite dans un fichier du code assembleur au format *microPIUP/ASM*.
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 relativement explicite comprenant, dans la mesure du possible, un numéro de ligne. Votre compilateur peut s'arrêter après chaque erreur syntaxique détectée (pas d'obligation de reprise). En revanche, votre compilateur doit impérativement poursuivre l'analyse après avoir signalé une erreur sémantique.

Vous utiliserez un dépot (SVN ou `git`) sur la forge de TELECOM Nancy. Créez votre projet comme un sous-projet de COMPILATION_2A. Votre projet doit être privé, l'identifiant sera de la forme `login1` (où `login1` est le login du membre chef de projet du groupe). Vous ajouterez Sébastien Da Silva, Pierre Monnin et Suzanne Collin en tant que "Manager" de votre projet. Votre répertoire devra contenir tous les sources de votre projet, le dossier final (au format pdf) ainsi que le *mode d'emploi* pour utiliser votre compilateur.
Les dépots seront régulièrement consultés par les enseignants chargés de vous évaluer lors de la séance de soutenance de votre projet
En cas de litige sur la participation active de chacun des membres du groupe au projet, le contenu de votre projet sur la forge sera examiné. Les notes peuvent être individualisées.

Le module PROJET DE COMPILATION (qui ne fait pas partie du module TRAD1) est composé de 7 séances de TP : vous serez évalué en fin de projet (fin mai 2017) lors d'une soutenance au cours de laquelle vous présenterez le fonctionnement de votre compilateur, mais également au cours des différentes séances de TP qui composent le module. Vous rendrez aussi en fin de projet un dossier qui entrera dans l'évaluation de votre projet.

Déroulement et dates à retenir.

Séances TP 1 à 4 : prise en main du logiciel ANTLR, définition complète de la grammaire du langage et construction de l'arbre abstrait. On vous propose une initiation au logiciel ANTLR lors de la première séance de TP. Ensuite, vous définirez la grammaire du langage LOOC et la soumettrez à ANTLR afin qu'il génère l'analyseur syntaxique descendant. On vous demande de définir une **grammaire LL(1)** pour ce langage. L'étape d'analyse lexicale est réalisée parallèlement à l'analyse syntaxique.
Vous implémenterez ensuite l'arbre abstrait (AST) en vous aidant de l'exemple vu en séance de TD et en consultant le chapitre 7 du livre décrivant Antlr.

Vous aurez testé votre grammaire et la construction de l'AST sur des exemples variés de programmes écrits en LOOC (avec et sans erreur lexicales et syntaxiques). Vous ferez impérativement une démonstration de cette étape lors de la séance TP 4 au plus tard. Vous obtiendrez une note N_1 correspondant à cette première évaluation. Vous pouvez bien entendu prendre de l'avance et commencer la construction de la table des symboles au cours de ces 4 semaines...

[1] Le jeu d'instructions et son codage ont été définis par Alexandre Parodi et la syntaxe du langage d'assemblage par Karol Proch.

Séances TP 5 et 6 : construction de la table des symboles et implémentation des contrôles sémantiques. Au cours de cette itération, vous réfléchirez dans un premier temps à la construction de de la table des symboles.
Ensuite vous listerez par catégorie l'ensemble des contrôles sémantiques à réaliser pour ce langage (contrôles liés aux méthodes, aux expressions, aux déclarations des variables, etc). Chaque membre du groupe implémentera au moins 5 contrôles sémantiques (ce qui fera donc au total une vingtaine de contrôles...).
A la fin de cette itération, vous montrerez la TDS et les contrôles sémantiques sur des exemples de programmes de test (une visualisation, même sommaire, de la TDS est indispensable). Vous serez évalués et vous obtiendrez une seconde note N_2 en séance TP 6.

Séance TP 7 et fin du projet. Vous terminez le projet par la phase de génération de code. Pour cette dernière étape, vous veillerez à générer le code assembleur de manière incrémentale, en commençant par les structures "simples" du langage.

Les tests.

Votre projet final sera testé par les enseignants de TP et se fera en votre présence. Il est impératif que vous ayiez prévu des exemples de programmes permettant de tester votre projet et ses limites (ces exemples ne seront pas à écrire le jour de la démonstration) : vous pourrez nous montrer votre "plus beau" programme... Vous obtiendrez alors une note N_3 de démonstration.

Le dossier.

A la fin du projet et pour la veille du jour de de votre soutenance, vous rendrez un dossier (qui fournira une note N_4) comportant une présentation de votre réalisation. Ce dossier comprendra *au moins* :

- la grammaire du langage,
- la structure de l'arbre abstrait et de la table des symboles que vous avez définis,
- les erreurs traitées par votre compilateur,
- les schémas de traduction (du langage proposé vers le langage assembleur) les plus pertinents,
- des *jeux d'essais* mettant en évidence le bon fonctionnement de votre programme (erreurs correctement traitées, exécutions dans le cas d'un programme correct), et ses limites éventuelles.
- une fiche contenant un résumé de la contribution de chacun des membres du groupe (avec répartition des tâches et estimation du temps passé sur chaque partie du projet).

Vous remettrez ce dossier dans le casier de votre enseignant de TP.

Pour finir...

Répartissez les forces du groupe et parallélisez dès le début du projet tout ce qui peut l'être.
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 oeuvre, etc... mais il est interdit de copier du code source sur vos camarades.

La note finale (sur 20) de votre projet prend en compte les 4 notes attribuées lors des différentes évaluations. *Rappel : les notes peuvent être individualisées.*

La fin du projet est fixée au **mardi 23 mai 2017**, date prévue pour les soutenances finales ; lors de cette soutenance, on vous demandera de faire une démonstration de votre projet. Un planning vous sera proposé pour fixer l'ordre de passage des groupes.

Aucun délai supplémentaire ne sera accordé pour la fin du projet : les notes doivent être harmonisées et attribuées le soir même des soutenances pour les communiquer au jury du 24 mai.

2 Présentation du langage.

Aspects lexicaux et syntaxiques.

Dans le langage Looc un commentaire peut apparaître n'importe où dans le texte source : les commentaires commencent par /* et se terminent par */. Ils ne sont pas imbriqués.

On donne ci-dessous la grammaire du langage LOOC. Dans cette grammaire, les terminaux sont en lettres minuscules et les non-terminaux en lettres majuscules. Les autres symboles, tels `( ) + -` sont aussi des symboles terminaux et sont écrits en caractères gras. Les mots-clés seront en minuscules et également en gras ; ils sont réservés.
Les terminaux génériques sont constitués de la façon suivante :

- **cste_ent** est une suite de chiffres décimaux (au moins un chiffre),

- **cste_chaine** est une suite non vide de caractères quelconques délimités par le caractère ". Par exemple, `"ceci est une chaîne : #!{ 35"`

- **identifiant** est une suite de lettres (minuscule ou majuscule), de chiffres ou du caractère _ à l'exception de tout autre caractère. Un identificateur commence obligatoirement par une lettre. Le nombre de caractères n'est pas limité.
Il existe deux classes d'identifiants : les identifiants commençant par une lettre majuscule, notés `Idf` seront utilisés pour les noms de classe. Les identifiants commençant par une lettre minuscule, notés `idf` seront utilisés pour les variables, attributs et méthodes.

Les **mots-clés** du langage seront notés en gras dans la grammaire

Syntaxe concrète du langage.

L'axiome est `PROGRAM`. 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 de fois supérieur ou égal à 1. La notation $[\alpha]$ signifie que la séquence α est optionnelle. Le symbole | indique une alternative dans la grammaire.

PROGRAM	⟶	CLASS_DECL* VAR_DECL* INSTRUCTION+
CLASS_DECL	⟶	**class** Idf [**inherit** Idf] = **(** CLASS_ITEM_DECL **)**
CLASS_ITEM_DECL	⟶	VAR_DECL* METHOD_DECL*
VAR_DECL	⟶	**var** idf : TYPE ;
TYPE	⟶	Idf | **int** | **string**
METHOD_DECL	⟶	**method** idf **(** METHOD_ARGS* **) {** VAR_DECL* INSTRUCTION+ **}**
		| **method** idf **(** METHOD_ARGS* **) :** TYPE **{** VAR_DECL* INSTRUCTION+ **}**
METHOD_ARGS	⟶	idf : TYPE {, idf : TYPE}*
INSTRUCTION	⟶	idf := EXPRESSION ;
		| idf := **nil** ;
		| **if** EXPRESSION **then** INSTRUCTION [**else** INSTRUCTION] **fi**
		for idf **in** EXPRESSION .. EXPRESSION **do** INSTRUCTION+ **end**
		| **{** VAR_DECL* INSTRUCTION+ **}**
		| **do** EXPRESSION . idf **(** EXPRESSION {, EXPRESSION}* **);**
		| PRINT
		| READ
		| RETURN
PRINT	⟶	**write** EXPRESSION ;
		| **write** cste_chaîne ;
READ	⟶	**read** idf ;
RETURN	⟶	**return (** EXPRESSION **);**
EXPRESSION	⟶	idf
		| **this**
		| **super**
		| cste_ent
		| **new** Idf
		| EXPRESSION . idf **(** EXPRESSION {, EXPRESSION}* **)**
		| **(** EXPRESSION **)**
		| - EXPRESSION
		| EXPRESSION OPER EXPRESSION
OPER	⟶	+ | - | ***** | < | <= | > | >= | == | !=

Description générale.

Un programme a la structure suivante :

```
liste éventuellement vide de définitions de classes
blocs d'instructions faisant office de programme principal
```

Une classe décrit les structures communes aux objets de cette classes. Les attributs représentent l'état "interne" d'un objet et les méthodes les actions qu'il peut exécuter. Une classe peut être décrite comme spécialisation d'une unique classe existante (sa super-classe) et réunit alors les caractéristiques de sa super-classes et les siennes. Elle peut redéfinir les méthodes de sa super-classe. Les attributs de la super-classe sont visibles des méthodes de la sous-classe. La relation d'héritage induit une relation de "sous-type".

Les objets communiquent par envoi de messages. Un message est composé du nom d'une méthode avec ou sans aruments. Il est envoyé à l'objet receveur qui exécute le corps de la méthode et renvoie le résultat éventuel à l'appelant. En cas de redéfinition d'une méthode par une sous-classe, la méthode exécutée dépend du type dynamique du receveur (principe de la liaison dynamique).

Déclaration d'une classe.

Une classe commence par le mot clé **class** suivi de son nom. Après le symbole **(** on trouve la liste éventuellement vide des attributs suivie d'une liste éventuellement vide des déclarations de méthodes. Une classe se termine par le symbole **)**.
La clause optionnelle **inherit** indique le nom de la super-classe dans la relation d'héritage.

Les attributs.

Il n'y a pas de valeur par défaut pour les attributs et variables locales des méthodes. Un programme qui référence à l'exécution une variable non initalisée est indéfini. Les attributs ne sont visibles que dans le corps des méthodes de la classe ou de ses sous-classes. Un attribut peut masquer un attribut d'une super-classe qui n'est alors plus visible dans la sous-classe.

Les méthodes.

La visibilité des attributs est liée à la classe : une méthode peut accéder aux attributs de son receveur et de ceux de ses paramètres et variables locales, dès lors que les règles de visibilité des varaibles soient respectées.
Le corps d'une méthode est un bloc (cf. grammaire ci-dessus). La valeur retournée par une méthode doit être compatible avec le type de retour de la méthode (dans le cas des méthodes retournant une valeur). Les méthodes peuvent évidemment être récursives.

Les expressions.

- Les régions (ou blocs) sont délimitées par **(** et **)**, **{** et **}**.
- Les identificateurs `idf` correspondent à des noms de paramètres, variables locales ou attributs. Il existe 2 identificateurs réservés : `this` a le même sens qu'en Java (`this` réfère l'instance elle-même), et `super` ne peut apparaître que comme destinataire d'un appel à une méthode redéfinie dans la classe, pour indiquer une liaison statique vers la méthode correspondante de la super-classe.
- Une instanciation a la forme `new Classe(...)`. Elle crée dynamiquement et renvoie un objet de la classe considérée après avoir initialisé les éventuels attributs.
- Les envois de messages correspondent à la notion habituelle en programmation objet : association d'une méthode et d'un destinataire qui doit être explicite. La méthode appelée doit être visible dans la classe du destinataire.
- Il est possible de définir des méthodes ne renvoyant aucun résultat. Un appel à une telle méthode est introduit pat le mot-clé **do**. Par exemple, si **foo** est une méthode sans valeur de retour (et sans argument) de la classe **Point**, un appel à cette méthode est syntaxiquement le suivant :

```
n := new Point;
do n.foo();
```

- Objets non initialisés : on autorise l'affectation d'un objet, quel que soit son type, à la valeur particulière *nil*, qui désigne en fait un objet non initialisé.
- Il est possible de définir des blocs anonymes : la syntaxe est alors la suivante `{ corps de bloc }` .
- La portée d'un identificateur de variable ou de paramètre formel est constituée de l'intégralité de la région où figure sa déclaration, moins les régions où le même identificateur est déclaré. La portée d'un identificateur de fonction est constituée de l'intégralité de la région englobant le texte de définition de cette fonction, moins les régions où le même identificateur est déclaré.

• Le passage des paramètres se fait par pointeurs, sauf pour les types prédéfinis `int` et `string`. Dans un appel de méthode, le type de chaque paramètre effectif doit être équivalent à celui du paramètre formel correspondant.

• Le domaine d'une itération est défini par un intervalle d'entiers. Le pas est égal à 1. La modification dans le corps de l'itération de l'une des variables intervenant dans la définition du domaine n'influe pas sur le nombre d'itérations effectuées.

• L'expression située derrière le mot clé **if** doit être de type **entier** ou égale à **nil**. Une expression évaluée à 0 sera considérée comme équivalente à $false$, toute autre valeur non nulle de l'expression donne à celle-ci la valeur $true$.

• L'affectation n'est autorisée qu'entre variables de type équivalent.

• Variables locales : on ajoute la possibilité de définir des variables locales au début du corps d'une méthode ou dans un bloc anonyme. Ces variables s'utilisent de la manière habituelle.

• La fonction prédéfinie **read** lit un entier sur l'entrée standard. La fonction **write** écrit sur une ligne de la sortie standard soit un entier, soit une chaîne de caractères.

• L'évaluation des opérateurs s'effectue de gauche à droite, et leur priorité est la suivante : $< > = <>$ ont même priorité. Celle-ci est supérieure à la priorité de $*$ elle-même supérieure à la priorité de $+$ et $-$ qui ont même priorité.

• L'opérateur unaire - note l'opposé pour un entier. Les opérateurs $<$ $>$ $<=$ $>=$ et $!=$ sont à opérandes entiers et à résultat booléen.

Aspects sémantiques.

Une classe ne peut référencer que des classes déjà définies. Un programme ne peut redéfinir une classe existante. Les noms des classes et des méthodes sont visibles partout. Dans une classe, un identificateur ne peut pas désigner à la fois un argument ou un attribut, et une méthode. Les régles de portée des arguments et des variables locales sont celles habituellement définies dans les langages impératifs, tels que Java. La portée d'une déclaration d'une variable locale commence à la fin de sa déclaration et se termine à la fin du bloc englobant cette déclaration. Une expression ne peut référencer que des variables ou attributs déjà déclarés.

Les contrôles de typages doivent être réalisés en tenant compte de l'héritage.

3 Génération de code.

Le code généré devra être en langage d'assemblage *microPIUP/ASM* écrit dans un fichier texte au format Linux d'extension `.src` , en utilisant un sous-ensemble des instructions de la machine.
Le fichier généré devra être assemblé à l'aide de l'assembleur qui générera un fichier de code machine d'extension `*.iup`.
Ce dernier sera exécuté à l'aide du simulateur du processeur `APR`[2].
Ces deux outils (assembleur et simulateur) fonctionnent sur toute machine Windows ou Linux disposant d'un runtime java. Ils sont inclus dans le fichier (archive java) `microPIUP.jar` disponible sur le serveur neptune dans le dossier `/home/depot/PFSI`.

Ce fichier supporte les commandes suivantes, en supposant que le fichier `microPIUP.jar` soit dans le dossier courant.

1. Assembler un fichier `projet.src` dans le fichier de code machine `projet.iup` :
   `java -jar microPIUP.jar -ass projet.src`
2. Exécuter en batch le fichier de code machine `projet.iup` :
   `java -jar microPIUP.jar -batch projet.iup`
3. Lancer le simulateur sur interface graphique :
   `java -jar microPIUP.jar -sim`

[2] Advanced Pedagogic RISC développé par Alexandre Parodi qui comporte toutes les instructions et modes d'adressage pour permettre l'enseignement général de l'assembleur, mais dont un sous-ensemble forme une machine RISC facilitant l'implémentation matérielle sur une puce et (on l'espère) l'écriture d'un compilateur.