Cours de Compilation-Exercices

Master d'Informatique M1 2011–2012

19 septembre 2011

Table des matières

2 Analyse lexicale

2.1 Expressions régulières et analyse lexicale

Le but est d'esquisser un analyseur lexical pour le mini-langage ci-dessous. Pour chaque entité lexicale, on indique le token associé, sa valeur ainsi que l'expression régulière qui le définit.

	Token	**Valeur**	**Expression régulière**
Mots clés	IF, THEN, ELSE, BEGIN, END		le mot lui-même
Identificateurs	ID	le nom de l'identificateur	lettre (lettre | chiffre)*
Entier	INT	la valeur	(-)?(chiffre)+
Opération arithmétique	OP	l'opérateur	+ | - | * | /
Opération relationnelle	REL	l'opérateur	< | <= | > | >= | = | <>
Chaînes de caractères	STRING	la valeur de la chaîne	"n'importe quoi"
Réels	REAL	la valeur	(-)?entier.(entier)?(E (-)?entier)? avec entier=(chiffre)+
Commentaires		ignorés	(* n'importe quoi *)

On adopte de plus les conventions suivantes :

- Les caractères blancs, tabulation et retour chariot ne sont pas significatifs et peuvent apparaître en nombre quelconque,
- Les chaînes de caractères peuvent contenir des caractères spéciaux qui commencent par le caractère \: \", \n, \t, \\.
- les commentaires se terminent au premier `*)` rencontré et ne peuvent pas être emboités.

1. Donner les automates finis déterministes pour chacune des expressions régulières puis pour l'analyseur lexical complet.
2. Donner un exemple de chaîne de caractères dans laquelle on peut reconnaître plusieurs préfixes correspondant à des tokens différents.

3. Quelles sont les différences entre un analyseur lexical et la reconnaissance des mots par un automate fini classique.
4. Donner la structure générale de l'analyseur lexical, étant donné les fonctions élémentaires suivantes :
   - `init` désigne l'état initial de l'automate
   - `chaine`(x, y) où x et y sont des positions dans le buffer, renvoie la chaîne comprise entre les deux positions.
   - `terminal`(s) où s est un état, indique si l'état est final ou non.
   - `token`(s) si s est un état final, renvoie le token associé.
   - `transit`(s, c) renvoie l'état suivant de s par la transition étiquetée par c, renvoie `echec` dans le cas où une telle transition n'existe pas.
   - `lire`(ptr) renvoie caractère lu à la position `ptr` dans le buffer. La fonction échoue si elle rencontre une fin de fichier.

2.2 Reconnaissance d'une chaîne de caractères par une expression régulière

(Partiel novembre 2000)

Dans cet exercice, on se propose d'écrire une fonction de test d'appartenance d'un mot au langage $L(r)$ défini par une expression régulière r. Les expressions régulières seront représentées par des objets du type Caml suivant:

```
type  expreg =
      | Vide                          (* Langage vide ∅ *)
      | Epsilon                       (* Mot vide ϵ *)
      | Caractere  of char            (* Caractère c *)
      | Union      of expreg * expreg (* r1|r2 *)
      | Produit    of expreg * expreg (* r1r2 *)
      | Etoile     of expreg          (* r* *)
```

1. Définir une fonction `contient_epsilon` : `expreg` $\rightarrow$ `bool` qui détermine si le mot vide ϵ appartient au langage défini par une expression régulière donnée.
2. On définit le *résidu* d'une expression régulière r par rapport à un caractère c de la manière suivante :
$$Res(r, c) = \{\ w \mid cw \in L(r)\ \}$$
   (a) Calculer $Res(r, c)$ dans le cas où $r = (a|b)^*a$ et $c \in \{a, b\}$.
   (b) Le langage $Res(r, c)$ peut lui-même être décrit par une expression régulière. Écrire une fonction `residu` : `expreg` $\rightarrow$ `char` $\rightarrow$ `expreg` calculant à partir de r et de c une expression régulière r' telle que $L(r') = Res(r, c)$.
   (c) Calculer `residu` r c dans le cas où $r = (a|b)^*a$ et $c \in \{a, b\}$.
3. En déduire une fonction `reconnait` : `expreg` $\rightarrow$ `char list` $\rightarrow$ `bool` qui détermine si une liste de caractères appartient au langage défini par une expression régulière donnée.
4. Appliquer cette fonction à la reconnaissance du mot aba par l'expression régulière $r = (a|b) * a$

2.3 Construction d'automates déterministes à partir d'expressions régulières

On remarque tout d'abord que si un automate fini reconnaît le langage correspondant à l'expression régulière r alors à chaque lettre lue sur l'entrée on peut faire correspondre une occurrence d'une lettre dans l'expression r.
On indexe chaque occurrence d'une lettre dans l'expression régulière par un entier de manière à distinguer les différentes occurrences d'une même lettre.

On prend comme exemple l'expression régulière $(a|b)^*a(a|b)$. Après avoir indicé les caractères, on obtient l'expression: $(a_1|b_1)^*a_2(a_3|b_2)$. Si l'entrée est le mot *aabaab*, alors il correspond à l'expression régulière de la manière suivante:

$$a_1a_1b_1a_1a_2b_2$$

L'idée est de construire un automate dont les états sont des ensembles de lettres indexées correspondant aux occurrences qui peuvent être lues à un instant. Au départ on regarde quelles sont les premières lettres possibles. Dans notre exemple, il s'agit de a_1, b_1, a_2

Pour construire les transitions, il suffit de pouvoir calculer pour chaque occurrence d'une lettre, les occurrences qui peuvent la suivre. Par exemple si on vient de lire a_1 alors les caractères possibles suivants sont a_1, b_1, a_2, si on vient de lire a_2 alors les caractères possibles suivants sont a_3, b_2.

Si on est dans l'état initial (a_1, b_1, a_2) et qu'on lit un a alors c'est soit a_1 soit a_2 et donc les caractères qui peuvent être lus ensuite sont a_1, b_1, a_2, a_3, b_2.

On va montrer comment calculer systématiquement l'ensemble des caractères suivants.

1. Définir par récurrence sur la structure d'une expression régulière une fonction booléenne *null* qui dit si le langage reconnu contient ϵ.
2. Définir par récurrence sur la structure d'une expression régulière une fonction *début* (resp. *fin*) qui renvoie l'ensemble des premiers (resp. derniers) caractères des mots reconnus.
3. Les caractères suivants c dans une expression régulière r sont caractérisés par la propriété suivante:
   $d \in \mathit{suivant}(c, r)$ ssi il existe r_1, r_2 des expressions régulières telles que r_1r_2 soit une sous-expression de r et $d \in \mathit{début}(r_2)$ et $c \in \mathit{fin}(r_1)$
   ou bien
   il existe r_1 une expression régulière telle que r_1^* soit une sous-expression de r et $d \in \mathit{début}(r_1)$ et $c \in \mathit{fin}(r_1)$
   à partir de cette caractérisation définir par récurrence une fonction $\mathit{suivant}(c_i, r)$.
4. Pour construire l'automate déterministe, on procède de la manière suivante sur l'expression régulière indéxée r:
   (a) Ajouter un nouveau caractère $\sharp$ à la fin de l'expression
   (b) Calculer la fonction *suivant* pour chaque caractère indéxé.
   (c) Construire l'automate dont les états sont des ensembles de caractères indéxés, l'état initial est $\mathit{début}(r)$ les états d'acceptation sont tous ceux qui contiennent $\sharp$. On a une transition de q vers q' à la lecture du caractère c si

$$q' = \bigcup_{c_i \in q} \mathit{suivant}(c_i, r)$$

   Utiliser cet algorithme pour construire des automates déterministes reconnaissant les expressions :
   – $((\epsilon|a)b^*)^*$
   – $(a_1|b_1)^*a_2(a_3|b_2)(a_4|b_3)$

2.4 Tables de transitions compressées

1. Écrire un automate déterministe pour la reconnaissance des classes lexicographiques suivantes :

   if, then,else,id

où la classe *id* est celle des identificateurs (une lettre suivie d'une suite de caractères ou chiffres).
Les entrées possibles sont les caractères alpha-numériques et les délimiteurs. Les délimiteurs en début de token devront être ignorés par l'automate.
On utilisera l'état spécial -1 pour désigner l'échec, et on donnera pour chaque état final le token associé.

2. Donner les classes distinctes à utiliser pour les entrées.
3. Donner la table de transitions pour cet automate.
4. Cette table étant très pleine, comment est-il possible de la rendre plus creuse?
5. Construire la table de transitions en utilisant la représentation linéaire vue en cours. Proposer une stratégie pour ranger les éléments.
6. Expliciter la fonction de calcul de l'état suivant d'un état pour une entrée donnée.

2.5 Analyseur lexical pour indexer un programme

(Partiel novembre 2000)

On se propose de documenter les programmes écrits dans le langage SCALPA. Pour cela on souhaite créer un index de tous les noms de fonctions, de variables et de paramètres apparaissant dans le programme. L'index doit permettre de trouver pour chaque identificateur du programme, les lignes du programme où il est déclaré (comme nom de fonction, de variable ou comme paramètre de procédure) et les lignes du programme où il est utilisé. On ne cherche pas à distinguer plusieurs déclarations du même identificateur.

Ainsi pour le programme :

```
program test
var n : int
function fib(i:int,j:int):int
begin if n=0 then return i
      else if n=1 then return j
      else begin n:=n-1; fib(j,i+j) end
end;
function fact(i:int):int
var j:int
begin if n=0 then i
      else j:=n*i; n:=n-1; fact(j)
end;
begin n:=10; n:=fib(0,1); n:=fact(1) end
```

On souhaite obtenir les tables de déclarations et d'utilisations suivantes (éventuellement dans un ordre différent) :

Déclarations	
n	2
fib	3
i	3, 8
j	3, 9
$fact$	8

Utilisations	
n	4, 5, 6, 10, 11, 13
fib	6, 13
i	4, 6, 10, 11
j	5, 6, 11
$fact$	11, 13

On suppose que l'on dispose d'un type de données (impératif) `table` qui nous permet de stocker les informations contenues dans les index. Les fonctions de base sur ces tables sont :

`init`	: `unit` → `table`	`init` *()* crée une nouvelle table.
`ajoute`	: `table` → `string` → `int` → `unit`	`ajoute` *tbl nom n* ajoute la ligne numéro n pour l'entrée *nom* dans la table *tbl*.
`imprime`	: `table` → `string` → `unit`	`imprime` *tbl titre* imprime la table *tbl* avec le titre *titre*.

Syntaxe de SCALPA Nous donnons ci-dessous la grammaire complète du langage SCALPA. Dans cette grammaire, les mots-clés et les terminaux apparaissent en fonte droite. Le terminal `cte` représente les constantes entières et booléennes (`true` et `false`); le terminal `opb` (resp. `opu`) représente les opérateurs binaires (resp. unaires) du langage; le terminal `ident` représente les identificateurs qui sont formés d'un caractère alphabétique suivi d'une suite de caractères alphanumériques ou d'une apostrophe ou du caractère de soulignement. Les commentaires débutent par les deux caractères `(*` et se terminent par les deux caractères `*)`, ils peuvent être imbriqués.

program	::= `program` `ident` vardecllist fundecllist instr
vardecllist	::= ϵ — varsdecl — varsdecl `;` vardecllist
varsdecl	::= `var` identlist `:` typename
identlist	::= `ident` — `ident` `,` identlist
typename	::= atomictype — arraytype
identlist	::= `ident` — `ident` `,` identlist
atomictype	::= `unit` — `bool` — `int`
arraytype	::= `array` `[` rangelist `]` `of` atomictype
rangelist	::= int `..` int — int `..` int `,` rangelist
fundecllist	::= ϵ — fundecl `;` fundecllist
fundecl	::= `function` `ident` `(` arglist `)` `:` atomictype vardecllist instr
arglist	::= ϵ — arg — arg `,` arglist
arg	::= `ident` `:` typename — `ref` `ident` `:` typename
instr	::= `if` expr `then` instr — `if` expr `then` instr `else` instr
	— `while` expr `do` instr — lvalue `:=` expr — `return` expr — `return`
	— `ident` `(` exprlist `)` — `begin` sequence `end` — `begin` `end`
sequence	::= instr `;` sequence — instr `;` — instr
lvalue	::= `ident` — `ident` `[` exprlist `]`
exprlist	::= expr — expr `,` exprlist
expr	::= `cte` — `(` expr `)` — expr `opb` expr — `opu` expr
	— `ident` `(` exprlist `)` — `ident` `[` exprlist `]` — `ident`

Questions

1. Donner un programme `ocamllex` qui définit une fonction `index` qui étant donné un fichier contenant un programme SCALPA correct, construit et imprime l'index correspondant.
   Indications : On remarquera que pour résoudre ce problème, il n'est pas nécessaire de connaître la grammaire complète de SCALPA, seule la forme syntaxique des déclarations et la liste des mots-clé est importante.
   La notation portera sur l'explication de la démarche et la spécification des différentes composantes plus que sur la précision syntaxique du code ocamllex.
2. Peut-on modifier le programme ocamllex pour construire trois tables différentes, l'une pour les déclarations de fonctions, l'autre pour les déclarations de variables et la troisième pour les déclarations de paramètres ? Peut-on faire la même chose pour les tables d'utilisation ? Comment ?
3. Peut-on modifier le programme ocamllex pour associer chaque utilisation d'un identificateur à la déclaration correspondante ? Pourquoi ?

2.6 Repérage des numéros de lignes

(D'après partiel novembre 2001)

Les outils d'analyse lexicale tels que `ocamllex` permettent de repérer automatiquement pour une unité lexicale sa position par rapport au début du fichier analysé. Cette position est repérée par le nombre de caractères depuis le début du fichier.

Plus précisément, la fonction `Lexing.lexeme_start` (resp. `Lexing.lexeme_end`) de type `lexbuf → int` permet dans une action de retrouver le numéro du caractère du début (resp. du caractère qui suit la fin) de l'unité lexicale reconnue par l'expression régulière associée.

Cette information est particulièrement utile pour rapporter une erreur. Cependant, il est usuel d'afficher la position de l'erreur en indiquant le *numéro de la ligne* où s'est produite l'erreur ainsi que le *numéro de colonne* c'est-à-dire le nombre de caractères depuis le début de la ligne.

Le but de l'exercice est d'étudier deux méthodes différentes de calcul des numéros de ligne et de colonne à partir de l'information de la position d'un caractère.

On se donne un analyseur simplifié dans lequel on ne reconnaît que des identificateurs, des commentaires non emboités (délimités par `(*` et `*)`) et des chaînes de caractères (délimitées par `"` et qui ne contiennent pas le caractère `"`).

Cet analyseur utilise des fonctions pour manipuler un buffer afin de construire une chaîne de caractères de taille quelconque. La spécification de ces fonctions est donnée dans la table suivante :

Nom	Type	Spécification
`contents_buffer`	`unit → string`	`contents_buffer ()` calcule la chaîne stockée dans le buffer
`add_buffer`	`string → unit`	`add_buffer` s ajoute la chaîne s à la fin du buffer
`reset_buffer`	`unit → unit`	`reset_buffer ()` remet le buffer à zéro

Le squelette de notre analyseur lexical est :

```
{   type token = Ident of string | String of string | Eof
    exception LexError of string
}
let chiffre = ['0'-'9']
let lettre  = ['a'-'z' 'A'-'Z']
let ident = lettre (lettre | chiffre)*
rule token = parse
    [' ' '\t' '\n'] {token lexbuf}
  | ident           {Ident(Lexing.lexeme lexbuf)}
  | '"'             {chaine lexbuf}
  | "(*"            {comment lexbuf; token lexbuf}
  | eof             {Eof}
and chaine = parse
    '"'             {let s = contents_buffer() in reset_buffer(); String(s)}
  | _               {add_buffer (Lexing.lexeme lexbuf); chaine lexbuf}
  | eof             {raise (LexError "chaîne non fermée")}
and comment = parse
    "*)"            {()}
  | _               {comment lexbuf}
  | eof             {raise (LexError "commentaire non fermé")}
```

1. Modifier l'analyseur lexical afin que l'erreur levée lorsque l'on rencontre la fin du fichier dans une chaîne ou un commentaire contienne également le numéro de caractère du début de la chaîne ou du commentaire qui n'a pas été fermé.
   Pour cela on modifie le type de l'exception `LexError` qui devient :

   ```
   exception LexError of string * int
   ```

2. On suppose que l'analyse lexicale, syntaxique ou sémantique a détecté une erreur au n-ème caractère d'un fichier d'entrée. On veut construire une fonction `locate_error` qui étant

donné un nom de fichier f et un numéro de caractère n permet de calculer les numéros de ligne et de colonne correspondant au n-ème caractère dans f. Par convention, le premier caractère du fichier a pour numéro 0; le premier caractère de chaque ligne est dans la colonne 0 et la première ligne du fichier a pour numéro 1.

(a) Proposer une version de `locate_error` construite à l'aide d'un programme `ocamllex`.

(b) Ecrire une version de `locate_error` qui n'utilise pas d'analyseur lexical mais simplement la fonction `input_char` de type `in_channel` $\rightarrow$ `char`.

3. On souhaite construire une solution qui évite d'analyser une seconde fois un fichier lorsqu'une erreur se produit. Pour cela, on stocke pour chaque nouvelle ligne rencontrée la position absolue dans le fichier du premier caractère de cette ligne associée au numéro de la ligne. Lorsqu'une erreur se produit au caractère n, il suffit alors de retrouver la position du dernier début de ligne avant n (c'est celui dont le numéro est le plus grand parmi les débuts de ligne inférieur à n). Par exemple, dans le fichier :

```
ident
"chaine"
(* commentaire
sur plusieurs lignes *)
```

les débuts de ligne sont placés aux caractères : $0, 6, 15$ et 30. La table des débuts de ligne comportera les éléments suivants : $(0, 1)(6, 2)(15, 3)(30, 4)$.
Pour retrouver la ligne et la colonne correspondant au caractère de position 25, il suffit de retrouver l'entrée qui a le numéro le plus grand parmi les entrées de numéro inférieur à 25. Dans l'exemple, il s'agit de l'entrée $(15, 3)$. Le caractère est donc sur la troisième ligne et le numéro de colonne est $25 - 15 = 10$.

On suppose que l'on dispose d'une table permettant de stocker des couples (n, l) avec n un numéro de caractère et l le numéro de ligne correspondant. On dispose de deux opérations `add` et `max_inf` pour manipuler cette table. Les types et spécifications de ces opérations sont données dans le tableau suivant :

`table_add`	`int` $\rightarrow$ `int` $\rightarrow$ `unit`	`table_add` n l ajoute le couple (n, l) à la table
`max_inf`	`int` $\rightarrow$ `int` $\times$ `int`	`max_inf` n calcule le couple (m, l) tel que m est maximum parmi les couples (m', l') de la table tels que $m' \leq m$

(a) Modifier l'analyseur lexical du langage pour construire la table des débuts de ligne.

(b) Proposer une nouvelle implantation de la fonction `locate_error` utilisant cette table.

4. On se propose d'implanter la structure de la table des débuts de ligne en utilisant une référence sur un arbre binaire de recherche équilibré contenant les couples (n, l) et ordonné suivant la relation $(n, l) < (n', l') \Leftrightarrow n < n'$. On suppose que le type α `btree` implante un type polymorphe d'arbres binaires de recherche stockant aux nœuds des valeurs de type α et que l'on dispose des opérations de base sur ce type spécifiées dans la table suivante :

`bt_add`	$\alpha \rightarrow \alpha$ `btree` $\rightarrow \alpha$ `btree`	`bt_add` v t est un arbre binaire de recherche équilibré contenant toutes les valeurs de l'arbre t plus la valeur v
`bt_empty`	α `btree` $\rightarrow$ `bool`	`bt_empty` t renvoie vrai si t correspond à un arbre vide
`bt_root`	α `btree` $\rightarrow \alpha$	`bt_root` t renvoie la valeur stockée à la racine de t si t n'est pas vide
`bt_left`	α `btree` $\rightarrow \alpha$ `btree`	`bt_left` t renvoie le sous arbre gauche de l'arbre t si t n'est pas vide
`bt_right`	α `btree` $\rightarrow \alpha$ `btree`	`bt_right` t renvoie le sous arbre droit de l'arbre t si t n'est pas vide

À l'aide du type `btree` et des opérations primitives sur ce type, proposer une implantation de la table des débuts de ligne et un codage des opérations `table_add` et `max_inf`.

5. Estimer la complexité de l'opération `locate_error` en fonction du nombre de lignes du fichier d'entrée lorsque le table des débuts de ligne est implantée par un arbre binaire de recherche équilibré.

2.7 Analyse lexicale d'un assembleur

D'après partiel 2009.

Dans cet exercice, on se propose de traiter l'analyse syntaxique d'un programme en langage assembleur à l'aide de `ocamllex`. La syntaxe est inspirée de celle de TIGCC, un assembleur pour les calculatrices TI89. Les principaux éléments de ce langage sont:

Commentaire commence par `/*` et se termine à la première occurrence de `*/` ou bien commence par une barre (`|`) et se termine au premier retour à la ligne. Un commentaire est considéré comme un espace.

Symbole suite de caractères formés des lettres de l'alphabet (majuscule ou minuscule) des chiffres et des trois caractères spéciaux `_`, `.` ou `$` et ne commençant pas par un chiffre.

Constante entière les constantes entières peuvent être données en plusieurs formats:
- binaire : $0b$ ou $0B$ suivi d'une suite de 0 ou de 1
- hexadecimal : $0x$ ou $0X$ suivi d'une suite de nombres hexadecimaux ($0-9$ $A-F$ $a-f$).
- decimal : suite de chiffres ne commençant pas par 0.

Déclaration formée de zéro ou plusieurs labels suivis d'une instruction et terminé par un retour à la ligne ou bien un point-virgule (;). Un label est un symbole immédiatement suivi de deux-points (:) sans espace.

Instruction
- Le langage contient une instruction "vide".
- Une instruction non vide est donnée par son nom (formé uniquement de lettres) suivi de zéro ou plus arguments séparés par des virgules (`,`).
- Un argument d'instruction est soit un registre (de la forme `%`nr avec nr le nom de registre qui est une suite de lettres), soit le contenu de l'adresse stockée dans un registre (noté `%`nr`@`) soit un symbole, soit une constante entière.

1. Proposer un type d'arbre de syntaxe abstraite pour représenter les instructions de ce langage. Les constantes entières seront représentées par des valeurs Ocaml de type `int`.
2. Ecrire un programme `ocamllex` qui analyse un programme en assembleur et renvoie la séquence d'instructions. On pourra utiliser des variables globales pour stocker les labels, nom d'instructions et arguments au fur et à mesure de la reconnaissance d'une déclaration.
3. Modifier le programme précédent (on n'indiquera que les lignes qui changent) pour stocker dans une table les labels en leur associant le numéro d'instruction correspondant. Si le même nom est utilisé à plusieurs reprises, alors on retient la dernière occurrence.
4. Le langage est étendu pour inclure des labels locaux. Ceux-ci sont donnés par une constante entière N suivie du symbole deux-points (`:`). On peut utiliser comme argument d'une instruction assembleur les entrées N`b` ou bien N`f` qui font référence au label N introduit juste avant (backward) l'instruction dans le cas de N`b` ou juste après (forward) dans le cas de N`f`. Ces labels sont transformés immédiatement en des labels ordinaires avec un nom `L`$N\backslash 002i$ avec $\backslash 002$ un caractère spécial pour éviter que ce nom coincide avec un nom de l'utilisateur et i un compteur qui repère le nombre d'utilisations du label local N avant cette instruction.
   Expliquer comment étendre votre analyseur pour prendre en compte cette extension (on ne demande pas d'écrire le code mais de décrire précisément et clairement la méthode).