

Compilateurs : Analyse lexicale

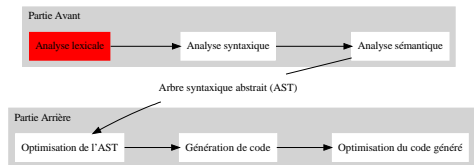
Matthieu Amiguet

2008 – 2009



- 1 Analyse lexicale
- 2 Implémenter un analyseur lexical
- 3 Traitements intermédiaires

“Vous êtes ici”



Analyse lexicale

- En entrée d'un compilateur, on a généralement un fichier
 - C'est à dire essentiellement une suite de caractères
- La première chose à faire est de regrouper ces caractères pour former des "mots"
 - nombres
 - identificateurs
 - mots réservés
 - ...
- Dans le vocabulaire des compilateurs, ces "mots" s'appellent des *lexèmes* (angl : *token*)
- L'étape d'identification des lexèmes s'appelle *analyse lexicale* (angl : *lexical analysis* ou *scanning*).

Les lexèmes

5

- Qu'est-ce qu'un lexème ? La réponse précise dépend beaucoup du contexte. . .
- Une bonne ligne de conduite :
 - Si un "truc" peut être séparé de ses voisins par des espaces sans changer la signification, c'est un lexème ; sinon pas.
- Ne signifie pas qu'un lexème ne doit pas contenir d'espaces !
 -
- Les commentaires et les espaces ne sont pas des lexèmes : ils sont simplement mis de côté par l'analyseur lexical. . .
 - . . . sauf s'ils sont conservés pour le diagnostic d'erreur, la réflexion, etc.

. . . et pas si simple !

7

- Dans certains cas, l'analyse lexicale est très facile :
 - `102*4-12*3`
- Mais parfois cela peut être bien plus délicat
 - `1e+3+3*pi-2*e-5`
 - `print "Il a dit : \"bonjour!\""`
 - `12.3+math.cos(180)`

Plus important qu'il n'y paraît. . .

6

- Malgré leur simplicité, les modules de lecture du texte source et d'analyse lexicale prennent beaucoup de temps
 - Jusqu'à 30% du temps de la partie avant !
- Ces deux modules sont en effet les seuls à voir tout le texte du programme
 - Une ligne moyenne fait 30-50 caractères. . .
 - et ne contient que quelques lexèmes (~5) !
- Il faut donc faire très attention à l'efficacité de leur implémentation !
- Pour un compilateur portable, c'est aussi à ce niveau-là qu'on se "débarrasse" des particularités des plate-formes (fins de lignes, . . .).

Le but du jeu

8

- Pour clarifier le but de l'analyse lexicale :
 - Il s'agit non seulement de séparer les lexèmes, mais aussi d'identifier leur type
- Exemple :
 - `12+3` \rightsquigarrow NUMBER(12) OPERATOR(+) NUMBER(1)
- La question maintenant est de faire ça de manière générique et efficace. . .

1 Analyse lexicale

2 Implémenter un analyseur lexical

- Écriture manuelle
- Générateurs
- Expressions régulières
- Retour aux générateurs

3 Traitements intermédiaires

Analyse lexicale
Implémenter un analyseur lexical
Générateurs
Génération automatique d'un analyseur lexical 11

- Il existe des générateurs de code
 - qui prennent en entrée une description des lexèmes ainsi que quelques informations annexes, et
 - qui produisent en sortie le code d'un analyseur lexical
- Avantages sur la génération à la main
 - Souplesse en cas de changement de description de lexèmes
 - Moins d'erreurs
 - Plus lisible
 - Algorithmes assez fortement optimisés
 - Si la description des lexèmes est assez "standard", elle peut également servir à documenter le langage...
- Le plus connu : Lex et ses variantes (Flex, JLex, ...)
- Le langage de description le plus courant : les *expressions régulières*

Analyse lexicale
Implémenter un analyseur lexical
Écriture manuelle
Écriture manuelle d'un analyseur lexical 10

- Il n'est pas très difficile d'écrire un analyseur lexical à la main
 - Un aiguillage sur le premier caractère lu permet souvent de séparer le problèmes en quelques sous-problèmes simples
 - Il ne reste plus qu'à écrire le code pour les sous-problèmes...
- Cette façon de faire est laborieuse, et peu adaptable en cas de changements/adaptations du langage...
- On préfère souvent générer automatiquement un analyseur sur la base de descriptions formelles des lexèmes.

Analyse lexicale
Implémenter un analyseur lexical
Expressions régulières
Expressions régulières 12

- Les expressions régulières sont un langage qui sert à définir des langages...
- L'idée est la suivante : grâce à quelques caractères à la signification spéciale, une chaîne de caractère servira à décrire tout un ensemble de chaînes de caractères

- L'intérêt des expressions régulières tient essentiellement dans deux faits
 - 1 Elles sont concises
 - 2 Elles peuvent être transformées facilement en code exécutable
- La traduction d'une expression régulière donne conceptuellement le même genre de code que l'écriture à la main
 - ↪ La traduction passe par la notion d'automate (cf. chapitre "Langages et Automates")

- Les expressions régulières sont plus ou moins standardisées...
- ... Mais souvent plutôt moins que plus !
- Les grands principes restent les mêmes, mais d'un contexte à l'autre de petits détails changent...
- Ce que nous décrirons dans ce cours est valable en python (au moins)
 - Tous les détails sur <http://docs.python.org/lib/re-syntax.html> et <http://www.amk.ca/python/howto/regex/>

- Les caractères spéciaux (ou *metacaractères*) sont `. ^ $ * + ? { } [] \ | ()`
- Tous les autres caractères se représentent eux-même
 - Ainsi, l'expression régulière `'toto'` représente exactement la chaîne de caractères `'toto'`
- Le métacaractère `.` représente tout caractère (sauf le retour à la ligne)
 - Ainsi, `HE.Arc` pourra représenter `HE-Arc`, mais aussi `HELArc`, `HE!Arc`, `HELlArc`, ... et même `HE.Arc!`

- `^` `$` début de ligne, fin de ligne
- `[...]` ensemble de caractères. `[ABC]` désigne `A` ou `B` ou `C`, `[A-Z]` n'importe quelle majuscule, `[A-Z0-9]` ...
- `[^...]` tout *sauf* l'ensemble de caractères. `[^A-Z]` désigne n'importe quoi sauf une majuscule.
- `r1|r2` désigne toute expression désignée par `r1` ou par `r2`. Par exemple, `bleu|blanc` désigne les chaînes `bleu` et `blanc`.
- `(? ...)` a le rôle des parenthèses dans une expression arithmétique. `t (? :oa|e) st` désigne `toast` et `test`.

- `\b` désigne le début ou la fin d'un mot. `\bbon` désignera le bon de bonjour mais pas de jambon
- `\d` équivalent à `[0-9]`
- `\D` équivalent à `[^0-9]`
- `\s` désigne un espace blanc (`[\t\n\r\f\v]`)
- `\S` tout mais pas `\s`
- `\w` `[a-zA-Z0-9_]`
- `\W` ...

- `r*` 0, 1 ou plusieurs fois `r`. Exemple : `ab*` désigne `a`, `ab`, `abb`, ...
- `r?` 0 ou 1 fois `r`
- `r+` 1 ou plusieurs fois `r`
- `r{m}` `m` fois `r`
- `r{m,n}` de `m` à `n` fois `r`

- Par défaut, une expression régulière correspond à la plus longue chaîne possible (comportement "gourmand" ou "glouton", angl. "greedy")

```
>>> re.compile(r'<.*>').findall('<b>gras</b>')
['<b>gras</b>']
```

- Une solution possible : l'exclusion

```
>>> re.compile(r'<[^>]*>').findall('<b>gras</b>')
['<b>', '</b>']
```

- Autre solution : les qualificateurs *non-gourmands* (non-standard !)

```
>>> re.compile(r'<.*?>').findall('<b>gras</b>')
['<b>', '</b>']
```

- Marche avec `*?`, `+?` et `??`

- On aimerait reconnaître la chaîne `(|)`... comment faire ?
- Les métacaractères peuvent
 - soit être échappés avec `\`
 - soit être inclus dans une classe de caractères `[]`
- Donc par exemple `\(|){2}` désigne `(|)`

L'enfer des échappements

21

- Supposons que l'on veut écrire une expression régulière qui corresponde à des expressions comme `\\labinfo\dfs`
- On peut essayer `"\\.\\.+.+"`
- Mais :
 - Il faut doubler les backslash dans les chaînes de caractères : `"\\\\\\.+.+"` nous donnera bien la chaîne ci-dessus
 - La chaîne ci-dessus vue comme expression régulière contient encore des échappements ! Il faut donc encore doubler : `"\\\\\\\\\\.+.+"`
- Le premier problème peut être résolu avec les *raw strings* en python : `r"\\\\\\.+.+"`

Applications

23

- Outils en ligne de commande (`grep`, `find`, ...) et langages de scripts (`sed`, `awk`, `perl`, ...)

```
local OSS="$(modprobe -l | grep "snd.*oss" | sed -e "s:\/.*\/::" -e "s:\\.\\.::")"
```
- Éditeurs (`emacs`, `vim`, `SciTe`, ...)
 - Chercher/remplacer
 - Coloration syntaxique
- Traitements de texte (`Openoffice`, `Word` – implémentation partielle)
- Conception de compilateurs (`Lex`, ...)
- Bibliothèques dans divers langages de programmation (`Java`, `C#`, `Python`, ...)
 - Vérification des entrées de l'utilisateur, ...

Exemples

22

- Ligne ne contenant que des caractères alphanumériques
`^[w*]$`
- Nombres entiers
`[+]?[d+]`
- Commentaires (suivant un #)
`#.*$`
- Chaînes de caractères
`"[^"]*" , ou
".*?"`

Décrire les lexèmes

24

- Avec un générateur, on peut se contenter de fournir une liste de lexèmes avec chacun l'expression régulière correspondante. . .
- . . . et laisser le générateur nous faire un code qui sépare et classe les lexèmes
- Dans les cas compliqué, on recourt parfois aux *descriptions régulières*

Description régulière

- Ensemble d'expressions régulières nommées
 - On peut réutiliser une expression définie précédemment
 - Par contre, une définition récursive est interdite !
-
- Par substitutions successives, on peut donc se ramener à une expression régulière.

- L'analyseur lexical doit déterminer quelle suite de caractères correspond à quelle expression régulière
- Que faire si plusieurs solutions sont possibles ?
 - Généralement, on considère l'ordre dans lequel les descriptions de lexèmes ont été données.
- Par exemple, on aura intérêt à donner la description d'un float *avant* celle d'un entier...

- "Un identificateur est une suite de lettres, de chiffres et de soulignements qui commence par une lettre ; deux soulignements consécutifs sont interdits, ainsi qu'un soulignement final"

- Description régulière (exemple !) :

```
lettre → [a-zA-Z]
chiffre → [0-9]
souligne → _
lettre_ou_chiffre → lettre | chiffre
fin_soulignee → souligne lettre_ou_chiffre+
identificateur → lettre lettre_ou_chiffre*
fin_soulignee*
```

- Expression régulière équivalente :

```
[a-zA-Z][a-zA-Z0-9]*(_[a-zA-Z0-9]+)*
```

- 1 Analyse lexicale
- 2 Implémenter un analyseur lexical
- 3 **Traitements intermédiaires**

Identification des lexèmes

29

- Dans son acception la plus "propre", le rôle de l'analyseur lexical est de fournir une suite de lexèmes
- Cependant, dans un certains nombre de cas, il est nécessaire de faire déjà un petit peu de sémantique
- Exemple : identificateur influançant sur la suite de l'analyse

```
Typedef int T;  
(T *) var; ~> correct!  
Mais  
int T;  
(T *) var; ~> incorrect!
```
- Dans certains cas, des traitements sont effectués à ce stade pour augmenter l'efficacité
 - Exemple : la chaîne de caractères "127" peut déjà être convertie en l'entier 127...

Autres traitements intermédiaires

31

- Inclusion de fichiers
- Compilation conditionnelle
- Généricité
- ...

Table des symboles

30

- Rassemble toutes les informations nécessaires sur les identificateurs
- Les informations récoltées dépendent du compilateur
- Conceptuellement, correspond à un tableau indexé par des chaînes de caractères
 - Dans les langages n'implémentant pas ce concept, il faut se débrouiller...
