TELECOM Nancy 2ème année
Octobre 2023 - Janvier 2024 - Avril 2024

Projet de compilation - Modules PCL1 et PCL2

L'objectif de ce projet est d'écrire un compilateur d'un langage de "haut niveau", en développant toutes les étapes qui le compose, depuis l'analyse lexicale jusqu'à la production de code assembleur ARM. Il s'agit d'un petit fragment du langage Ada[1] *que nous dénommerons* `canAda` *par la suite*[2].
D'après un sujet ENS - 2016.

Le sujet ci-dessous décrit la syntaxe du langage `canAda` ainsi que le travail à réaliser au cours des modules PCL1 et PCL2.

1 Réalisation du projet

L'écriture du compilateur complet concerne les deux modules PCL1 et PCL2. La partie PCL1 (octobre à mi-janvier) s'adressera à tous les élèves : il s'agit d'écrire les analyseurs lexicaux et syntaxiques, et d'implémenter la construction de l'arbre abstrait. La partie PCL2 (février à mi-mai) concernera le développement des contrôles sémantiques et la génération de code assembleur. Seuls les élèves des approfondissements IL et ISS sont concernés par ce second module PCL2.

Pour ce projet, les élèves des approfondissements IAMD, SLE et SIE ainsi que les élèves en mobilité au semestre 8 travailleront ensemble et formeront des groupes de 4. Les élèves des approfondissements IL et ISS sont concernés par les modules PCL1 et PCL2 et formeront des groupes de 4 sur ces 2 approfondissements uniquement.

Vous n'utiliserez aucun générateur d'analyseur lexical et syntaxique : vous aurez donc à développer par vous-même l'automate de reconnaissance des unités lexicales ainsi que l'analyseur syntaxique. La construction de l'analyseur syntaxique descendant à l'aide des "fonctions récursives" est conseillée.

Nous n'imposerons aucun langage pour le développement de votre compilateur, c'est à vous de choisir celui pour lequel l'ensemble des membres du groupe possède un maximum de compétences : C, Java, Python. Tous les membres des groupes de projet doivent travailler à part égale sur le projet.
Les groupes qui souhaitent utiliser un autre langage que ceux cités précédemment doivent contacter leur encadrant de TP.

Vous utiliserez un dépôt Git sur le Gitlab de TELECOM Nancy. Il devra être privé et l'identifiant sera de la forme login1 (où login1 est le login du membre chef de projet de votre groupe). Vous ajouterez Olivier Festor, Baptiste Buchi et Suzanne Collin en tant que "Master" de votre projet. Votre répertoire devra contenir tous les fichiers sources de votre projet, les dossiers intermédiaires et finals (au format PDF) ainsi qu'un *mode d'emploi* pour utiliser votre compilateur.

Les dépots seront régulièrement consultés par les enseignants chargés de vous évaluer lors des séances de TP et lors de la soutenance finale 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 le dépôt sera examiné. Les notes peuvent être individualisées.

2 Première partie : module PCL1

PCL1 : séances TP 1 à 4 - définition complète de la grammaire du langage, conception des analyseurs lexical et syntaxique, construction de l'arbre abstrait, visualisation.

Les 4 séances de TP ont lieu les semaines 41, 42, 46 et 50. Vous aurez commencé l'implémentation de l'analyseur

[1] Langage de programmation orienté objet dont une première version est sortie au début des années 1980. Le nom *Ada* a été choisi en hommage à Ada Lovelace, sans doute la première informaticienne de l'histoire.
[2] canAda : "If you can do it, you can Ada" - Christophe Bouthier.

lexical à la semaine 42. En particulier, vous aurez défini sa structure et vous pourrez montrer quelques exmples d'unités lexicales reconnues.

Lors du TP de la semaine 46, vous aurez défini la grammaire, vous aurez commencé l'analyse syntaxique et écrits des programmes de test. Vous aurez également réfléchi à la structure de l'arbre abstrait, à sa mise en oeuvre, et à sa visulisation.
Nous prendrons note de vos réalisations à cette date.

Votre compilateur doit signaler les erreurs lexicales, syntaxiques (et par la suite, sémantiques) rencontrées. Lorsqu'une de ces erreurs est rencontrée, elle doit être signalée par un message le plus explicite possible, comprenant, dans la mesure du possible, un numéro de ligne.
Votre compilateur ne doit pas s'arrêter après une erreur lexicale ou syntaxique détectée, mais tenter de poursuivre l'analyse syntaxique.

Pour la partie PCL2 : votre compilateur devra impérativement poursuivre l'analyse sémantique après avoir signalé une erreur sémantique.

Le TP de la semaine 50 est une séance d'évaluation intermédiaire où vous nous montrerez l'avancement de votre projet. Vous aurez testé votre compilateur sur des exemples variés de programmes corrects écrits en langage CANADA, et sur des exemples comprenant des erreurs lexicales et syntaxiques.

Cette évaluation intermédiaire entrera dans la note finale du module PCL1.

La soutenance finale de cette partie PCL1 est fixée aux 17 et 18 janvier 2024.

Vous montrerez lors de cette soutenance des exemples de programmes écrits en `canAda` et les arbres abstraits correspondants. Il est impératif que vous ayez prévu pour la soutenance des exemples de programmes permettant de tester votre projet. Ces exemples ne seront pas à écrire le jour de la démonstration.

Vous rendrez lors de cette soutenance un court **rapport d'activité** présentant le travail réalisé et les difficultés rencontrées. Vous pouvez inclure dans ces documents la grammaire du langage, la structure de l'arbre abstrait et quelques jeux d'essais. Ce rapport n'a pas besoin d'être long, mais il doit être informatif.
N'oubliez pas les éléments de gestion de projet : répartition du travail et des tâches au sein du groupe et estimation du temps passé sur chaque étape de cette première partie du projet.

Vous remettrez ce rapport d'activité dans le casier de votre enseignant de TP pour le lundi 15 janvier 2024.

3 Seconde partie : module PCL2

Vous continuez votre projet en implémentant les contrôles sémantiques spécifiques à ce langage. Vous aurez réfléchi à la structure de la TDS pour la première séance de TP de PCL2.

Une fois cette étape de contrôles sémantiques réalisée, vous passerez à l'étape de génération de code. Pour cette dernière phase, vous veillerez à générer le code assembleur ARM de manière incrémentale, en commençant par les constructions "simples" du langage.

Des séances de TP sont prévues pour cette seconde partie PCL2. Nous reviendrons vers vous courant janvier à ce sujet.

A la fin du projet, donc à la fin du module PCL2, et pour le **lundi 13 mai 2024 - 18h**, vous rendrez un dossier qui complètera le précédent rapport d'activités de PCL1 et comprendra :

- les schémas de traduction du langage proposé vers le langage assembleur
- des *jeux d'essais* mettant en évidence le bon fonctionnement de votre compilateur, et ses limites éventuelles.
- une partie *gestion de projet*, à savoir une fiche d'évaluation de la répartition du travail sur cette seconde période avec la répartition des tâches au sein de votre binôme, l'estimation du temps passé sur chaque partie du projet, et le Gantt final.
- les divers CR de réunion que vous aurez rédigés.

La fin du projet est fixée au mardi 14 mai 2024, date prévue pour les soutenances finales.

Lors de cette soutenance, on vous demandera de faire une démonstration de votre compilateur. Un planning vous sera proposé pour fixer l'ordre de passage des groupes.
Il est impératif que vous ayez prévu des exemples de programmes permettant de tester votre projet et ses limites : vous nous montrerez lors de votre soutenance votre "plus beau programme" écrit en langage CANADA...

Aucun délai supplémentaire ne sera accordé pour la fin du projet.

Concernant l'évaluation de la génération de code, on vous fournira 4 "niveaux" de programmes écrits dans le langage du projet, ce qui vous permettra de vous situer dans cette phase de génération de code. Le niveau 2 est généralement celui requis pour obtenir une note de 10/20. Ceci n'est qu'une indication, car la note finale dépend aussi des évaluations intermédiaires et du dossier rendu.
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. Tout comme il est vivement déconseillé de demander à des "intelligences" d'écrire le compilateur à votre place... Si cela devait toutefois se produire, nous saurons en tenir compte dans nos évaluations.

4 Présentation du langage

4.1 Conventions lexicales

Les espaces, tabulation et retour-chariots constituent les "blancs" dans le texte d'un programme. Un commentaire débute par -- et s'étend jusqu'à la fin de la ligne.

Les *identificateurs* d'un programme commencent par une lettre minuscule ou majuscule et peuvent ensuite être constitués d'une répétition de lettres (minuscule ou majuscule) ou de chiffres de 0 à 9 ou du caractère _
L'expression régulière suivante $\langle ident \rangle$ décrit un identificateur :

$\langle chiffre \rangle$ ::= [0-9]
$\langle alpha \rangle$ ::= [a-z | A-Z]
$\langle ident \rangle$ ::= $\langle alpha \rangle$ ($\langle alpha \rangle$ | $\langle chiffre \rangle$ | _)*

Les constantes entières et caractères sont définies de la manière suivante :

$\langle entier \rangle ::= \langle chiffre \rangle^+$
$\langle caractere \rangle$ ::= ' $\langle un\ des\ 95\ caracteres\ ASCII\ imprimables \rangle$ '

Les mots clés du langage sont énoncés dans la section suivante.
Les identificateurs et les mots clés sont insensibles à la casse.

4.2 Syntaxe

Dans la spécification de la grammaire du langage, on utilisera les notations suivantes :

$\langle motif \rangle^*$: répétition du motif $\langle motif \rangle$ un nombre quelconque de fois ou aucune fois
$\langle motif \rangle^*_t$: comme précédemment, avec les occurences séparées par le terminal t
$\langle motif \rangle^+$: répétition du motif $\langle motif \rangle$ au moins une fois
$\langle motif \rangle^+_t$: comme précédemment, avec les occurences séparées par le terminal t
$\langle motif \rangle$? : utilisation optionnelle de $\langle motif \rangle$, c'est à dire 0 ou 1 fois

Les associativités et précédences des opérateurs sont données dans la table ci-dessous, de la plus faible à la plus forte précédence.

opérateur	associativité	précédence
or, *or else*	gauche	plus faible
and, *and then*	gauche	
not		
$=$ $/=$		
$>$ $>=$ $<$ $<=$		
$+$ $-$	gauche	
$*$ / *rem*	gauche	
$-$ (*unaire*)		
.	gauche	plus forte

Les fichiers sources que vous écrirez respecteront la "grammaire" donnée page suivante.

Dans cette grammaire, l'axiome est le non-terminal `fichier`. Les mots clés sont listés ci-dessous:

```
access      and       begin      else      elsif     end
false       for       function   if        in        is
loop        new       not        null      or        out
procedure   record    rem        return    reverse   then
true        type      use        while     with
```

⟨*fichier*⟩ ::= `with Ada.Text_IO; use Ada.Text_IO;`
`procedure` ⟨*ident*⟩ `is` ⟨*decl*⟩*
`begin` ⟨*instr*⟩+ `end` ⟨*ident*⟩? `;` `EOF`

⟨*decl*⟩ ::= `type` ⟨*ident*⟩ `;`
| `type` ⟨*ident*⟩ `is access` ⟨*ident*⟩ `;`
| `type` ⟨*ident*⟩ `is record` ⟨*champs*⟩+ `end record ;`
| ⟨*ident*⟩+, `:` ⟨*type*⟩ (`:=` ⟨*expr*⟩)? `;`
| `procedure` ⟨*ident*⟩ ⟨*params*⟩? `is` ⟨*decl*⟩*
`begin` ⟨*instr*⟩+ `end` ⟨*ident*⟩? `;`
| `function` ⟨*ident*⟩ ⟨*params*⟩? `return` ⟨*type*⟩ `is` ⟨*decl*⟩*
`begin` ⟨*instr*⟩+ `end` ⟨*ident*⟩? `;`

⟨*champs*⟩ ::= ⟨*ident*⟩+, `:` ⟨*type*⟩ `;`

⟨*type*⟩ ::= ⟨*ident*⟩
| `access` ⟨*ident*⟩

⟨*params*⟩ ::= `(` ⟨*param*⟩+; `)`

⟨*param*⟩ ::= ⟨*ident*⟩+, `:` ⟨*mode*⟩? ⟨*type*⟩

⟨*mode*⟩ ::= `in` | `in out`

⟨*expr*⟩ ::= ⟨*entier*⟩ | ⟨*caractère*⟩ | `true` | `false` | `null`
| `(` ⟨*expr*⟩ `)`
| ⟨*accès*⟩
| ⟨*expr*⟩ ⟨*opérateur*⟩ ⟨*expr*⟩
| `not` ⟨*expr*⟩ | `-` ⟨*expr*⟩
| `new` ⟨*ident*⟩
| ⟨*ident*⟩ `(` ⟨*expr*⟩+, `)`
| `character ' val (` ⟨*expr*⟩ `)`

⟨*instr*⟩ ::= ⟨*accès*⟩ `:=` ⟨*expr*⟩ `;`
| ⟨*ident*⟩ `;`
| ⟨*ident*⟩ `(` ⟨*expr*⟩+, `) ;`
| `return` ⟨*expr*⟩? `;`
| `begin` ⟨*instr*⟩+ `end ;`
| `if` ⟨*expr*⟩ `then` ⟨*instr*⟩+ (`elsif` ⟨*expr*⟩ `then` ⟨*instr*⟩+)*
(`else` ⟨*instr*⟩+)? `end if ;`
| `for` ⟨*ident*⟩ `in reverse?` ⟨*expr*⟩ `..` ⟨*expr*⟩
`loop` ⟨*instr*⟩+ `end loop ;`
| `while` ⟨*expr*⟩ `loop` ⟨*instr*⟩+ `end loop ;`

⟨*opérateur*⟩ ::= `=` | `/=` | `<` | `<=` | `>` | `>=`
| `+` | `-` | `*` | `/` | `rem`
| `and` | `and then` | `or` | `or else`

⟨*accès*⟩ ::= ⟨*ident*⟩ | ⟨*expr*⟩ `.` ⟨*ident*⟩

4.3 Compléments sur la syntaxe du langage canAda

- Le mot clé *reverse* dans une itération indique que l'intervalle est à parcourir en sens inverse de celui précisé.

- Les opérateurs logiques sont les opérateurs *or*, *or else*, *and*, *and then* et *not*.
L'opérateur *rem* représente le reste de la division entière, et le symbole / définit le quotient de la division entière.

- Les modes de passage des paramètres sont les suivants :

> `in` : les noms qui suivent ce mot clé dénoteront des constantes dont la valeur aura été spécifiée lors de l'appel. Ces paramètres sont donc uniquement accessibles en lecture.
>
> `in out` : les noms qui suivent dénoteront des variables. Toutes les opérations sont autorisées (lecture et écriture). La valeur du paramètre est prise en compte et peut donc être modifiée.

Sans mode indiqué pour un paramètre, le mode par défaut est `in`.

Remarque : il existe un 3^{ieme} mode de passage de paramètre en Ada, le mode `out`. Dans ce cas, la seule opération autorisée pour ce paramètre est l'affectation d'une valeur.

- Dans une déclaration de procédure ou de fonction, si un identificateur suit le mot clé `end`, alors celui-ci doit être identique au nom de la procédure ou de la fonction déclarée.

- Une structure, encore appelée enregistrement, est un objet composé d'une séquence de membres de types divers, portant tous un nom. Les enregistrements se déclarent à l'aide du mot clé `record`. On a ainsi :

```
type point is record
   abcisse : integer ;
   ordonnee : integer ;
end record;
```

- Les accès s'opèrent à l'aide du point `.`

```
p point; -- declaration d'une variable p de type point
p.x := 10; -- accès au champ x
```

- La notation `T'val(e)` signifie : e est de type entier et `T'val(e)` est l'élément de position e dans le type `T`. Dans la grammaire, il est prévu la notation : `character'val(e)` ce qui retournera le caractère `A` pour l'instruction `character'val(65)`.

- Les mots clés `access` et `new` sont relatifs à l'utilisation de pointeurs et à la définition de listes chaînées. Vous pourrez ne pas en tenir compte dans votre grammaire et supprimer les productions correspondantes.

4.4 Sémantique et vérifications complémentaires

Cette section détaille quelques vérifications complémentaires à effectuer :

- Dans une déclaration de procédure ou de fonction, si un identificateur suit le mot clé `end`, alors celui-ci doit être identique au nom de la procédure ou de la fonction déclarée.

- Dans un appel de fonction ou de procédure, si un paramètre formel est déclaré `in out`, alors le paramètre effectif correspondant doit être une *valeur gauche*, c'est à dire :

> - soit une variable,
> - soit une expression `x.f` avec x une variable de type enregistrement.

- Dans une fonction ou une procédure, si un paramètre formel `x` est déclaré `in` explicitement ou par défaut, alors sa valeur ne peut pas être modifiée avec une affectation `x := e`. De la même façon, si `x` est de type enregistrement, un champ de `x` ne peut pas être modifié avec une affectation `x.f := e`.

- La variable de la boucle `for` ne peut pas être affectée.

- L'exécution de toute fonction doit impérativement se terminer par une instruction `return`.

- Toutes les déclarations d'un même niveau doivent porter des noms différents. La seule exception est celle d'un type enregistrement déclaré, puis défini plus loin.

- Toute variable déclarée entre `is` et `begin` a une durée de vie qui est liée à la durée de vie de la procédure (lorsque la procédure commence, elle est créée, lorsque la procédure se termine, la variable est détruite). De plus, ces variables sont inaccessibles en dehors de leur procédure.

- Un type enregistrement déclaré doit être défini avant la fin des déclarations de même niveau, et avant toute introduction d'un niveau de déclaration supérieur.

- Tous les champs d'un même enregistrement doivent avoir des noms différents.

- Les fonctionnalités d'entrées/sorties sont disponibles en Ada dans le paquetage `Ada.Text_IO` pour les caractères et `Ada.Integer_IO` pour les entiers. Dans ce projet, on se limitera à la sortie `put()` qui imprimera à l'écran la valeur du paramètre. Ainsi, `put('a')` imprimera le caractère a et `put(x)` imprimera le contenu de x, x étant une variable). Le nom `put` est réservé.

4.5 Un exemple de programme écrit en canAda

Voici un petit exemple de programme écrit en canAda. Vous trouverez sur internet un grand nombre d'exemples écrits en Ada, cela vous donnera un aperçu de ce qu'il est possible d'écrire dans la syntaxe du langage canAda pour ce projet.

```
with Ada.Text_IO ;  use Ada.Text_IO ;

procedure unDebut is

  function aireRectangle(larg : integer; long : integer) return integer is
  aire: integer;
  begin
     aire := larg * long ;
  return aire
  end aireRectangle ;

  function perimetreRectangle(larg : integer; long : integer) return integer is
  p : integer
  begin
     p := larg*2 + long*2 ;
  return p
  end perimetreRectangle;

     -- VARIABLES
choix : integer ;

     -- PROCEDURE PRINCIPALE

begin
  choix := 2;

  if choix = 1
     then valeur := perimetreRectangle(2, 3) ;
          put(valeur) ;
     else valeur := aireRectangale(2, 3) ;
          put(valeur) ;
  end if;
end unDebut ;
```

4.6 Pour vous documenter...

Sur Arche, dans la rubrique Projet de Compilation, vous trouverez :

1. Pour la théorie et l'implémentation des analyseurs lexicaux et syntaxiques : l'excellent "Dragon Book".

2. Pour l'implémentation des tables des symboles : le chapitre de Grüne et al.

3. *Le langage Ada : une introduction* : Document du LIRMM - Comme vous le verrez, le langage du projet n'est qu'une toute petite partie du langage...

4. *Ada - Crash Course*