TELECOM Nancy 2ème année
Janvier 2013 - Mai 2014

Projet de compilation.

L'objectif de ce projet est d'écrire un compilateur du langage `LEAC` *(Langage Elémentaire Algorithmique pour 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 devrez utiliser un dépot SVN sur la forge de TELECOM Nancy (`https://redmine.esial.uhp-nancy.fr`). Créez votre projet comme un sous-projet de COMPILATION-2014 (dans Projets 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 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.

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 2014) 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.

Semaines 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 Leac (avec et sans erreur lexicales et syntaxiques).

Vous ferez impérativement une démonstration de cette étape lors de la séance de la semaine 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...

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

Au cours de ces deux semaines 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 semaine 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.

Semaine 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 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 sur vos camarades.

La note finale (sur 20) de votre projet prend en compte les 4 notes attribuées lors des différentes évaluations.

La fin du projet est fixée au **lundi 12 mai 2014**, date à partir de laquelle on vous demandera de faire une démonstration de votre projet. Un planning vous sera proposé pour fixer l'odre 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.

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	→	**int** .. **int** | **int** .. **int**, 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 dans un super langage */
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()
}
```

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 `toto.src` dans le fichier de code machine `toto.iup` :
   `java -jar microPIUP.jar -ass toto.src`
2. Exécuter en batch le fichier de code machine `toto.iup` :
   `java -jar microPIUP.jar -batch toto.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.

Sous-ensemble ("Reduced Instruction Set") des instructions et modes d'adressage à utiliser :

```
ADC, ADD, SUB, ADC, XOR, AND, OR (mode registre)

NEG, NOT, SWB, RRC, RRC, RLC, SHL, SRL, SRA  (mode registre)

ADQ, LDQ  (mode rapide)

LDW, LDB, STW, STB (modes registre, indirect, immédiat seulement);

MPC (mode registre seulement)

JEA (mode indirect seulement)

Bcc  (saut relatif court avec déplacement en mode rapide)
avec les conditions: always MP, BEQ, BNE, GE, LE, GT, LW, AE, AB, BL, BE, VS, VC

NOP

TRP (appel à une trappe logicielle en mode immédiat seulement)
```

Exemples de code.

On suppose qu'on a écrit au début :

```
EXIT_EXC   EQU 64              // n° d'exception de EXIT
READ_EXC   EQU 65              // n° d'exception de READ
WRITE_EXC  EQU 66              // n° d'exception de WRITE
STACK_ADRS EQU 0x1000          // base de pile en 1000h
SP         EQU R15             // alias pour R15
...
           LDW SP, #STACK_ADRS  // charge SP avec STACK_ADRS
```

Écrire une chaîne de caractères sur le canal standard (écran) :
R0 contient l'adresse de la chaîne de caractères

```
TRP #WRITE_EXC  // lance la trappe WRITE
```

Lire une chaîne de caractères depuis le clavier après pression sur la touche "Entrée" :
R0 contient l'adresse de la zone mémoire où placer la chaîne de caractères

```
TRP #READ_EXC  // lance la trappe READ
```

Arrêter l'exécution :

```
TRP #EXIT_EXC // lance la trappe EXIT
```

Empiler le contenu d'un registre R :

```
ADQ -2, SP  // décrémente le pointeur de pile SP
STW R, (SP) // sauvegarde le contenu du registre R sur la pile
```

Dépiler le contenu d'un registre R :

```
LDW R,(SP)  // charge le registre R avec le sommet de pile
ADQ 2,SP    // incrémente le pointeur de pile SP
```

Appeler une fonction ou procédure dont l'adresse est contenue dans le registre R :

```
MPC R1        // charge le contenu du PC dans R1
ADQ 8, R1     // ajoute 4 à R1: R1 contient l'adresse de retour
ADQ -2, SP    // décrémente le pointeur de pile SP
STW R1, (SP)  // sauvegarde l'adresse de retour sur le sommet de pile
JEA (R)       // saute à l'instruction d'adresse absolue dans R
```

Retour d'une procédure ou fonction :

```
LDW R1, (SP)  // charge R1 avec l'adresse de retour
ADQ 2, SP     // incrémente le pointeur de pile SP
JEA (R1)      // saute à l'instruction d'adresse absolue dans R
```

Mise à jour de l'environnement de pile ("stack frame") lors de l'exécution d'une routine :
BP est le Base Pointer (a priori le registre R13)
SP est le stack Pointer (nécessairement le registre R15)

```
ADQ -2, SP    // décrémente le pointeur de pile SP
STW BP, (SP)  // sauvegarde le contenu du registre BP sur la pile
LDW BP, SP    // charge BP avec SP (pointe sur la sauvegarde de l'ancien BP dans la pile)
SUB SP, R, SP // réserve R octets sur la pile pour variables (etc.) locales
```