TELECOM Nancy 2ème année
Janvier 2015 - Mai 2015

Projet de compilation.

L'objectif de ce projet est d'écrire un compilateur du langage `PLIC` *(Petit Langage d'Initiation à la Compilation), dont les constructions sont proches de celles du langage Pascal et du langage C que vous connaissez bien. 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 doit également essayer de poursuivre l'analyse après avoir signalé une erreur sémantique.

Vous utiliserez un dépot (SVN ou `git`) sur la forge de TELECOM Nancy (`https://forge.telecomnancy.univ-lorraine.fr`). Créez votre projet comme un sous-projet de COMPILATION_2A. (dans Projets Divers). 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 aux membres développeurs 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.
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 (courant mai 2015) 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 et définition complète de la grammaire du langage.

On vous propose une initiation au logiciel ANTLR lors de la première séance. Ensuite, 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 variés de programmes écrits en PLIC (avec et sans erreur lexicales et syntaxiques).

Vous ferez impérativement une démonstration de cette étape lors de la séance de la séance TP 4 au plus tard. Vous obtiendrez une note N_1 correspondant à cette première évaluation. Il n'est pas interdit de prendre de l'avance et de commencer la construction de l'arbre abstrait au cours de ces 4 semaines...

Séances TP 5 et 6 : construction de l'arbre abstrait et de la table des symboles.

Au cours de ces deux séances vous réfléchirez à la construction de l'arbre abstrait et de la table des symboles. A la fin de cette itération, vous montrerez ces deux structures sur des exemples de programmes de test (une visualisation, même sommaire, de ces structures est indispensable). Vous serez évalués et vous obtiendrez une seconde note N_2 en séance TP 6.

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

Séance TP 7 et 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. 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 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 d'évaluation de la répartition du travail : répartition des tâches au sein de votre binôme, estimation du temps passé sur chaque partie du projet.

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

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 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 19 mai 2015**, date à partir de laquelle on vous demandera de faire une démonstration de votre projet. Un planning vous sera proposé pour fixer l'ordre de passage des binômes.

Aucun délai supplémentaire ne sera accordé pour la fin du projet : les soutenances de vos projets doivent être réalisées pour la fin du mois de mai, et les notes harmonisées et attribuées avant le premier jury de juin.

2 Présentation du langage.

Aspects lexicaux et syntaxiques.

Dans le langage PLIC 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 complète du langage PLIC. 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 délimités par le caractère **"**
- **idf** est une suite de lettres (minuscule ou majuscule) et de chiffres, à l'exception de tout autre caractère. Un identificateur commence obligatoirement par une lettre. Le nombre de caractères n'est pas limité.

La différence minuscule/majuscule est significative : ainsi un texte peut contenir un identificateur DO qui ne sera pas confondu avec le mot clé **do**.

Grammaire 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	→	**do** {DECLARATION}* {INSTRUCTION}+ **end**
DECLARATION	→	DEC_VAR | DEC_FUNC | DEC_PROC
DEC_VAR	→	TYPE **idf** {, **idf**}*
TYPE	→	**integer** | **boolean** | ARRAY
ARRAY	→	**array [** BOUNDS **]**
BOUNDS	→	**cste_ent .. cste_ent** {, **cste_ent .. cste_ent**}*
DEC_FUNC	→	ENT_FUNC {DECLARATION}* {INSTRUCTION}+ **end**
DEC_PROC	→	ENT_PROC {DECLARATION}* {INSTRUCTION}+ **end**
ENT_FUNC	→	**function** TYPE **idf** PARAM
ENT_PROC	→	**procedure idf** PARAM
PARAM	→	({FORMAL {, FORMAL}* })
FORMAL	→	{ **adr**} **idf :** TYPE
INSTRUCTION	→	AFFECTATION |
		BLOC |
		ITERATION |
		CONDITION |
		RETURN |
		READ |
		WRITE
BLOC	→	**begin** {DECLARATION}* {INSTRUCTION}+ **end**
AFFECTATION	→	**idf** = EXP
ITERATION	→	**for idf in** EXP **..** EXP **do** {INSTRUCTION}+ **end**
CONDITION	→	**if** EXP **then** {INSTRUCTION}+ {**else** {INSTRUCTION}+} **fi**
RETURN	→	**return** (EXP)
READ	→	**read idf**
WRITE	→	**write** EXP |
		write cste_chaine
EXP	→	**idf** |
		cste_ent |
		true |
		false |
		idf ({EXP {, EXP}* }) |
		(EXP) |
		- EXP |
		EXP OPER EXP
OPER	→	+ | - | * | < | <= | > | >= | == | !=

Aspects sémantiques.

• Les régions (ou blocs) sont délimitées par **do** et **end**, **function** et **end**, **procedure** et **end**, **begin** et **end**.

• Il y a un seul espace des noms.

• 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 mot clé **adr** devant un paramètre formel signifie que ce paramètre est passé par adresse. En l'absence de ce mot-clé, le mode de passage des paramètres est le mode par valeur. Seule une variable peut être passée par adresse.

• Les éléments de tableau sont de type entier.

• 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 **boolean**.

• Deux objets O_1 et O_2 sont dits de types équivalents s'ils vérifient l'une des conditions suivante :

- O_1 et O_2 sont toutes deux des variables entières ou booléennes,
- O_1 et O_2 sont des fonctions avec le même nombre de paramètres, chacun d'eux de type équivalent, ainsi que leur résultat.

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

• Dans un appel de fonction ou de procédure, le type de chaque paramètre effectif doit être équivalent à celui du paramètre formel correspondant.

• L'instruction **return** permet la sortie d'une fonction en retournant la valeur de l'expression donnée en argument. En l'absence de **return**, la valeur retournée est indéfinie.

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

• Si les deux opérandes des opérateurs binaires $+$ $-$ et $*$ sont entiers, le résultat est entier. Si les deux opérandes des opérateurs binaires $+$ $-$ et $*$ sont booléens, le résultat est booléen. Dans ce cas, ces opérateurs notent alors respectivement le **ou**, **ou exclusif** et le **et** logiques. Dans les autres cas, le type du résultat est indéfini. L'opérateur unaire - note l'opposé pour un entier et la négation pour un booléen. Les opérateurs $<$ $>$ $<=$ $>=$ et $!=$ sont à opérandes entiers et à résultat booléen.

Exemples de programmes.

Le programme suivant est un exemple de programme simple écrit en PLIC.

```
/* Un exemple de programme très simple */
do
   integer i, j, maximum

   function int fMax (x: integer, y: integer)
      integer result
      if x > y then result = x
               else result = y
      return (result)
    end /* function */

    procedure theEnd ()
      write "that's all !"
    end

   i = -3
   j = 0
   maximum = fMax(i, j)
   write maximum
   theEnd()
end
```

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.

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.

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`