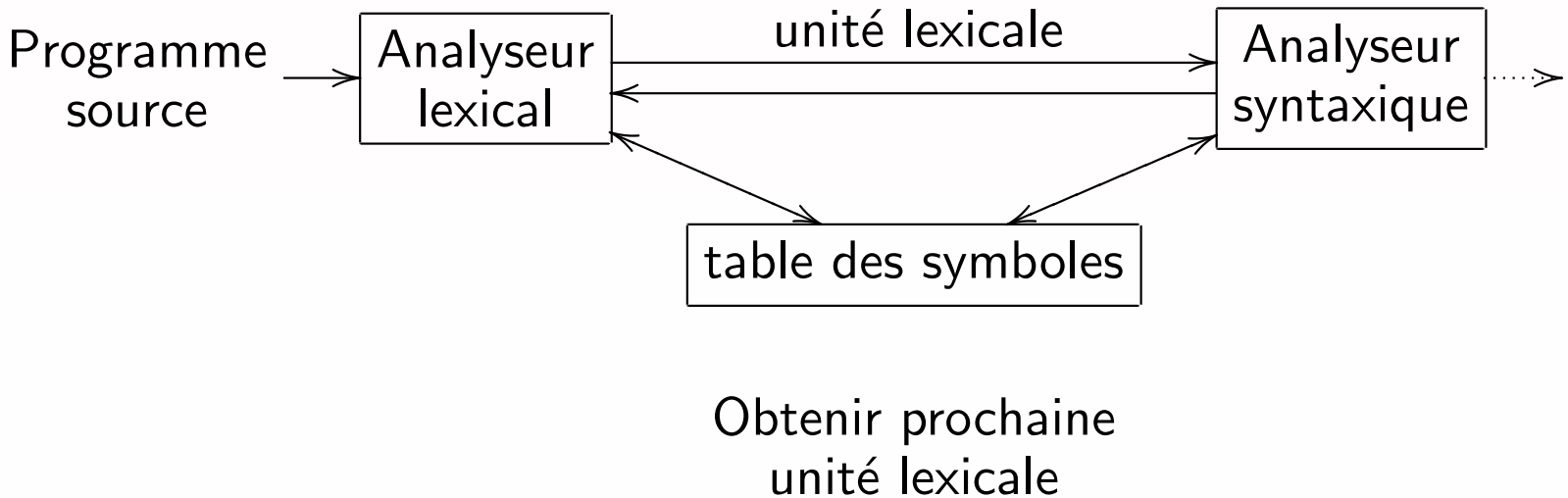
L'analyseur lexical constitue la première phase d'un compilateur. Sa tâche principale est de lire les caractères d'entrée et de produire comme résultat une suite d'unités lexicales que l'analyseur syntaxique va utiliser.
Cette interaction est couramment implantée en faisant de l'analyseur lexical un sous-programme ou une coroutine de l'analyseur syntaxique.
À la réception d'une commande « prochaine unité lexicale » émanant de l'analyseur syntaxique, l'analyseur lexical lit les caractères d'entrée jusqu'à ce qu'il puisse identifier la prochaine unité lexicale.



Modèle

En général la même unitée lexicale peut être produite en sortie par plusieurs chaînes d'entrée. L'ensemble de ces chaînes est décrit par une règle appelée modèle, associé à l'unité lexicale en question. On dit que le modèle filtre chaque chaîne de l'ensemble.

Lexème

Un lexème est une suite de caractères du programme source qui concorde avec le modèle d'une unité lexicale.
Par exemple, dans la déclaration Caml suivante :
`let pi = 3.1416 ; ;`
La sous-chaîne `pi` st un lexème de l'unité lexicale « *identificateur* ».

Attributs des unités lexicales

Quand plus d'un modèle reconnaît un lexème, l'analyseur lexical doit fournir aux phases suivantes du compilateur des informations additionnelles sur le lexème reconnu.

Par exemple le modèle *nb* (nombre) correspond à la fois aux chaînes 0 et 1, mais il est essentiel pour le générateur de code de savoir quelle chaîne a effectivement été reconnue.

L'analyseur lexical réunit les informations sur les unités lexicales dans les attributs qui leur sont associés.

- Les unités lexicales influent sur les décisions d'analyse syntaxique.
- Les attributs influent sur la traduction des unités lexicales.

En pratique, une unité lexicale a en général un seul attribut : un pointeur vers l'entrée de la table de symboles dans laquelle l'information sur l'unité lexicale est conservée ; le pointeur devient l'attribut de l'unité lexicale.

Exemple

$$
\begin{array}{lll}
\textit{istr} \rightarrow & \textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{istr} \\
\quad | & \textbf{si}\ \textit{expr}\ \textbf{alors}\ \textit{istr}\ \textbf{sinon}\ \textit{istr} \\
\quad | & \epsilon \\
\textit{expr} \rightarrow & \textit{term}\ \textbf{oprel}\ \textit{term} \\
\quad | & \textit{term} \\
\textit{term} \rightarrow & \textbf{id} \\
\quad | & \textbf{nb}
\end{array}
$$

Les terminaux **si**, **alors**, **sinon**, **oprel**, **id** et **nb** engendrent les ensembles de chaînes données par les définitions régulières suivantes :

- **si** $\rightarrow$ *si*
- **alors** $\rightarrow$ *alors*
- **sinon** $\rightarrow$ *sinon*
- **oprel** $\rightarrow$ $< \mid <= \mid = \mid <> \mid > \mid >=$
- **id** $\rightarrow$ $\textit{lettre}(\textit{lettre} \mid \textit{chiffre})^*$
- **nb** $\rightarrow$ $\textit{chiffre}^+(.\textit{chiffre}^+)?(E(+ \mid -)?\textit{chiffre}+)?$
- **lettre** $\rightarrow$ $[A - Za - z]$
- **chiffre** $\rightarrow$ $[0 - 9]$

Exemple (la suite)

- Pour ce fragment de langage, l'analyseur lexical reconnaîtra les mots clés **si**, **alors** et **sinon** au même titre que les lexèmes dénotés par **oprel**, **id** et **nb**.
- Pour simplifier l'exposé, on suppose que les mots clés sont réservés, c'est à dire qu'ils ne peuvent pas être utilisés comme identificateurs.
- On suppose que les lexèmes sont séparés par un espace constitué d'une suite non vide de blancs, tabulations et fins de ligne. Notre analyseur lexical délimitera ces espaces en comparant une chaîne avec la définition régulière **nb** suivante :
  - **delim** $\rightarrow$ *blanc* | *tabulation* | *fin de ligne*
  - **bl** $\rightarrow$ **delim**$^+$
- Si l'analyseur lexical trouve une correspondance avec **bl**, il ne retourne pas d'unité lexicale à l'analyseur syntaxique. Il continue par rechercher l'unité lexicale qui suit la suite de blancs et la retourne à l'analyseur syntaxique.

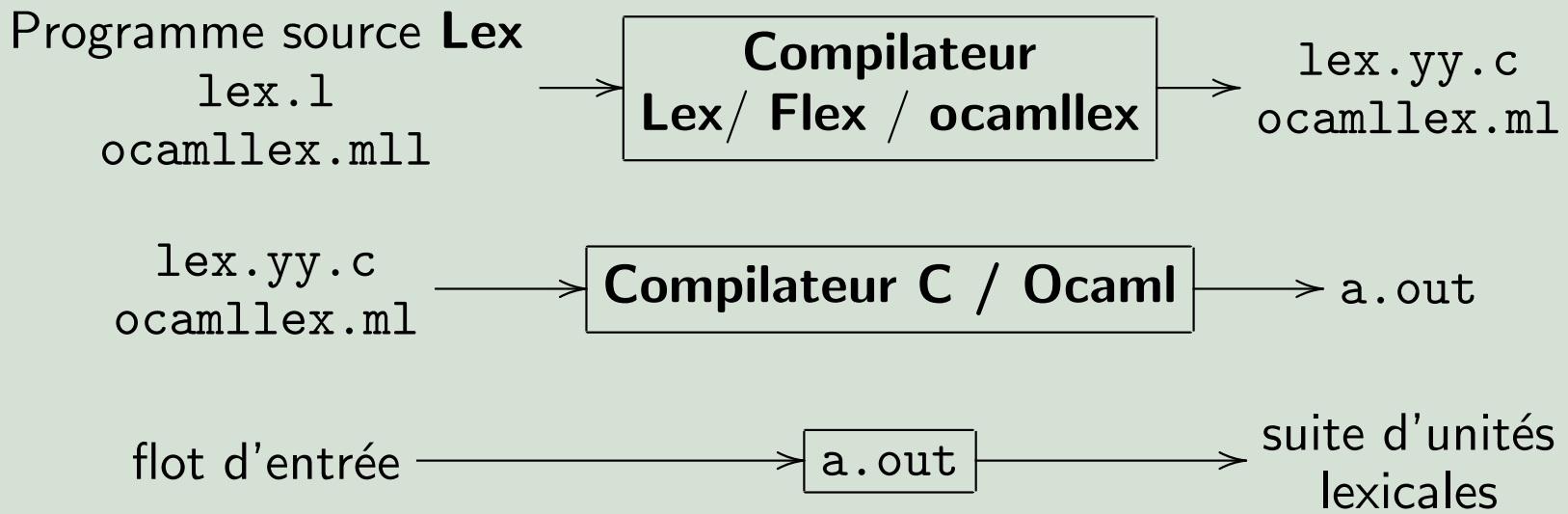
Exemple (la revanche)

Expression régulière	Unité lexicale	Valeur d'attribut
bl	—	—
si	**si**	—
alors	**alors**	—
sinon	**sinon**	—
id	**id**	pointeur vers une entrée de la table
nb	**nb**	pointeur vers une entrée de la table
$<$	**oprel**	*PPQ*
$<=$	**oprel**	*PPE*
$=$	**oprel**	*EGA*
$<>$	**oprel**	*DIF*
$>$	**oprel**	*PGQ*
$>=$	**oprel**	*PGE*

Les valeurs d'attribut pour les opérateurs de relation sont données par les constantes symboliques *PPQ*, *PPE*, *EGA*, *DIF*, *PGQ* et *PGE*.

Schéma

Création d'un analyseur lexical avec **Lex**.



- **Lex** est généralement utilisé de la manière décrite précédemment :
  - On prépare une spécification d'analyseur lexical en créant le programme `lex.l`, `ocamllex.mll`, etc. dans le langage Lex.
  - On compile `lex.l`, `ocamllex.mll`, etc. avec un compilateur comme **Lex**, **Flex** ou **ocamllex** pour produire un programme `lex.yy.c` ou `ocamllex.ml`.
- Le programme `lex.yy.c`, `ocamllex.ml`, etc. est une représentation sous forme de tableau d'un diagramme de transition construit à partir des expressions régulières de `lex.l`, `ocamllex.mll`, etc., accompagné d'une fonction standard qui utilise la table pour reconnaître les lexèmes.
- Les actions associées aux expressions régulières dans `lex.l`, `ocamllex.mll` sont des morceaux de code C, Ocaml, etc. et sont recopiées directement dans `lex.yy.c`, `ocamllex.ml`, etc.
- Enfin, `lex.yy.c`, `ocamllex.ml`, etc. est soumis au compilateur C, Ocaml, etc. pour produire un programme objet `a.out`, qui est l'analyseur lexical qui transforme un flot d'entrée en une suite d'unités lexicales.

Spécifications en Lex

Un programme **Lex** consiste en trois parties :

```
déclarations
%%
règles de traduction
%%
procédures auxiliaires
```

Déclarations

La section des déclarations comprend des déclarations de variables, des constantes littérales et des définitions régulières.

Constante littérale

Une constante littérale est un identificateur qui est déclaré pour représenter une constante.

Règles de traduction

Les règles de traduction d'un programme **Lex** sont des introduction de la forme :

$$\begin{array}{l} m_1 \ \{action_1\} \\ m_2 \ \{action_2\} \\ \quad \ldots \\ m_n \ \{action_n\} \end{array}$$

où chaque m_i est une expression régulière et chaque $action_i$ est un fragment de programme qui décrit quelle action l'analyseur lexical devrait réaliser quand un lexème concorde avec le modèle m_i.

Procédures auxilliaires

La troisième section contient toutes les Procédures auxilliaires qui pourraient être utiles dans les actions. On pourrait aussi compiler séparément ces procédures et les relier avec l'analyseur lexical.

- Quand l'analyseur lexical est activé par l'analyseur syntaxique, il commence à lire le texte d'entrée qui lui reste, caractère par caractère, jusqu'à ce qu'il ait trouvé le plus long préfixe du texte d'entrée qui corresponde à l'une des expressions régulières m_i. Il exécute alors l'action $action_i$.
- Typiquement, l'action rend le contrôle à l'analyseur syntaxique. Cependant, s'il ne le fait pas, l'analyseur lexical continue à chercher d'autres lexèmes jusqu'à ce qu'une action rende enfin le contrôle à l'analyseur syntaxique.
- La recherche répétée de lexèmes jusqu'à un retour explicite permet à l'analyseur lexical de traiter les espaces et les commentaires de manière simple.
- L'analyseur lexical retourne une seule information, l'unité lexicale, à l'analyseur syntaxique. Pour passer une valeur d'attribut donnant des informations sur le lexème, on peut l'affecter à une variable globale appelée *yylval*.

```
%{
  /* définitions des constantes littérales
  PPQ, PPE, EGA, DIF, PGQ, PGE,
  SI, ALORS, SINON, ID, NB, OPREL
%}

/* définitions régulières */
delim       [ \t\n]
bl          {delim}+
lettre      [A-Za-z]
chiffre     [0-9]
id          {lettre}+({lettre}|{chiffre})*
nombre      {chiffre}+(\.{chiffre}+)?(E[+\-]?{chiffre}+)?

%%
{bl}        {/* pas d'action et pas de retour */}
si          {return(SI);}
alors       {return(ALORS);}
sinon       {return(SINON);}
{id}        {yylval = RangerId(); return(ID);}
{nombre}    {yylval = RangerNb(); return(NB);}
"<"         {yylval = PPQ; return(OPREL);}
"<=         {yylval = PPE; return(OPREL);}
"="         {yylval = EGA; return(OPREL);}
"<>         {yylval = DIF; return(OPREL);}
">"         {yylval = PGQ; return(OPREL);}
">="        {yylval = PGE; return(OPREL);}

%%

RangerId()
{
  /* Procédure pour ranger dans la table des symboles le lexème dont le premier caractère
     est pointé par yytext et dont la longueur est yyleng et retourner un pointeur sur son entrée */
}

RangerNb()
{
  \begin{block}{}
  /* procédure similaire pour ranger un lexème qui est un nombre */
}
```

Remarques

- La section des déclarations contient éventuellement la déclaration de certaines constantes littérales utilises par les règles de traduction.
- Ces déclarations sont entourées par des parenthèses : `%{` et `%}`.
- Tout ce qui apparaît entre ces parenthèses est copié directement dans l'analyseur lexical `lex.yy.c` et n'est pas considéré comme faisant partie des définitions régulières des règles de traduction.
- Le même traitement s'applique aux procédures auxiliaires de la troisième section, `RangerId` et `RangerNb`, qui sont utilisées par les règles de traduction ; ces procédures seront copiées textuellement dans `lex.yy.c`.
- La section des déclaration comprend aussi quelques définitions régulières. Chacune de ces définitions consiste en un nom et une expression régulière dénotée par ce nom.
- Par exemple le premier nom défini est `delim`, il représente la classe de caractère `[ \t\n]`, donc un des trois caractères blanc, tabulation ou fin de ligne.

Remarques (oui, encore)

- Notez que le mot `delim` doit etre entouré d'accolades en **Lex**, pour le distinguer du modèle composé des cinq lettres *delim*.
- On note aussi l'antislash utilisé comme échappement pour laisser un caractère métasymbole de **Lex** retrouver son sens naturel. En particulier, le point décimal dans la définition d'un nombre s'exprime par `\.` car un point par lui même représente la classe de tous les caractères excepté la fin de ligne.
- Il existe une autre manière de s'assurer que les caractères aient leur sens naturel même si ce sont des metasymboles de **Lex** : les entourer de guillemets anglais : `"`.

Remarques (c'est pas fini)

Considérons les règles de traduction dans la section qui suit le premier %%

- La première règle dit que si on reconnaît `bl`, c'est à dire une suite maximale de blancs, tabulations et fins de ligne, on n'exécute aucune action. En particulier on ne retourne pas à l'analyseur syntaxique.
- La seconde règle spécifie que si on voit les lettres `si`, on retourne l'unité lexicale `SI` qui est une constante littérale représentant un certain entier interprété par l'analyseur syntaxique comme étant l'unité lexicale `si`.
- Dans la règle correspondant à `id`, on voit deux instructions dans l'action associée. D'abord, la variable `yylval` se voit affecter la valeur de la fonction `RangerId` ; la définition de cette fonction se trouve dans la troisième section.
- `yylval` est une variable dont la définition apparaît dans le résultat de **Lex** `lex.yy.c` et qui est aussi accessible par l'analyseur syntaxique. Le but de `yylval` est de contenir la valeur lexicale retournée, alors que la seconde instruction de l'action, `return(ID)`, se borne à retourner un code identifiant la classe lexicale.

Remarques (c'est bientôt bon)

Supposons qu'on donne à l'analyseur lexical résultant du programme précédent une entrée constituée de deux tabulations, des lettres `si` et d'un blanc.

- Les deux tabulations sont le préfixe initial le plus long reconnu par un modèle, ici le modèle `bl`. L'action associée à `bl` est de ne rien faire, donc l'analyseur lexical déplace le pointeur de début d'unité lexicale, `yytext`, jusqu'au `s` et commence à rechercher une autre unité lexicale.
- Le lexème suivant à reconnaître est `si`. Notons que les modèles `si` et `{id}` concordent tous les deux avec le lexème et qu'aucun modèle ne reconnaît une chaîne plus longue. Étant donné que le modèle pour le mot clé `si` précède celui des identificateurs dans la liste du programme, le conflit est résolu en faveur du mot clé.
- En général, cette stratégie de résolution de conflit facilite la réservation des mots clés ; il suffit de le placer avant le modèle définissant les identificateurs.

Remarques (courage)

- Prenons un autre exemple ; supposons que <= soient les deux premiers caractères lus. Bien que le modèle concorde avec le premier caractère, ce n'est pas la plus longue concordance d'un préfixe du texte d'entrée.
- De cette manière, la stratégie de **Lex** consistant à choisir le préfixe le plus long reconnu par un modèle facilite la résolution du conflit entre < et <= de la manière attendue : en choisissant <= comme prochaine unité lexicale.

Les constructions d'expressions régulières permisespar **Lex** sont données sur la page suivante, dans un ordre décroissant de priorité. Dans cette table `c` représente un caractère, `r` une expression régulière et `s` une chaîne et `\".^$[]*+?{}|/` des symboles d'opérateurs.

Expression	Désigne	Exemple
c	tout caractère c qui n'est pas un opérateur	a
\c	caractère littéral c	*
"s"	chaîne littérale s	"**"
.	tout caractère sauf fin de ligne	a.*b
^	début de ligne	^abc
$	fin de ligne	abc$
[s]	tout caractère appartenant à s	[abc]
[^s]	tout caractère n'appartenant pas à s	[^abc]
r*	zéro ou plusieurs r	a*
r+	un ou plusieurs r	a+
r?	zéro ou un r	a?
r{m,n}	entre m et n occurences de r	a{2,5}
r_1 r_2	r_1 puis r_2	ab
r_1 | r_2	r_1 ou r_2	a | b
(r)	r	(a | b)
r_1 / r_2	r_1 quand suivi de r_2	abc/123