Analyse lexicale

Sections 2.6 et 3.1 à 3.4

* Contenu *

- Rôle de l'analyseur lexical
- Vocabulaire
- Gestion des erreurs
- Définitions en théorie des langages
- Reconnaissance des jetons
- Analyse lexicale vorace
- Programmation d'un analyseur lexical
- Exemple d'analyseur lexical plus complet en C
- Outils automatiques: Flex

Extensions aux expressions arithmétiques

Section 2.6

Dans les notes de cours du chapitre 2, on a vu la grammaire qui génère les "suites de chiffres séparés par des signes d'addition ou de soustraction".

$$
\begin{array}{rcl}
\textit{list} & \rightarrow & \textit{list} \texttt{ + } \textit{digit} \\
\textit{list} & \rightarrow & \textit{list} \texttt{ - } \textit{digit} \\
\textit{list} & \rightarrow & \textit{digit} \\
\textit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{array}
$$

En utilisant cette syntaxe telle quelle, il n'y aurait pas vraiment de besoin d'analyse lexicale.

Chaque jeton (terminal) correspond à un seul caractère.

Aussi, rien n'est prévu pour des espacements ou des sauts de ligne.

Les langages réalistes ont tendance à être plus permissifs.

Espacements et commentaires

La plupart des langages permettent la présence d'espacements entre les jetons.

Aussi, la plupart offrent une notation qui permet d'ajouter des commentaires au programme.

Bien que les espacements et les commentaires peuvent possiblement jouer un rôle dans la séparation des jetons au niveau du programme source, le traducteur n'est pas intéressé outre mesure en leur nature.

Ils sont donc rapidement éliminés par les compilateurs. Ils peuvent l'être au niveau de l'analyseur lexical ou au niveau de l'analyseur syntaxique.

C'est normalement beaucoup plus simple de le faire au niveau de l'analyseur lexical.

Constantes à plusieurs chiffres

Par exemple, dans un langage plus réaliste, on remplacerait les constantes numériques à un caractère par des constantes à *au moins* un caractère.

C'est normalement l'analyseur lexical qui s'occupe de rassembler la suite de chiffres en un jeton. C'est plus simple ainsi. De plus, l'analyseur syntaxique ne gagne rien à recevoir les chiffres un à un.

Quand l'analyseur lexical tombe sur une suite de chiffres en entrée, il envoie le jeton **num** à l'analyseur syntaxique. La valeur du nombre est envoyée en tant qu'*attribut* du jeton.

Par exemple, l'entrée:

```
31 + 28 + 59
```

est transformée en la séquence de paires jetons/attributs:

$\langle$**num**, 31$\rangle$ $\langle$+, $\rangle$ $\langle$**num**, 28$\rangle$ $\langle$+, $\rangle$ $\langle$**num**, 59$\rangle$

Identificateurs et mots-clés

On souhaite que l'analyseur lexical transforme l'entrée suivante:

```
count = count + increment;
```

en la suite de jetons suivants:

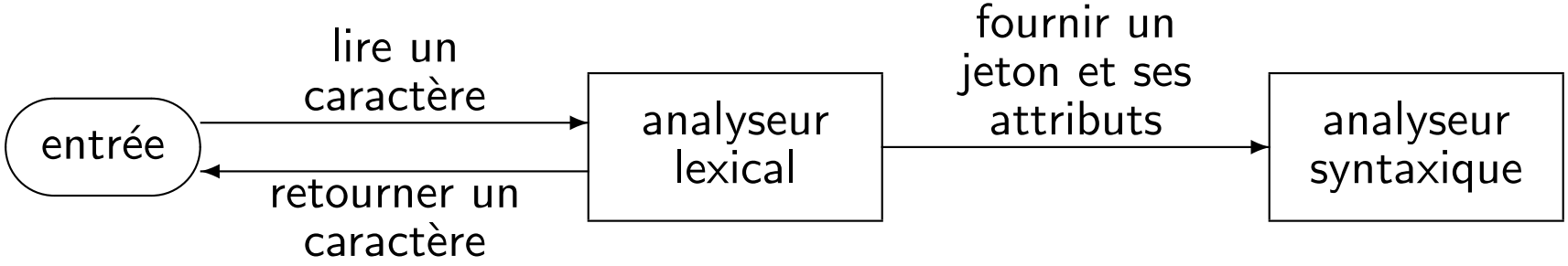
id = **id** + **id** ;

Toutefois, les attributs liés aux deux premières apparitions de **id** doivent indiquer qu'on a affaire à l'identificateur `count` et celui lié à la troisième, à l'identificateur `increment`.

Les identificateurs sont normalement listés dans la *table des symboles*. Les attributs liés aux trois jetons **id** sont alors des pointeurs vers des entrées dans cette table.

Lorsqu'il y a des mots-clés et qu'ils sont *réservés*, les mots-clés sont inscrits au préalable dans la table des symboles. Une note spéciale indique qu'il s'agit de mots-clés réservés et non de simples identificateurs.

Interface avec l'analyseur lexical



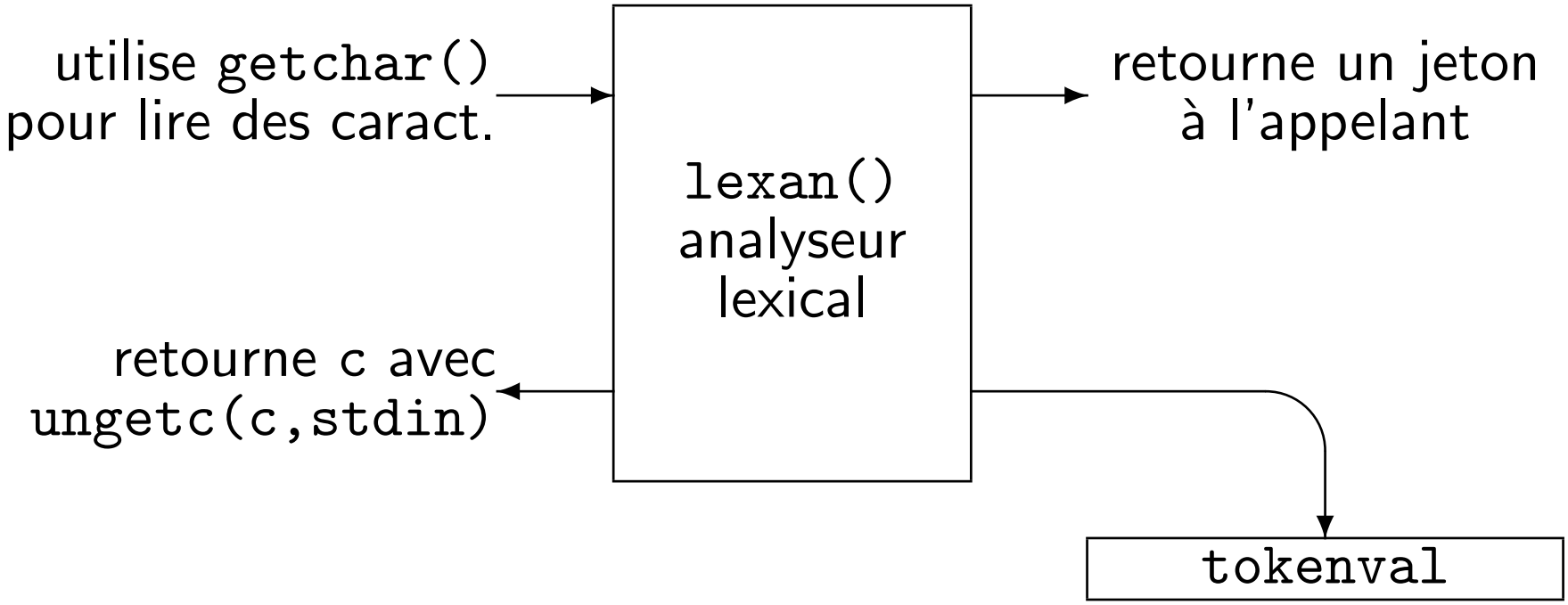
L'analyseur lexical et l'analyseur syntaxique forment une paire *producteur/consommateur*.

L'analyseur lexical doit parfois retourner un caractère qu'il vient de lire. Par exemple, il retourne un caractère lorsqu'il vient de lire '<' et constate que le caractère nouvellement lu n'est pas '='. (Lien avec l'horizon de l'analyseur lexical.)

Le canal entre l'analyseur lexical et l'analyseur syntaxique est un tampon d'une capacité d'un certain nombre de jetons. L'analyseur syntaxique a parfois besoin de consulter les prochains jetons sans les consommer. (Lien avec l'horizon de l'analyseur syntaxique.)

En pratique, ce tampon a rarement besoin d'être de taille plus que 1. Ainsi, l'analyseur lexical est une fonction directement appelée par l'analyseur syntaxique et qui retourne le prochain jeton.

Un analyseur lexical simple en C



Les jetons sont retournés séparément des attributs.

Les jetons sont retournés (à la C) à l'appelant sous la forme d'un entier tandis que les attributs sont mis dans une variable globale (`tokenval`).

Un analyseur lexical simple en C

```
#include <stdio.h>
#include <ctype.h>
#define NUM 256

int lineno = 1;
int tokenval = NONE;

int lexan()
{
  int t;
  while (1) {
    t = getchar();
    if (t == ' ' || t == '\t')
      ;  /* elimine les espaces
            et les tabulations */
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit(t)) {
      tokenval = t - '0';
      t = getchar();
      while (isdigit(t)) {
        tokenval =
          tokenval*10 + t - '0';
        t = getchar();
      }
      ungetc(t, stdin);
      return NUM;
    }
    else {
      tokenval = NONE;
      return t;
    }
  }
}
```

Analyse lexicale

Chapitre 3

Aborde la spécification et l'implantation des analyseurs lexicaux.

Une méthode simple d'implantation d'analyseurs lexicaux consiste à bâtir des diagrammes qui montrent la structure des jetons et ensuite à traduire à la main ces diagrammes en code.

Les techniques utilisées en analyse lexicale s'appliquent non seulement en compilation mais dans les langages de requêtes, les systèmes d'extraction d'information et aussi en génétique.

Toutes ces applications ont en commun la détection de *motifs* dans des chaînes. Un langage de création automatisée d'analyseurs lexicaux, appelé Lex, permet de spécifier le comportement d'un analyseur à l'aide d'expressions régulières. Les expressions régulières sont transformées en un automate fini déterministe efficace.

Plusieurs autres langages emploient des expressions régulières. L'utilitaire AWK reçoit ses instructions notamment grâce à des expressions régulières qui spécifient quelles lignes doivent être traitées et comment. Les interprètes de commandes, tels que `bash` ou `DOS`, reconnaissent des expressions régulières simples comme '`*.pdf`'. Le langage Perl les supporte aussi.

Rôles de l'analyseur lexical

- Lire une suite de caractères de l'entrée et produire une suite de jetons (adéquats pour l'analyse syntaxique).
- Débarrasser le programme des commentaires et des espacements.
- Tenir à jour les coordonnées (ligne, colonne) dans le programme pour fin de production des éventuels messages d'erreur.
- ...

Séparation lexical-syntaxique

Il y a plusieurs raisons pour avoir une séparation entre l'analyseur lexical et l'analyseur syntaxique:

- Simplicité du design.
- Efficacité du compilateur.
- Plus grande portabilité du compilateur. Le jeu de caractères en entrée peut varier d'une machine à l'autre. Une plus petite partie du compilateur est "contaminée" par ces spécificités.

Vocabulaire

Jeton: Catégorie lexicale qui est transmise à l'analyseur syntaxique.

Motif: (*pattern*) Description des chaînes qui peuvent être interprétées comme un jeton donné.

Lexème: Sous-chaîne de l'entrée qui a été reconnue comme étant conforme à un motif. Sert habituellement à calculer le(s) attribut(s) du jeton qui s'en trouve émis.

Jeton	Description du motif	Exemples de lexèmes
const	`const`	`const`
if	`if`	`if`
relation	< ou <= ou = ou <> ou > ou >=	<, <=, =, <>, >, >=
id	une lettre suivie de lettres et de chiffres	`pi`, `count`, `D2`
num	une constante numérique	`3.1416`, `0`, `6.02E23`
literal	des caractères (sauf ") entre " et "	`"core dumped"`

Gestion des erreurs

Certaines erreurs sont véritablement de nature lexicale. Par exemple, rencontrer le caractère ASCII numéro 14, qui ne doit jamais apparaître dans les programmes sources d'un langage donné.

Plusieurs erreurs ne peuvent pas être géré au niveau de l'analyseur lexical. Par exemple, on retrouve dans le programme source la chaîne:

```
fi ( a == f(x) ) ...
```

Cette erreur nous semble de nature lexicale à cause de la faute d'orthographe mais '`fi`' constitue un identificateur tout à fait valide.

Gestion des erreurs

Il y a plusieurs moyens de gérer une erreur lexicale:

- Arrêter toute la compilation avec un message d'erreur.
- Abandonner des caractères de l'entrée jusqu'à ce qu'un jeton bien formé soit disponible.
- Effectuer une ou plusieurs opérations d'édition comme:
  - Effacer un caractère. (Assure la terminaison.)
  - Insérer un caractère [adéquat].
  - Remplacer un caractère par un autre.
  - Inverser l'ordre de deux caractères consécutifs.
- Trouver une distance d'édition minimale pour obtenir un programme lexicalement valide à partir du programme source.

Un avantage à tenter de recouvrer un état stable après une erreur lexicale consiste à possiblement fournir une liste d'erreurs plus complète pour le programme.

Un inconvénient consiste à possiblement confondre les étapes suivantes de la compilation et ainsi générer des erreurs de compilation incongrues.

Analyse Lexicale :

Théorie des langages

Définitions en théorie des langages

Alphabet: ensemble fini de symboles.

Chaîne: suite finie de symboles tirés d'un alphabet donné. Parfois, on dit aussi *mot*.

Longueur d'une chaîne: nombre de symboles qui constitue la chaîne. On dénote la longueur d'une chaîne w par $|w|$.

Chaîne vide: chaîne de longueur 0, qu'on dénote ϵ.

Langage: Étant donné un alphabet Σ, un ensemble de chaînes toutes tirées de Σ.

Concaténation: On dénote la concaténation de deux chaînes w et x par wx et parfois par $w \cdot x$, lorsqu'on veut un opérateur visible.

Exponentiation: Répétition d'une même concaténation, dénotée w^n où $n \in \mathbb{N}$. On la définit comme: $w^0 = \epsilon$ et $w^{(n+1)} = w \cdot w^n$.

Définitions en théorie des langages

Terme	Définition
préfixe de s	Une chaîne w telle que $\exists\ v$ tel que $wv = s$.
suffixe de s	Une chaîne w telle que $\exists\ v$ tel que $vw = s$.
sous-chaîne de s	Une chaîne w telle que $\exists\ u, v$ tel que $uwv = s$.
préfixe, suffixe ou sous-chaîne *propre* de s	Une chaîne w, $s \neq w \neq \epsilon$, telle que w est un préfixe, suffixe ou sous-chaîne de s, respectivement.
sous-séquence de s	Une chaîne w obtenue de s en laissant tomber certains des symboles mais en conservant leur ordre.

Opération	Définition
union de L et M	$L \cup M = \{s \mid s \in L \text{ ou } s \in M\}$
concaténation de L et M	$LM = \{st \mid s \in L \text{ et } t \in M\}$
fermeture de Kleene de L	$L^* = \bigcup_{i=0}^{\infty} L^i$
fermeture positive de L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Définitions en théorie des langages

Les *expressions régulières* peuvent être définies inductivement ainsi. Supposons que r et s sont des expressions régulières, alors les termes suivants sont aussi des expressions régulières:

- $t = \emptyset$, tel que $L(t) = \emptyset$;
- $t = \epsilon$, tel que $L(t) = \{\epsilon\}$;
- $t = a$ où $a \in \Sigma$, tel que $L(t) = \{a\}$;
- $t = r \mid s$, tel que $L(t) = L(r) \cup L(s)$;
- $t = rs$, tel que $L(t) = L(r)L(s)$;
- $t = r^*$, tel que $L(t) = (L(r))^*$;
- $t = (r)$, tel que $L(t) = L(r)$.

On dit qu'un langage L est *régulier* s'il existe une expression régulière r telle que $L(r) = L$.

Il faut utiliser des parenthèses lorsqu'on veut affecter l'ordre d'application des opérateurs sachant qu'ils sont ordonnés ainsi: *, $\cdot$ et $|$, du plus prioritaire au moins prioritaire.

Il existe des langages qui ne sont pas réguliers. Par exemple, $\{ww \mid w \in \{a, b\}^*\}$ est irrégulier.

Deux expressions régulières r et s sont dites *équivalentes* si $L(r) = L(s)$.

Définitions en théorie des langages

Lois
$r \mid r \equiv r$
$r \mid s \equiv s \mid r$
$r \mid (s \mid t) \equiv (r \mid s) \mid t$
$r(st) \equiv (rs)t$
$r(s \mid t) \equiv rs \mid rt$
$(r \mid s)t \equiv rt \mid st$
$\epsilon r \equiv r\epsilon \equiv r$
$r^* \equiv (r \mid \epsilon)^*$
$r^{**} \equiv r^*$
$r \mid \emptyset \equiv \emptyset \mid r \equiv r$
$r\emptyset \equiv \emptyset r \equiv \emptyset$
$\emptyset^* \equiv \epsilon$

Définitions en théorie des langages

Une *définition régulière*, étant donné Σ, est une suite de définitions de la forme:

$$
\begin{aligned}
&d_1 \rightarrow r_1 \\
&d_2 \rightarrow r_2 \\
&\ldots \\
&d_n \rightarrow r_n
\end{aligned}
$$

où r_i est une expression régulière sur l'alphabet $\Sigma \cup \{d_1, \ldots, d_{i-1}\}$.

Note: il ne faut pas confondre les définitions régulières avec les grammaires hors-contexte.

Définitions en théorie des langages

Exemple 3.5: Les identificateurs en C sont des chaînes de lettres, chiffres et *underscore*s.

$$\begin{aligned}
\textbf{letter_} &\rightarrow \texttt{A} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{_} \\
\textbf{digit} &\rightarrow \texttt{0} \mid \ldots \mid \texttt{9} \\
\textbf{id} &\rightarrow \textbf{letter_} \; (\; \textbf{letter_} \mid \textbf{digit} \;)^*
\end{aligned}$$

Exemple 3.6: Les nombres entiers ou fractionnaires non signés sont des chaînes comme `5280`, `0.01234`, `6.336E4` ou `1.89E-4`

$$\begin{aligned}
\textbf{digit} &\rightarrow \texttt{0} \mid \ldots \mid \texttt{9} \\
\textbf{digits} &\rightarrow \textbf{digit digit}^* \\
\textbf{optionalFraction} &\rightarrow \texttt{.} \; \textbf{digits} \mid \epsilon \\
\textbf{optionalExponent} &\rightarrow (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \textbf{digits} \;) \mid \epsilon \\
\textbf{number} &\rightarrow \textbf{digits optionalFraction optionalExponent}
\end{aligned}$$

Définitions en théorie des langages

Voici quelques abréviations:

- $r^+ \equiv rr^*$
- $r^? \equiv r \mid \epsilon$
- $[\mathtt{abc}] \equiv \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$
- $[\mathtt{a}-\mathtt{z}] \equiv \mathtt{a} \mid \ldots \mid \mathtt{z}$
- $[\hat{}\mathtt{a}-\mathtt{z}] \equiv$ n'importe quel caractère sauf $\mathtt{a}, \ldots, \mathtt{z}$
- $[\mathtt{A}-\mathtt{Za}-\mathtt{z\_}] \equiv [\mathtt{A}-\mathtt{Z}] \mid [\mathtt{a}-\mathtt{z}] \mid [\mathtt{\_}]$
- $[\hat{}\mathtt{A}-\mathtt{Za}-\mathtt{z\_}] \equiv$
  n'importe quel caractère sauf ceux dans $[\mathtt{A}-\mathtt{Za}-\mathtt{z\_}]$

Analyse Lexicale :

Reconnaissance des jetons

Reconnaissance des jetons

Notre but est d'être capable de reconnaître les jetons pour une certaine grammaire.

Voici le fragment de grammaire:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
 & | & \epsilon \\
\textit{expr} & \rightarrow & \textit{term} \textbf{ relop } \textit{term} \\
 & | & \textit{term} \\
\textit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{num}
\end{array}
$$

et voici les définitions régulières qui génèrent les terminaux qui nous intéressent:

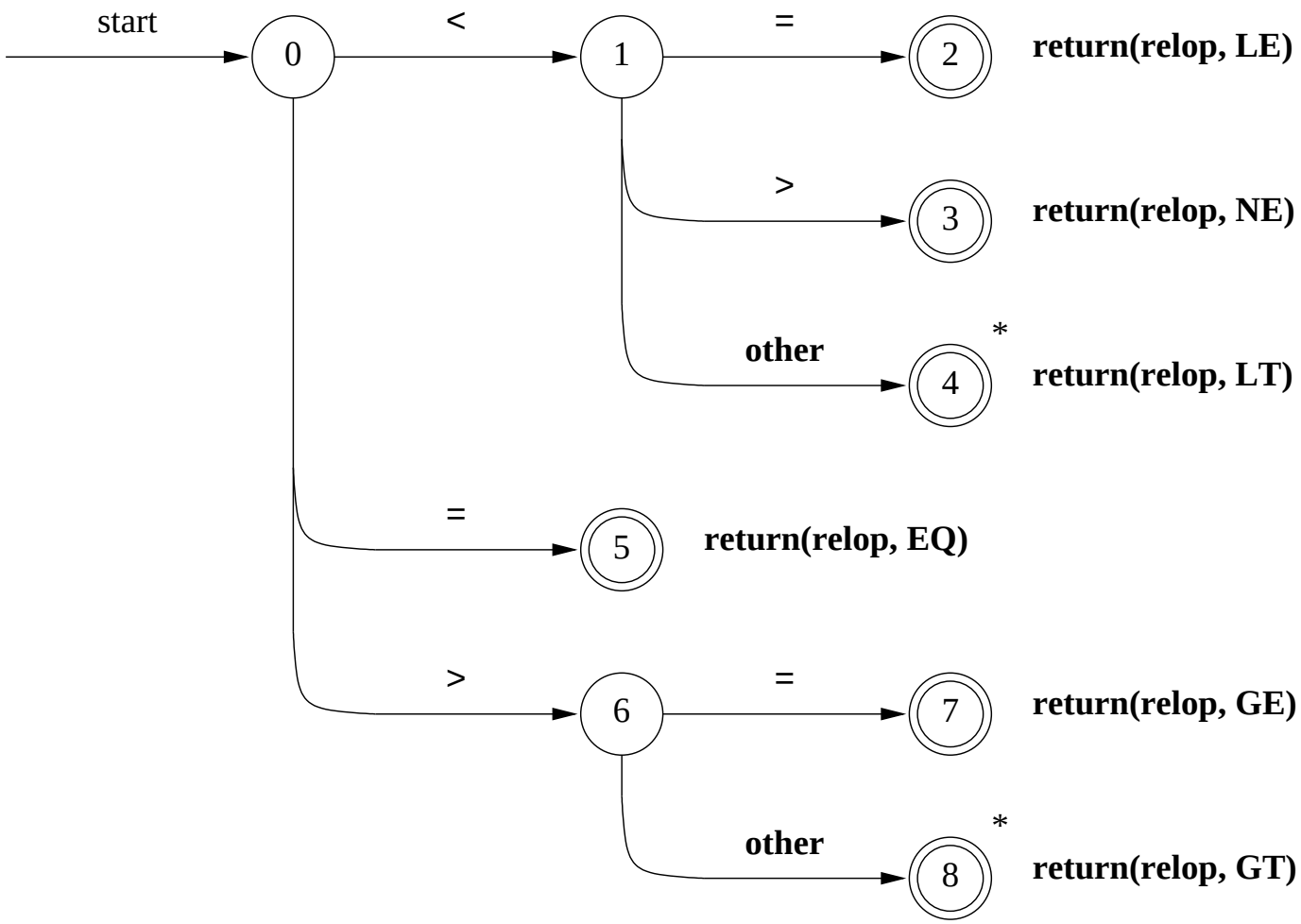
$$
\begin{array}{rcl}
\textbf{if} & \rightarrow & \texttt{if} \\
\textbf{then} & \rightarrow & \texttt{then} \\
\textbf{else} & \rightarrow & \texttt{else} \\
\textbf{relop} & \rightarrow & \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \\
\textbf{id} & \rightarrow & \textbf{letter} \left(\textbf{letter} \mid \textbf{digit} \right)^{*} \\
\textbf{num} & \rightarrow & \textbf{digit}^{+} \left(\texttt{.} \ \textbf{digit}^{+} \right)^{?} \left(\texttt{E} \left(\texttt{+} \mid \texttt{-} \right)^{?} \textbf{digit}^{+} \right)^{?} \\
\textbf{delim} & \rightarrow & \textbf{blank} \mid \textbf{tab} \mid \textbf{newline} \\
\textbf{ws} & \rightarrow & \textbf{delim}^{+}
\end{array}
$$

Reconnaissance des jetons

Expression régulière	Jeton	Attribut
ws	-	-
if	**if**	-
then	**then**	-
else	**else**	-
id	**id**	pointeur vers la table des symb.
num	**num**	valeur de la constante
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

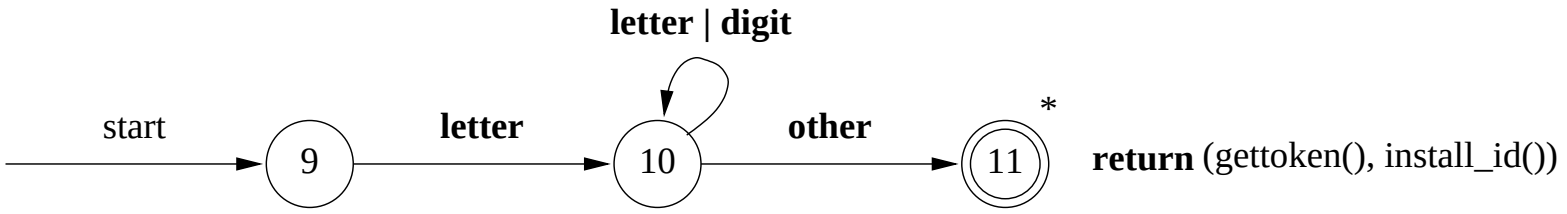
Reconnaissance des jetons

Opérateurs relationnels



Reconnaissance des jetons

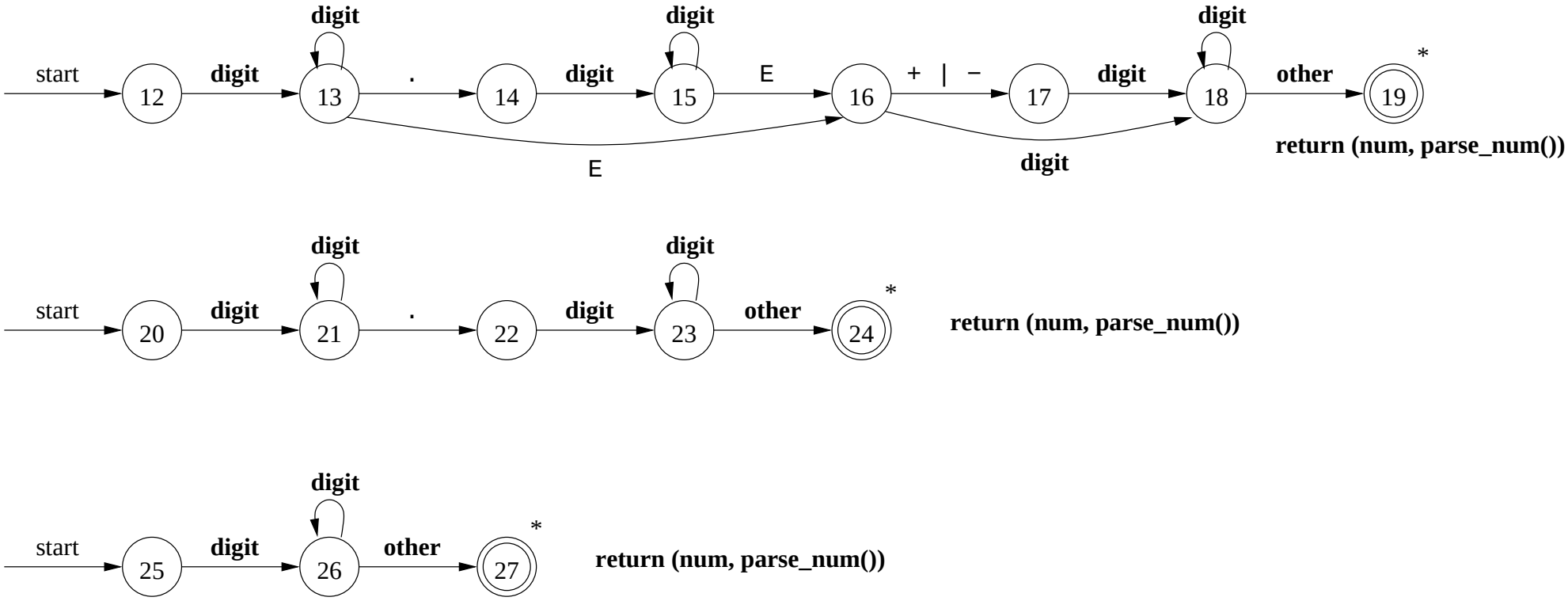
Identificateurs et mots-clés



Les mots-clés sont préinscrits dans la table des symboles.
Celle-ci contient la catégorie lexicale de chacun.

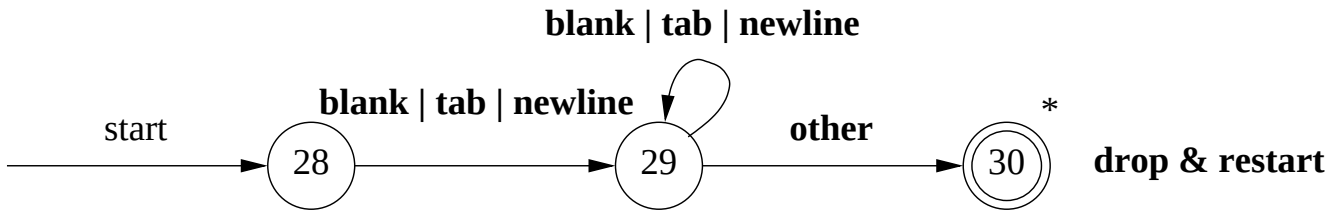
Reconnaissance des jetons

Constantes numériques



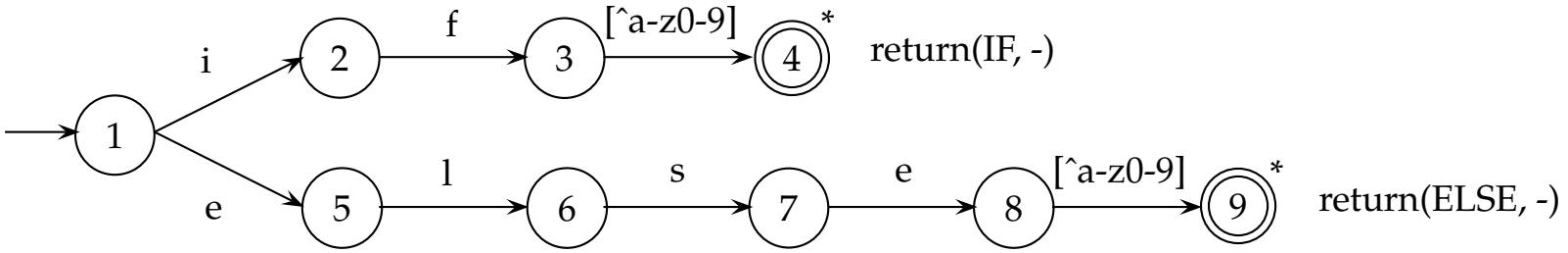
Reconnaissance des jetons

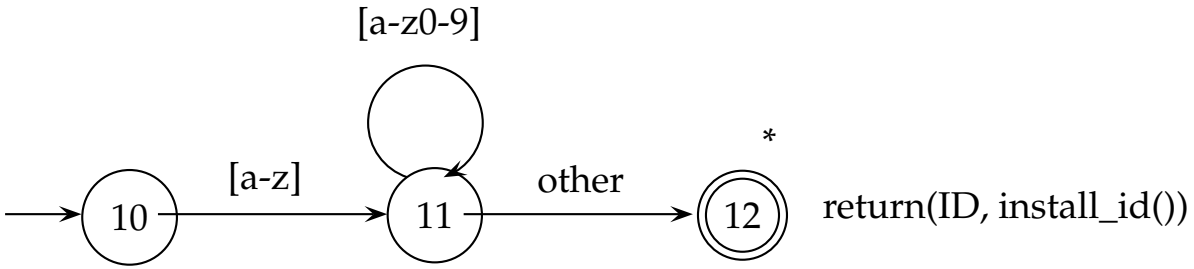
Espaces blancs



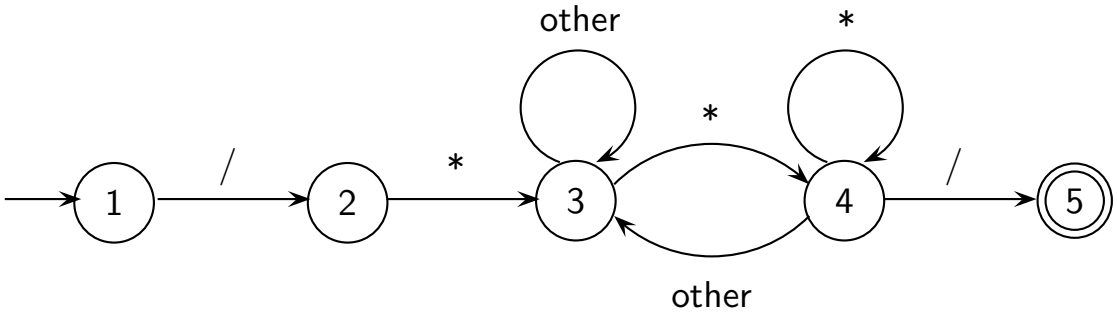
Exemples supplémentaires

Exemple: Traitement des mots-clés (`if` et `else`) par des automates.





Exemple: Commentaires du langage C.



Reconnaissance des jetons

Il y a quelques principes à observer:

- un état acceptant ne consomme pas de caractères;
- un état non-acceptant consomme (habituellement) des caractères;
- chaque arc ne permet de consommer qu'un caractère à la fois, possiblement sélectionné parmi plusieurs possibilités;
- les possibilités offertes par deux arcs sortant du même état sont disjointes;
- un état acceptant peut avoir au plus une étoile;
- les automates doivent s'assurer de consommer le plus possible de caractères à chaque jeton reconnu (analyse lexicale *vorace* ou *maximal-munch tokenization*).

Analyse lexicale vorace (maximal-munch tokenization)

Lors de la reconnaissance d'un jeton, il faut lire le nombre maximum de caractères avant d'accepter.

Exemple : Identificateur `ma_variable`

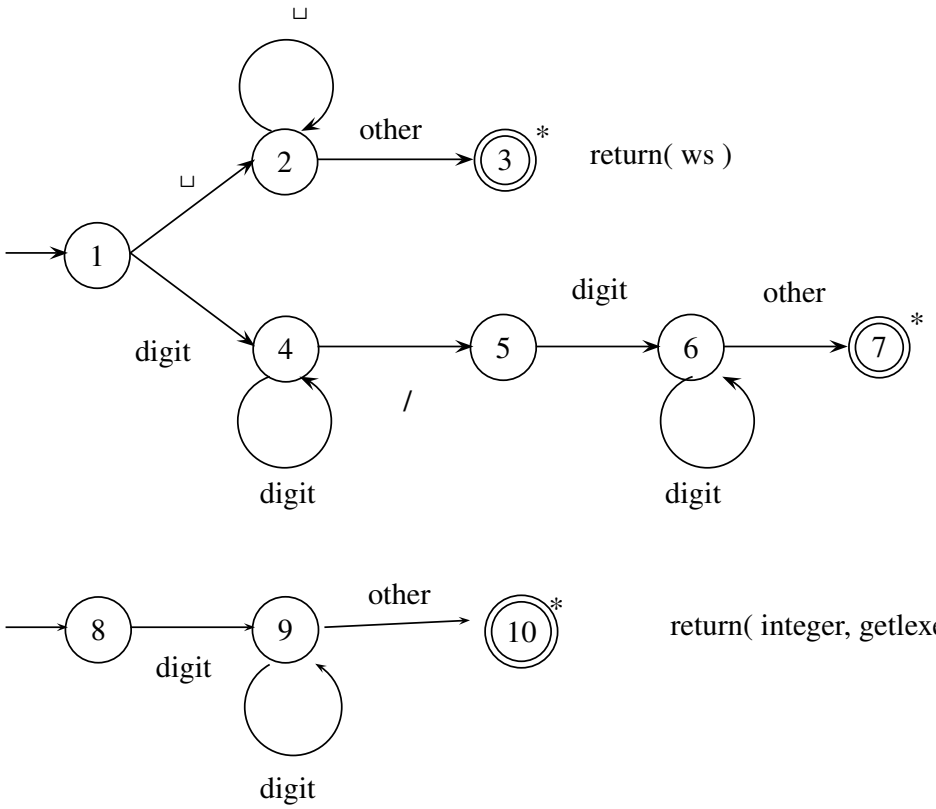
> L'analyseur lexical doit avoir lu l'identificateur en entier avant de l'accepter; par exemple, il ne faut pas atteindre un état acceptant après n'avoir lu que la première lettre (`m` serait un identificateur valide à lui seul, mais ce n'est pas la signification attendue dans un morceau de code contenant l'identificateur `ma_variable`).

Exemple : Nombre décimal `7.2`

> Il ne faudrait pas se contenter de lire `7` puis accepter; bien que `7` soit un jeton "*constante numérique*" valide, ce n'est pas l'interprétation attendue.

Exemples supplémentaires

Exemple: Entiers et rationnels:



Jeton	Description du motif
ws	$␣^+$
rational	$[0-9]^+/[0-9]^+$
integer	$[0-9]^+$

Notez qu'il faut trouver un caractère "other" (non-digit) avant d'accepter le jeton : c'est le principe du **maximal-munch tokenization**. Exemple d'entrée: `37/s`.

Analyse Lexicale :

Programmation d'un analyseur lexical

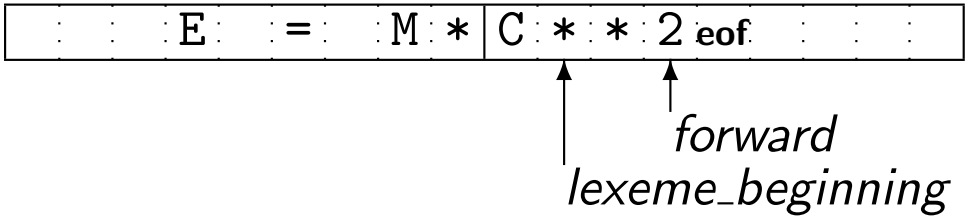
Programmation d'un analyseur lexical

Pour implanter un analyseur lexical, on peut:

- Utiliser un générateur d'analyseur lexical.
- Écrire soi-même l'analyseur dans un langage de haut niveau.
- Écrire soi-même l'analyseur en assembleur et gérer la lecture des caractères efficacement.

Programmation d'un analyseur lexical

Dans une implantation par soi-même, les auteurs du manuel suggèrent d'utiliser un tampon divisé en deux moitiés d'une taille correspondant à la taille d'un bloc sur disque.



À chaque fois qu'on arrive à la fin d'une des moitiés, on lit un bloc en entrée et on l'installe dans l'autre moitié pour pouvoir continuer l'analyse.

Ils proposent deux façons de détecter la fin de tampon: tout simplement en comparant le pointeur avec l'adresse de la fin du tampon; en ajoutant un caractère sentinelle à la fin du tampon, lequel est traité spécialement.

Ces zones tampons posent une limite à la longueur des lexèmes et à l'horizon de l'analyseur lexical.

Programmation d'un analyseur lexical

Les diagrammes de transitions peuvent être convertis en code de manière relativement systématique.

À chaque état, on associe un bout de code. Le bout de code correspondant à un état est rendu accessible en tant qu'alternative ('`case`') à l'intérieur d'un énoncé de sélection ('`switch`'). Enfin, l'énoncé '`switch`' est placé dans une boucle infinie ('`while (TRUE)`').

Les bouts de code utilisent les pointeurs `token_beginning` et `forward` et les variables entières `start` et `state`. Les pointeurs `token_beginning` et `forward` servent à la gestion du tampon d'entrée (voir page 37). La variable `start` indique l'état de départ du diagramme courant. La variable `state` indique l'état courant.

Programmation d'un analyseur lexical

La reconnaissance d'un jeton consiste à initialiser `start` et `state` à l'état de départ du premier diagramme et à exécuter successivement le code associé à chacun des états rejoints jusqu'à ce qu'un succès ou une erreur lexicale soit atteint.

- Si un état (`state`) indique un succès, son code comporte les éléments suivants:
  1. si l'état comporte une étoile, renvoyer le dernier caractère lu;
  2. déterminer le jeton et ses attributs, possiblement en fonction du lexème reconnu, lequel se trouve entre les pointeurs `token_beginning` et `forward`;
  3. faire avancer le pointeur `token_beginning` jusqu'à la position de `forward`.
- Sinon, son code comporte les éléments suivants:
  1. lire le prochain caractère, ce qui fait avancer le pointeur `forward`;
  2. si le caractère lu correspond à l'étiquette d'un des arcs, effectuer une transition à l'état qui est à l'autre bout de l'arc (modifier `state`);
  3. sinon, appeler la fonction `fail` pour déterminer le prochain état.

Programmation d'un analyseur lexical

Chaque appel à la fonction `fail` fait d'abord reculer le pointeur `forward` jusqu'à la position de `token_beginning`. Ensuite, la variable `start` est consultée pour identifier le diagramme courant. En fonction du diagramme courant, `start` et `state` sont affectées à l'état de départ du prochain diagramme.

Si la fonction `fail` est appelée alors que le dernier diagramme était simulé, une erreur lexicale est signalée.

À titre d'exemple, la figure 3.18 présente un extrait du code de la fonction qui contient les bouts de code associés à tous les états et la section 3.4.4 décrit la fonction `fail`.

Programmation d'un analyseur lexical

Code pour un diagramme: diagramme des identificateurs (voir page 28)

```
while (true)
{
  switch(state)
  {
    ...
    case 9:
      c := buffer[forward];
      forward ++;
      if (is_letter(c))
        state := 10;
      else
        state := fail();
      break;
    case 10:
      c := buffer[forward];
      forward ++;
      if (is_letter(c) OR is_digit(c))
        state := 10;
      else
        state := 11;
      break;
    case 11:
      forward --;
      jeton := gettoken();
      attribut := install_id();
      token_beginning := forward;
      return jeton;
    ...
  }
}
```

Programmation d'un analyseur lexical

Code pour tous les diagrammes (voir pages 27–30)

```
int lexical_analyzer()
{
  start := 0;  // -> 1er diagramme
  state := start;
  forward := token_beginning;
  while (true)
  {
    switch(state)
    {
      case 0:  // Diagramme RELOP
        c := buffer[forward];
        forward ++;
        if (c == '<')
          state := 1;
        else if (c == '=')
          state := 5;
        else if (c == '>')
          state := 6;
        else
          state := fail();
        break;
      case 1:
        c := buffer[forward];
        forward ++;
        if (c == '=')
          state := 2;
        else if (c == '>')
          state := 3;
        else
          state := 4;
        break;
      case 2:
        jeton := RELOP;
        attribut := LE;
        token_beginning := forward;
        return jeton;
      case 3:
        jeton := RELOP;
        attribut := NE;
        token_beginning := forward;
        return jeton;
      case 4:
        forward --;
        jeton := RELOP;
        attribut := LT;
        token_beginning := forward;
        return jeton;
      case 5:
        jeton := RELOP;
        attribut := EQ;
        token_beginning := forward;
        return jeton;
      case 6:
        c := buffer[forward];
        forward ++;
        if (c == '=')
          state := 7;
        else
          state := 8;
        break;
      case 7:
        jeton := RELOP;
        attribut := GE;
        token_beginning := forward;
        return jeton;
      case 8:
        forward --;
        jeton := RELOP;
        attribut := GT;
        token_beginning := forward;
        return jeton;
      case 9:  // Diagramme ID
        c := buffer[forward];
        forward ++;
        if (is_letter(c))
          state := 10;
        else
          state := fail();
        break;
      case 10:
        c := buffer[forward];
        forward ++;
        if (is_letter(c) ||
            is_digit(c))
          state := 10;
        else
          state := 11;
        break;
      case 11:
        forward --;
        jeton := gettoken();
        attribut := install_id();
        token_beginning := forward;
        return jeton;
      case 12:  // 1er diagramme NUM
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 13;
        else
          state := fail();
        break;
      case 13:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 13;
        else if (c == '.')
          state := 14;
        else if (c == 'E')
          state := 16;
        else
          state := fail();
        break;
      case 14:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 15;
        else
          state := fail();
        break;
      case 15:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 15;
        else if (c == 'E')
          state := 16;
        else
          state := fail();
        break;
      case 16:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 18;
        else if ((c == '+') ||
                 (c == '-'))
          state := 17;
        else
          state := fail();
        break;
      case 17:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 18;
        else
          state := fail();
        break;
      case 18:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 18;
        else
          state := 19;
        break;
      case 19:
        forward --;
        jeton := NUM;
        attribut := parse_num();
        token_beginning := forward;
        return jeton;
      case 20:  // 2e diagramme NUM
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 21;
        else
          state := fail();
        break;
      case 21:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 21;
        else if (c == '.')
          state := 22;
        else
          state := fail();
        break;
      case 22:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 23;
        else
          state := fail();
        break;
      case 23:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 23;
        else
          state := 24;
        break;
      case 24:
        forward --;
        jeton := NUM;
        attribut := parse_num();
        token_beginning := forward;
        return jeton;
      case 25:  // 3e diagramme NUM
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 26;
        else
          state := fail();
        break;
      case 26:
        c := buffer[forward];
        forward ++;
        if (is_digit(c))
          state := 26;
        else
          state := 27;
        break;
      case 27:
        forward --;
        jeton := NUM;
        attribut := parse_num();
        token_beginning := forward;
        return jeton;
      case 28:  // Diagramme WS
        c := buffer[forward];
        forward ++;
        if ((c == ' ') ||
            (c == '\t') ||
            (c == '\n'))
          state := 29;
        else
          state := fail();
        break;
      case 29:
        c := buffer[forward];
        forward ++;
        if ((c == ' ') ||
            (c == '\t') ||
            (c == '\n'))
          state := 29;
        else
          state := 30;
        break;
      case 30:
        forward --;
        // Aucun jeton produit
        token_beginning := forward;
        start := 0;
        state := start;
        break;
    }
  }
}

int fail()
{
  forward := token_beginning;
  switch (start)
  {
    case 0:
      start := 9;
      break;
    case 9:
      start := 12;
      break;
    case 12:
      start := 20;
      break;
    case 20:
      start := 25;
      break;
    case 25:
      start := 28;
      break;
    default:
      lexical_error();
      break
  }
  return start;
}
```

Analyse Lexicale :

Outils automatisés : Flex

Flex

Il existe des outils permettant de générer automatiquement un analyseur lexical à partir de définitions régulières. Flex (http://flex.sf.net/) génère un analyseur lexical à partir d'un fichier de configuration en 3 sections (séparées par "%%") :

```
%{ // Du code à inclure dans le résultat
   #include <stdio.h>
%}

/* Définitions régulières */
DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+            { printf( "Jeton NombreEntier<%s>\n", yytext );     }
{DIGIT}+"."{DIGIT}* { printf( "Jeton NombreDecimal<%s>\n", yytext );    }
if|while|return     { printf( "Jeton MotCle<%s>\n", yytext );           }
{ID}                { printf( "Jeton Identificateur<%s>\n", yytext );   }
"+"|"-"|"*"|"/"     { printf( "jeton Operator<%s>\n", yytext );         }
"="|"<"|">"|"!="    { printf( "jeton Comparaison<%s>\n", yytext );      }
"("                 { printf( "jeton ParentheseGauche<%s>\n", yytext ); }
")"                 { printf( "jeton ParentheseDroite<%s>\n", yytext ); }
[ \t\n]+            /* Ignorer les caractères d'espacement */
.                   { printf( "Erreur lexicale: %s\n", yytext );        }

%%

// Du code à insérer dans le fichier généré, ici nous lisons un fichier
int main() {
    yyin = fopen( "input.txt", "r" ); yylex(); return 0;
}
```

Flex

Un fichier de configuration de Flex contient trois sections.

- **Première section**:
  - du code à insérer au début du fichier C généré (habituellement des includes);
  - des expressions régulières à l'aide d'une syntaxe similaire à celle vue en classe.
- **Seconde section**:
  - le patron de chaque jeton à reconnaitre (le patron peut utiliser les définitions régulières définies dans la première section avec des accolades);
  - le code à exécuter pour chaque type de jeton, où "`yytext`" est le lexème (dans cet exemple on se contente de l'imprimer avec printf; dans un vrai compilateur il faudrait plutôt transmettre le jeton à l'analyseur syntaxique).
- **Troisième section**: permet d'ajouter du code arbitraire (des fonctions utilitaires, par exemple).

Flex

Une fois le fichier de configuration prêt, il suffit d'invoquer flex sur le terminal:

```
flex --outfile=MonAnalyseurLexical.c  Configuration.yy
```

Cette commande générera le fichier "MonAnalyseurLexical.c", qui sera alors prêt à être compilé et exécuté afin d'effectuer l'analyse lexicale.

Note : si plusieurs patrons définis reconnaissent une entrée (par exemple '<' vs '<='), Flex choisit :

1. le patron qui reconnait le mot le plus long (maximal-munch);
2. (dans le cas de mots de longueur égale) l'ordre dans lequel les patrons sont déclarés dans le fichier.

Flex

Liste d'opérateurs dans Flex:

regularexpressions

Anchors	
^	Start of line +
\A	Start of string +
$	End of line +
\Z	End of string +
\b	Word boundary +
\B	Not word boundary +
\<	Start of word
\>	End of word

Character Classes	
\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\xhh	Hexadecimal character hh
\Oxxx	Octal character xxx

POSIX Character Classes	
[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions	
?=	Lookahead assertion +
?!	Negative lookahead +
?<=	Lookbehind assertion +
?!= or ?<!	Negative lookbehind +
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Note Items marked + should work in most regular expression implementations.

Sample Patterns	
([A-Za-z0-9-]+)	Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})	Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	jpg, gif or png image
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Any number from 1 to 50 inclusive
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	8 to 15 character string with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
(\<(/?[^\>]+)\>)	HTML Tags

Note *These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.*

Quantifiers	
*	0 or more +
*?	0 or more, ungreedy +
+	1 or more +
+?	1 or more, ungreedy +
?	0 or 1 +
??	0 or 1, ungreedy +
{3}	Exactly 3 +
{3,}	3 or more +
{3,5}	3, 4 or 5 +
{3,5}?	3, 4 or 5, ungreedy +

Special Characters	
\	Escape Character +
\n	New line +
\r	Carriage return +
\t	Tab +
\v	Vertical tab +
\f	Form feed +
\a	Alarm
[\b]	Backspace
\e	Escape
\N{name}	Named Character

String Replacement (Backreferences)	
$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string
$_	Entire input string
$$	Literal "$"

Ranges	
.	Any character except new line (\n) +
(a|b)	a or b +
(...)	Group +
(?:...)	Passive Group +
[abc]	Range (a or b or c) +
[^abc]	Not a or b or c +
[a-q]	Letter between a and q +
[A-Q]	Upper case letter + between A and Q +
[0-7]	Digit between 0 and 7 +
\n	nth group/subpattern +

Note *Ranges are inclusive.*

Pattern Modifiers	
g	Global match
i	Case-insensitive
m	Multiple lines
s	Treat string as single line
x	Allow comments and white space in pattern
e	Evaluate replacement
U	Ungreedy pattern

Metacharacters (must be escaped)		
^	[	.
$	{	*
(	\	+
)	|	?
<	>	

Available free from AddedBytes.com