Analyse lexicale
10 septembre 2010

Rappel

Objectif de l'analyse lexicale et syntaxique

Entrée : suite de caractères
Sortie : arbre de syntaxe abstraite représentant le programme sous-jacent
Structure intermédiaire : suite d'entités lexicales (tokens)

Outils théoriques : expressions régulières, grammaires
Outils logiciels : Lex et Yacc

Principes de l'analyse lexicale

- Chaque entité lexicale est décrite par une expression régulière.
- A chaque entité lexicale est associée une action.
- L'action produit en général un token, mais peut aussi avoir d'autres effets (compter des lignes, ignorer ou transformer le texte).

Sommaire

Expressions régulières

Les constructions de base:

$$
\begin{array}{rcl}
\mathit{regexp} & ::= & \text{'}\mathit{caractère}\text{'} \\
& | & \mathit{regexp}\ \mathit{regexp} \\
& | & \mathit{regexp}\ |\ \mathit{regexp} \\
& | & \mathit{regexp}\ * \\
\end{array}
$$

Dans la théorie, on ajoute aussi des expressions régulières pour:

- le mot vide notée ϵ
- l'ensemble vide notée $\emptyset$

mais qui ne sont pas utilisées dans les analyseurs lexicaux.
Chaque expression e représente un langage (ensemble de mots $L(e)$) qui est dit régulier.

$$
\begin{array}{ll}
L(c) & = \{c\} \\
L(e_1\ e_2) & = \{m_1 m_2 | m_1 \in L(e_1), m_2 \in L(e_2)\} \\
L(e_1|e_2) & = L(e_1) \cup L(e_2) \\
L(e^*) & = \bigcup_{0 \leq n} L(e^n) \qquad e^0 = \epsilon \ \ e^{n+1} = e(e^n)
\end{array}
$$

Expressions régulières étendues

La syntaxe autorisée par ocamllex:

regexp	::=	"*chaîne*"
	|	*regexp* `?`
	|	*regexp* `+`
	|	`[` *caractères* `]`
	|	`[^` *caractères* `]`
	|	`eof`
	|	`_`
	|	`(` *regexp* `)`
	|	*regexp* `as` *ident*
caractères	::=	(`'`*caractère*`'` | `'`*caractère*`'` `-` `'`*caractère*`'`)`+`

Précédence plus haute `+` et `*`, ensuite `?`, puis la concaténation puis le choix `|`.
Exemple: `'a'|'b'* 'c'+`
se lit: `'a'|(('b'*) ('c'+))`

Résultats

- Si e est une expression régulière alors il existe un automate fini (que l'on calcule à partir de e) qui reconnait le langage $L(e)$.
- La réciproque est vraie.
- Le langage des expressions bien parenthésées $\{a^n b^n | n \in \mathbb{N}\}$ n'est pas régulier.

Exercice donner une expression régulière pour les identificateurs (suite de caractères alphabétiques) qui ne sont pas le mot clé **set**.
Construire l'automate correspondant.

Solution

Sommaire

ocamllex

Syntaxe d'un fichier ocamllex

```
  { header }
let ident = regexp ...
rule entrypoint [arg1... argn] =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint [arg1... argn] =
  parse ...
and ...
{ trailer }
```

Résultat de ocamllex

La commande

`ocamllex` *`file.mll`*

1. produit un fichier *file.ml* qui doit ensuite être compilé par ocaml;
2. affiche la taille de l'automate et de la table stockant les transitions.

- Chaque *entrypoint* définit une fonction ocaml.
- Ces fonctions peuvent être paramétrées et ont toujours un argument supplémentaire `lexbuf`.
- Ces fonctions peuvent être appelées récursivement dans les actions.

Principe de fonctionnement de ocamllex

Le fichier ocaml engendré:

```
 Le code ocaml  du prélude
let lex_tables = { ... }
let rec entrypoint lexbuf =
    lex_entrypoint_rec lexbuf 0
and lex_entrypoint_rec lexbuf lex_state =
  match Lexing.engine lex_tables lex_state lexbuf with
    | 0 ->
        ( code de l'action de l'expr reconnue à l'état 0 )
    | 1 ->
        ( code de l'action de l'expr reconnue à l'état 1 )
    ...
    | lex_state -> lexbuf.Lexing.refill_buff lexbuf;
                       lex_entrypoint_rec lexbuf lex_state
 Le code ocaml  du postlude
```

Principe de fonctionnement de ocamllex (2)

- La table de transitions associe à un état et un caractère l'état suivant dans l'automate.
- Le stockage de la table de transitions utilise des techniques de représentation de matrices creuses de manière linéaire dans des chaînes de caractères.
- Des erreurs dans le code ocaml du prélude, des actions ou du postlude ne seront repérées que lors de la compilation de *file.ml* par ocaml.
- Des directives dans le fichier *file.ml* `#18 "test.mll"` permettent de relier les lignes de *file.ml* à celles du fichier ocamllex correspondant.

Un peu d'algorithmique (matrice creuse)

On veut représenter une matrice $M(i, j)$ de taille $n \times m$.

On suppose que de nombreuses cases de M ont une valeur constante z et on cherche à représenter M en évitant de stocker z.

- On utilise un tableau T et on cherche à placer les lignes de M sans les valeurs z dans le tableau T.
- Pour savoir à quelle ligne appartient un élément de T, on lui associe un second tableau C de même taille.

Un peu d'algorithmique (2)

- Pour placer la $i$ème ligne dans T:
  - on cherche une position p la plus petite possible telle que si $M(i,j) \neq d$ alors $T(p+j)$ est vide.
  - On stocke p dans un tableau $d[i]$ de taille n.
  - Pour tout j tel que $M(i,j) \neq d$ on met à jour $T[p+j] := M(i,j)$ et $C[p+j] := i$
- La fonction d'accès à $M(i,j)$ à partir de T, d et C sera vue en TD. Cette fonction a un coût constant.

Exemple

M:

	0	1	2	3	4	5
0	z	z	a	z	b	z
1	a	b	z	z	z	c
2	z	b	z	z	a	z
3	a	z	z	z	a	z
4	b	z	z	z	b	z

d:

0	-2
1	3
2	5
3	1
4	6

	0	1	2	3	4	5	6	7	8	9	10	11	12
T:	a	a	b	a	b	a	b	b	c	a		b	
C:	0	3	0	1	1	3	2	4	1	2		4	

Sommaire

Commentaires

Exercice

- Trouver l'expression régulière correspondant à des commentaires qui débutent par $/*$ et se terminent à la première occurrence de $*/$
- Construire l'automate correspondant

Solution

- Peut-on mettre en commentaire un code qui contient des commentaires ?
- Quel est l'avantage des commentaires qui débutent par un caractère spécial par exemple $\#$ et se terminent par la fin de ligne ?

Commentaires imbriqués

Les analyseurs lexicaux permettent de traiter des commentaires imbriqués. On utilise une deuxième fonction d'analyse chargée de lire jusqu'à la fin du commentaire.

- Possibilité d'avoir un compteur pour marquer le nombre de commentaires restant à fermer.

```
rule nexttoken = parse
   "/*"    {com:=1; comment lexbuf; nexttoken lexbuf}
and comment = parse
   "*/"    {if !com > 1 then
               begin decr com; comment lexbuf end}
 | "/*"    {incr com; comment lexbuf}
 | eof     {raise Error "commentaire non terminé"}
 | _       {comment lexbuf}
```

Commentaires imbriqués (2)

- Utilisation de la pile d'appels

```
  rule nexttoken = parse
   "/*"    {comment lexbuf; nexttoken lexbuf}
and comment = parse
   "*/"    {}
 | "/*"    {comment lexbuf; comment lexbuf}
 | eof     {raise Error "commentaire non terminé"}
 | _       {comment lexbuf}
```

Le cas des chaînes de caractères

- Les expressions régulières pour les chaînes de caractère sont assez complexes.
  Par exemple on écrira `"Nom:\"PP\"\\\n"` pour représenter la chaîne `Nom:"PP"\` terminée par un retour chariot.
- Elles peuvent aussi être arbitrairement longues.
- Il est recommandé de lire les chaînes avec une entrée de l'analyseur qui met les caractères au fur et à mesure dans un buffer.

Sommaire

Localisation des erreurs

- ocamllex conserve dans la variable `lexbuf` le nombre de caractères lus par rapport au début du fichier
  - `Lexing.lexeme_start lexbuf` de type `int` le nombre de caractères jusqu'au début de l'entité reconnue.
  - `Lexing.lexeme_end lexbuf` le nombre de caractères jusqu'à la fin de l'entité reconnue.
- Un type position défini dans `Lexing`.

```
type position =
  pos_fname : string;   (* nom fichier *)
  pos_lnum : int;       (* no de ligne *)
  pos_bol : int;        (* nbre de car entre début de fichier
                           et début de ligne *)
  pos_cnum : int;       (* nbre de car entre début de fichier
                           et position courante *)
```

- Accès à la position: `lexbuf.Lexing.lex_curr_p` de type `position`
  Attention: seul `pos_cnum` est mis à jour automatiquement.

Mise à jour des numéros de ligne

- En prélude de l'analyseur une fonction à appeller à chaque retour chariot:

```
let newline lexbuf =
let pos = lexbuf.lex_curr_p in
lexbuf.lex_curr_p <-
 { pos with pos_lnum = pos.pos_lnum + 1;
            pos_bol = pos.pos_cnum }
```

- Dans l'analyseur:

```
rule nexttoken = parse
      | '\n'     { newline lexbuf; nexttoken lexbuf }
```

- A faire aussi dans les commentaires ...

Sommaire

Autres utilisations des analyseurs lexicaux

Les analyseurs lexicaux sont aussi utiles pour d'autres tâches que la reconnaissance de tokens.
Par exemple pour faire un préprocesseur qui définit des constantes et des directives de compilation (qui peuvent être imbriquées).

```
#define nom valeur
#define A
#ifdef A
dans le cas où A est définie
#else
dans le cas contraire
#endif
```

Les directives sont sur une ligne.

Modélisation du pré-processeur

- Les définitions de macros sont dans une table de hachage:

```
let definitions = Hashtbl.create 97
let define = Hashtbl.add definitions
let defined = Hashtbl.mem definitions
let definition = Hashtbl.find definitions
```

- Une entrée `scan` qui écrit le fichier processé.
- Une entrée `skip` qui va ignorer le texte jusqu'au `#else` ou `#endif` correspondant.
- Une entrée `skip_to_endif` qui va ignorer le texte jusqu'au `#endif` fermant.

Pré-processeur-scan

```
rule scan = parse
  | "#define" space+ (ident as x) space* ([^'\n']* as v) '\n'
       { define x v; scan lexbuf }
  | "#ifdef" space+ (ident as x) space* '\n'
       { if defined x then scan lexbuf else skip lexbuf }
  | "#else" space* '\n'
       { skip lexbuf }
  | "#endif" space* '\n'
       { scan lexbuf }
  | ident as x
       { printf "%s" (if defined x then definition x else x)
          scan lexbuf }
  | _ as c
       { printf "%c" c; scan lexbuf }
  | eof
       { () }
```

Il faudrait traiter spécifiquement les chaînes de caractères et les commentaires.

Pré-processeur-skip

- `skip` ignore l'entrée jusqu'au `#else` ou `#endif` correspondant et continue.
- `skip_to_endif` ignore l'entrée jusqu'au `#endif` correspondant.

```
and skip = parse
  | ("#else" | "#endif") space* '\n'
      { scan lexbuf }
  | "#ifdef" space+ ident space* '\n'
      { skip_to_endif lexbuf; skip lexbuf }
  | _   { skip lexbuf }
  | eof { () }

and skip_to_endif = parse
  | "#endif" space* '\n'
      { () }
  | "#ifdef" space+ ident space* '\n'
      { skip_to_endif lexbuf; skip_to_endif lexbuf }
  | _   { skip_to_endif lexbuf }
  | eof { () }
```

Récapitulatif

- Bien comprendre les conventions lexicales : quelle expression régulière pour quelle entité?
- Définir les unités lexicales : est-ce que la grammaire a besoin de distinguer plusieurs classes ?
  Par exemple pour les opérateurs binaires.
- Quelques “trucs” à connaître:
  - traitement des commentaires
  - traitement des mots clés
  - gestion des numéros de ligne
  - utiliser un analyseur auxilliaire plutôt qu’une expression régulière complexe pour les chaînes, les commentaires . . .

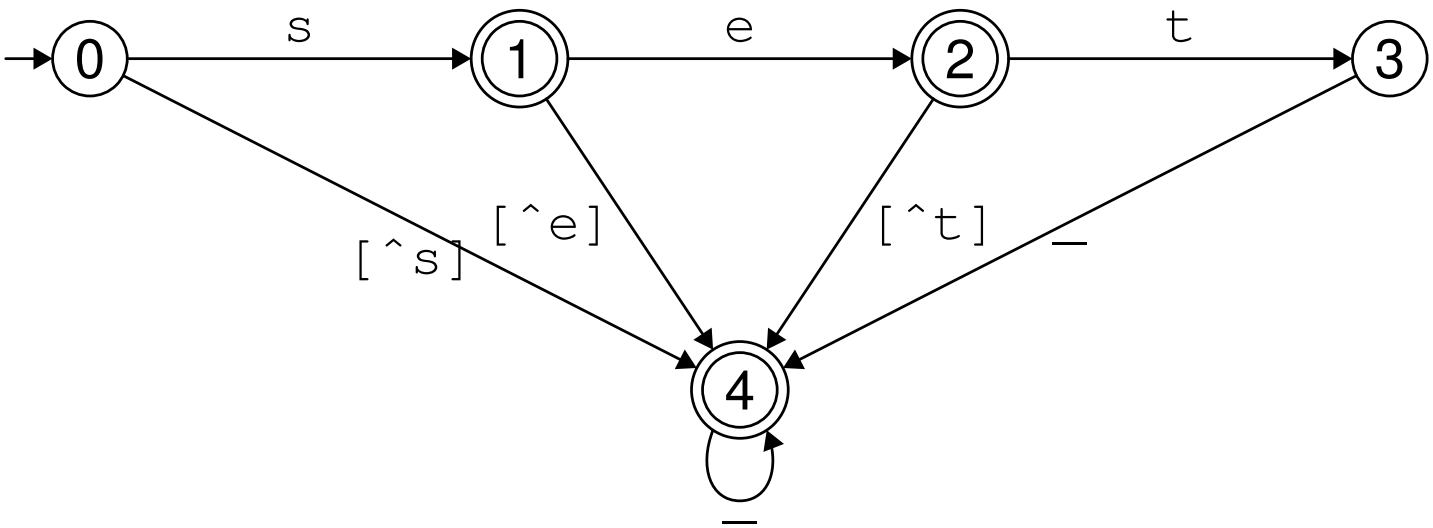
Identifiants différents de **set**

Expression régulière:

```
let alphaok=['a'-'d' 'f'-'r' 'u'-'z' 'A'-'Z']
let alphadig = ['a'-'z' 'A'-'Z' '0'-'9']
let id_not_set =
    (alphaok|'e'|'t') alphadig*
  | 's' | 's' (alphaok|'s'|'t') alphadig*
  | "se" | "se" (alphaok|'s'|'e') alphadig*
  | "set" alphadig+
```

Identifiants différents de **set** (2)

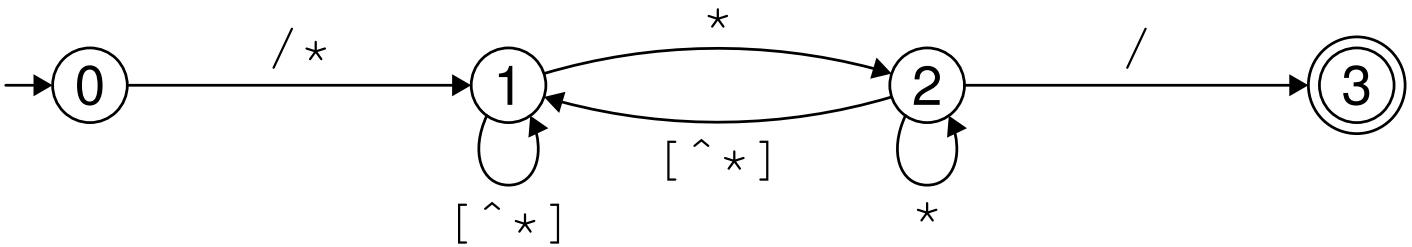
Automate:



Retour

Commentaires

Automate:



Expression régulière: $/*(*^*\,[\hat{}*])^*|(*)^+/$

Retour