Automates et Analyse Lexicale

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

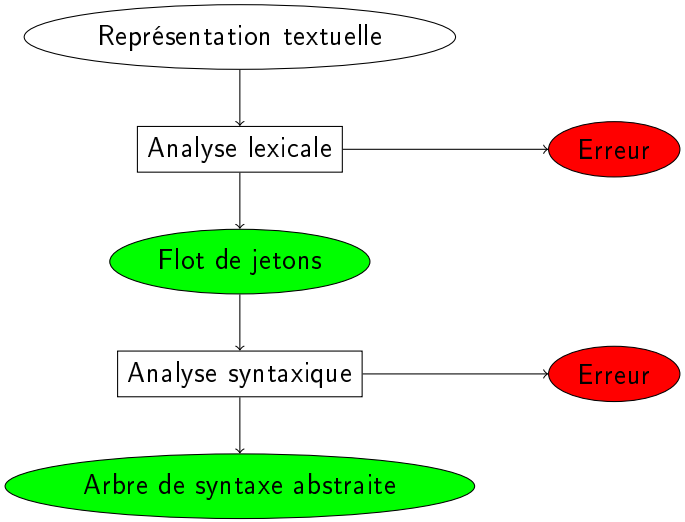
treinen@irif.fr

26 octobre 2021

L'objectif de l'analyse lexicale

- Découper un texte d'entrée en une séquence de *lexèmes*, et les représenter par des *jetons* (*tokens* en anglais)
- À la base : Classification des lexèmes qui peuvent paraître dans un texte d'entrée, à l'aide des expressions régulières.
- La phase suivante de l'analyse (l'analyse syntaxique, voir plus tard) va travailler sur le résultat de ce découpage : il s'agit d'une *abstraction* du texte d'entrée.

Les deux phases de l'analyse

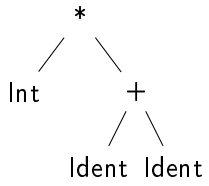


Exemple

- Texte d'entrée :

(	7	5	6	e	2		*		(	e	5	e	7		+		v	a	l	e	u	r	2	)	)

- Jetons :
  PARG INT MULT PARG IDENT PLUS IDENT PARD PARD
- Résultat de l'analyse syntaxique (voir plus tard) :



Définition des catégories lexicales

Sur l'exemple des expressions arithmétiques (on utilise \ comme symbole d'échappement) :

- INT : [0..9]+(e[0..9]+)?
- IDENT : [a..zA..Z][a..zA..Z0..9]*
- PARG : (
- PARD :)
- MULT : *
- PLUS : +

Jetons avec arguments

- En réalité, on veut aussi garder certaines informations avec les jetons, comme la valeur d'une constante entière, ou le nom d'un identificateur.
- Certains jetons doivent donc avoir un argument :
  - IDENT(string)
  - INT(int) (c'est bien `int` et pas `string` !)
- Séquence des jetons obtenue sur l'exemple :
  PARG INT(75600) MULT PARG IDENT("e5e7") PLUS IDENT("valeur2") PARD PARD

Ignorer des informations pas pertinentes

L'analyse lexicale sert aussi à faire abstraction de certaines informations dans le texte d'entrée qui ne sont pas pertinentes pour l'analyse du texte. Souvent il s'agit de :

- Les espaces : sont utiles pour indiquer la fin d'un mot. Les espaces sont utiles *pour* l'analyse lexicale, mais une fois le découpage fait on peut les oublier.
- Les commentaires : souvent l'analyse lexicale vérifie l'écriture correcte des commentaires, mais ne les représente pas dans sa sortie.

Exemple

Différents textes d'entrée qui peuvent donner la même séquence de jetons :

- `34 * (x + y)`
- `34*(x+y)`
- `34 *( x+ y)`
- `34 * (x+y) /* Ceci est un commentaire */`

Quelle information retenir dans les jetons

- On retient dans les jetons seulement l'information qui est utile pour la suite.
- La distinction entre information utile/inutile dépend de l'application.
- Par exemple : Les commentaires peuvent être utiles à retenir pour certaines applications.
- Il peut être utile de conserver avec les jetons aussi des informations de *localisation* : nom du fichier source, numéro de ligne, numéro de colonne.

Résoudre les ambiguïtés

- L'analyse lexicale va, pour produire le jetons suivant, chercher un *préfixe* du reste du texte qui correspond à une des catégories lexicales, et construire le jeton correspondant.
- Il y a deux sources d'ambiguïtés :
  - Des préfixes de longueurs différents peuvent être reconnues
  - Les expressions régulières peuvent avoir une intersection non vide

Préfixes de longueur différentes reconnues

Exemple :

- Catégorie lexicale :
  - IDENT : `[a..z]+`
- Début du texte d'entrée :
  `xyz`
- Plusieurs possibilités de découpage :
  1. `IDENT("x") IDENT("y") IDENT("z")`
  2. `IDENT("xy") IDENT("z")`
  3. `IDENT("x") IDENT("yz")`
  4. `IDENT("xyz")`
- La règle normale est : on cherche le préfixe *maximal*. Dans l'exemple ca donne `IDENT("xyz")`.

Plusieurs expressions régulières s'appliquent

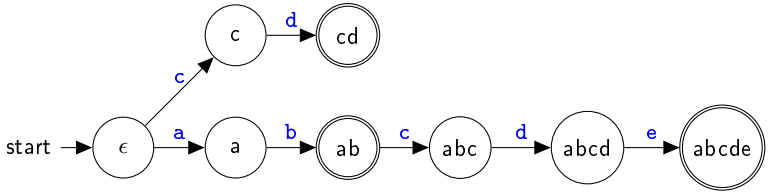
Exemple :

- Catégories lexicales :
  - PUBLIC : `public`
  - IDENT : `[a..z]+`
- Début du texte d'entrée :
  `public publication`
- Plusieurs possibilités de découpage :
  1. `PUBLIC IDENT("publication")`
  2. `IDENT("public") IDENT("publication")`
- La règle normale est : à longueur égale du mot reconnu, c'est la première expression régulière qui gagne.

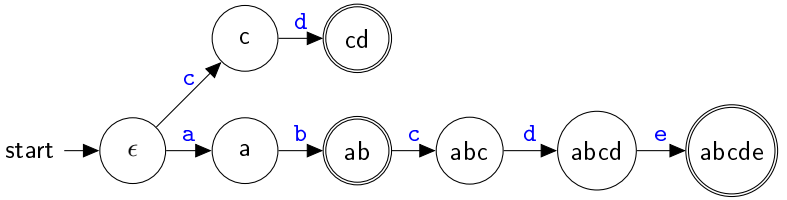
Exécution de l'automate pour le recherche d'un préfixe maximal

Exemple (artificiel) :

- Catégorie lexicales :
  - AB : `ab`
  - CD : `cd`
  - ABCDE : `abcde`
- Automate pour $\{\texttt{ab}, \texttt{cd}, \texttt{abcde}\}$:



Exemple



- Supposons qu'on cherche le préfixe maximal de l'entrée qui est reconnu par l'automate (cas normal).
- Entrée : `abcd`
- L'automate consomme tout le mot, et arrive dans l'état abcd.
- Il aurait dû trouver `ab` !

Exécution de l'automate pour le recherche d'un préfixe maximal

- Si on cherche le préfixe le plus *court* reconnu on doit s'arrêter dès qu'on arrive dans un état acceptant.
- Si on cherche le préfixe le plus *long* reconnu on doit continuer à lire tant que possible, et quand on passe par un état acceptant :
  - mémoriser l'état acceptant ;
  - mémoriser la position dans le mot d'entrée.

  Si l'automate ne peut plus continuer : remettre le pointeur de lecture dans l'entrée à la position où on a vu le dernier état acceptant.

Conclusion jusqu'ici : Objectif de l'analyse lexicale

- Lire le texte d'entrée, et faire un premier traitement en vue d'une simplification pour les étapes suivantes :
- *Découpage* de l'entrée en *lexèmes* (des mots élémentaires)
- *Classer* les lexèmes identifiés, création de *jetons*. Les jetons seront les symboles de l'alphabet de la grammaire (voir la semaine prochaine).
- *Interpréter* les lexèmes quand pertinent, par exemple transformer une suite de chiffres en un entier. Ces valeurs seront attachées au jetons, elles se retrouveront dans la syntaxe abstraite produite par la grammaire (voir dans quelque semaines).
- *Abstraire* l'entrée : ignorer des détails non pertinents pour la suite (espaces, commentaires, ...)

Implémenter une analyse lexicale

- Soit *écrire un programme* (Java ou autre) à la main, basé sur un automate fini : discuté au dernier cours.
  Les premiers compilateurs étaient effectivement écrits de cette façon (compilateur du langage FORTRAN, Backus et al. 1957 : 18 personnes-années).
- Soit faire *engendrer* un analyseur lexicale à partir d'une spécification (ce cours).
  C'est la technique utilisée pour l'écriture des compilateurs modernes.

Différents générateurs

- Existent pour presque tous les langages de programmation.
- Le premier générateur était *lex*, publié en 1975 par Mike Lesk et Eric Schmidt. Engendre du code en C.
- Successeur : *flex*, 1987.
- Les générateurs modernes sont souvent issus de flex. Nous utilisons ici un générateur pour Java : *jflex*.
- Les générateurs pour des autres langages de programmation sont très similaires.

Le code engendré dans le cas de jflex

- Une classe pour l'analyseur lexicale, le nom de la classe peut être défini dans la spécification (dans nos exemples : `Lexer`).
- La création d'un objet de cette classe (un analyseur) prend en argument un objet qui représente le *flot d'entrée*, par exemple un fichier, ou l'entrée standard.
- Il y a une méthode pour demander le jeton suivant. Le nom de cette méthode, et le type des jetons, peuvent être définis dans la spécification.

La spécification

Trois parties, séparées par des lignes `%%` :

1. Code utilisateur
2. Options et déclarations
3. Règles lexicales

Fichier `arith.flex` I

```
// partie code utilisateur : vide

%%
// partie options et déclarations
%public
%class Lexer
%unicode
%type Token

EspaceChar = [ \n\r\f\t]
Ch         = [0-9]
Le         = [a-zA-Z]

%%
// partie règles lexicales
{Ch}+             {return new IntToken(Sym.INT,
                                Integer.parseInt(yytext()));}
{Le}({Le}|{Ch})*  {return new StringToken(Sym.IDENT,yytext());}
```


Fichier `arith.flex` II

```
"("               {return new Token(Sym.PARG);}
")"               {return new Token(Sym.PARD);}
"*"               {return new Token(Sym.MULT);}
"+"               {return new Token(Sym.PLUS);}
{EspaceChar}+     {}
```


La partie *Code utilisateur*

- copiée simplement au début du fichier engendré (devant la définition de la classe)
- partie souvent vide (sauf commentaires, et `import ...`)

La partie *Options et Déclarations*

Options : commencent avec le symbole %. Parmi les options les plus importantes :

- `%class` *nom* : donne le nom de la classe engendrée.
- `%public` : la classe engendrée est publique
- `%type` *t* : le type de résultat de la fonction principale de l'analyse lexicale `yylex`.
- `%unicode` : accepte des caractères Unicode en entrée de l'analyse lexicale (recommandé)
- `%line` : compte lignes pendant l'analyse lexicale (disponible en `yyline`).
- `%column` : compte colonnes pendant l'analyse lexicale (disponible en `yycolumn`).
- `%state` *s* : Déclaration de l'état *s* (voir plus tard)

La partie *Options et Déclarations*

- Code entre `%{` et `%}` (peut être sur plusieurs lignes) :
  - copié au début de la classe engendré
  - ce code a donc accès aux champs de la classe (par exemple, `yyline`, `yycolumn`)
- Code entre `%eofval{` et `%eofval}` : code exécuté quand l'analyse lexicale arrive à la fin de l'entrée (défaut : `null`).

La partie *Options et Déclarations*

Macros : système de définitions d'expressions régulières

- mettre les mots entre apostrophes " et "
- pour utiliser une expression régulières préalablement définie, par exemple de nom *r* : `{r}`.
- classes de caractères, par exemple `[a-z]`
- quelque classes de caractère prédéfinies, par exemple `[:letter:]`, `[:digit:]`, `[:uppercase:]`, `[:lowercase:]`.

La partie *Règles Lexicales*

- Séquence de
  expression-régulière `{` *code-java* `}`
- Dans le cas le plus simple, le code java est un `return ...`
- Règles d'exécution : on cherche le lexeme le plus long possible, et on applique l'action de la première expression régulières qui s'applique.

Le premier exemple

- Analyse lexicale pour des expressions arithmétiques comme vu la dernière fois.
- Petite différence au premier exemple : les entiers ne contiennent pas d'exposant.
- Définition des classes pour les Symboles (type de jetons), puis pour les jetons éventuellement avec des arguments.
- Le fichier de spécification pour `jflex`.
- Un petit programme principal pour tester.

Fichier Sym.java

```
// symboles, sont utilisés dans les jetons

public enum Sym {
    INT, IDENT, PARG, PARD, MULT, PLUS;
}
```


Fichier Token.java I

```
// jetons, résultat de l'analyse lexicale

public class Token {
    // jeton simple : seulement un symbole
    protected Sym symbol;

    public Token(Sym s) {
        this.symbol = s;
    }

    public Sym symbol() {
        return this.symbol;
    }

    public String toString(){
        return this.symbol.toString();
    }
}
```


Fichier Token.java II

```
class StringToken extends Token {
    // jeton StringToken, avec valeur de type string
    private String value;

    public StringToken(Sym c, String s) {
        super(c);
        this.value = s;
    }

    public String toString(){
        return this.symbol + "(" + this.value + ")";
    }
}

class IntToken extends Token {
    // jeton IntToken, avec valeur de type int
    private int value;
```


Fichier Token.java III

```
    public IntToken(Sym c, int i) {
        super(c);
        this.value = i;
    }

    public String toString(){
        return this.symbol + "(" + this.value + ")";
    }
}
```

Fichier arith.flex I

```
// partie code utilisateur : vide

%%
// partie options et déclarations
%public
%class Lexer
%unicode
%type Token

EspaceChar = [ \n\r\f\t]
Ch         = [0-9]
Le         = [a-zA-Z]

%%
// partie règles lexicales
{Ch}+             {return new IntToken(Sym.INT,
                                Integer.parseInt(yytext()));}
{Le}({Le}|{Ch})*  {return new StringToken(Sym.IDENT,yytext());}
```


Fichier arith.flex II

```
"("               {return new Token(Sym.PARG);}
")"               {return new Token(Sym.PARD);}
"*"               {return new Token(Sym.MULT);}
"+"               {return new Token(Sym.PLUS);}
{EspaceChar}+     {}
```

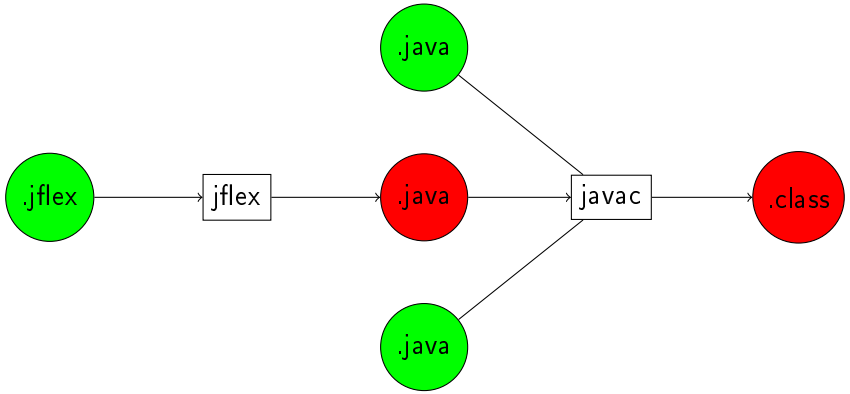

Fichier Test.java I

```
import java.io.*;

public class Test {
    public static void main(String[] args){
        try {
            // créer le reader utilisé par l'analyseur lexicale pour lir
            // le texte d'entrée
            BufferedReader reader =
                new BufferedReader(new FileReader(new File(args[0])));

            // créer l'analyseur lexical
            Lexer lexer = new Lexer(reader);

            // boucle: analyse lexicale du texte entier, et afficher
            // les jetons engendrés.
            Token t;
            do {
                t = lexer.yylex();
```


Fichier Test.java II

```
                if (t != null) {
                    System.out.println(t);
                }
            } while (t != null);
        } catch (IOException e){
            System.err.println("IOException: " + e.getMessage());
        }
    }
}
```

Utilisation de jflex



Ce que JFlex fait pour vous

- Création des classes de caractères : tous les caractères qui ne sont jamais distingués par les expressions régulières sont groupés dans la même classe.
- Les classes crées doivent être disjointes.
- Exemple : expressions régulières :
  `"end"`
  `[a-z]*`
- Quatre classes de caractères disjointes :
  `[e]`, `[n]`, `[d]`, `[a-cf-mo-z]`

Ce que JFlex fait pour vous

- Création d'un automate non-déterministe pour l'union de toutes les expressions régulières.
- Déterminiser l'automate (et éliminer les ϵ-transitions).
- Minimiser l'automate.
- On peut demander à `jflex` de montrer ces trois automates (option `-dot`, visualiser les automates avec `xdot` par exemple)
- Il y a également un mode autonome (*standalone*) pour des applications simple (pas de génération de jetons), voir le TP.

Les états de l'analyseur lexical

- Par défaut (comme sur le premier exemple), votre analyseur lexical a un seul état.
- Il peut être utile d'avoir plusieurs états — dans chaque état, Flex peut utiliser des expressions régulières différentes.
- Pour en avoir plusieurs :
  - les déclarer à l'aide de `%state` (sauf YYINITIAL)
  - mettre toutes les règles dans le contexte d'un état
  - dans les actions : changer d'état à l'aide de `yybegin`.
- Ne pas confondre les états de Flex avec les états de l'automate fini obtenu à partir des expressions régulières

Pourquoi utiliser plusieurs états ?

- Un premier exemple sont les commentaires : avec une expression régulières comme "/*" .* "*/" on a un problème quand il y a plusieurs commentaires dans le texte (pourquoi ?)
- Dans ce cas on veut on fait trouver le mot *le plus court* décrit par l'expression régulière. Cela peut être simulé en utilisant deux états.

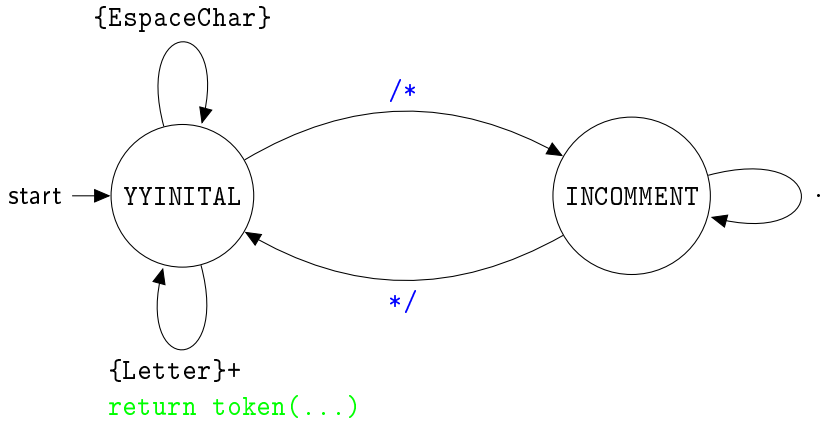
Reconnaître les commentaires (simplifié)

```
%%
%type Token
EspaceChar = [ \n\r\f\t]
Letter     = [a-zA-Z]
%state INCOMMENT
%%
<YYINITIAL> {
  {Letter}+     {return token(Sym.IDENT, yytext());}
  {EspaceChar}  {}
  "/*"          {yybegin(INCOMMENT);}
}
<INCOMMENT> {
  "*/"   {yybegin(YYINITIAL);}
  .      {}
}
```


Reconnaître les commentaires

- `YYINITIAL` est l'état par défaut
- Il est important que la dernière règle s'applique à un mot de longueur 1 seulement (c.-à-d. expression régulière ., et pas .+)

Les états de Flex dans l'exemple des commentaires



Exemple : découper un mot en plusieurs parties

- Retour à notre premier exemple : on souhaite maintenant aussi reconnaître des entiers avec exposant (756e2, par exemple).
- On utilise deux états : quand on trouve un symbole "e" après une séquence de chiffres on stocke la valeur entière trouvée (devant le "e") dans une variable, puis on va dans un deuxième état pour lire exposant.

Nouvelle version de `arith.flex` I

```
%%

%public
%class Lexer
%type Token
%unicode

%{
  int intbuff=0;
  private String chop(String s) {
      // envoie s sans son dernier caractère
      return(s.substring(0,s.length()-1));
  }
  private int expo(int base, int ex) {
      // envoie (base * 10**ex)
    int result=base;
    for(int i = 1; i<=ex; i++) {
      result=result*10;
```


Nouvelle version de `arith.flex` II

```
    }
    return result;
}
%}

EspaceChar = [ \n\r\f\t]
Ch         = [0-9]
Le         = [a-zA-Z]

%state EXPONENT

%%

<YYINITIAL> {
  {Ch}+             {return new IntToken(Sym.INT,
                                  Integer.parseInt(yytext()));}
  {Le}({Le}|{Ch})*  {return new StringToken(Sym.IDENT,yytext());}
  "("               {return new Token(Sym.PARG);}
```


Nouvelle version de `arith.flex` III

```
  ")"               {return new Token(Sym.PARD);}
  "*"               {return new Token(Sym.MULT);}
  "+"               {return new Token(Sym.PLUS);}
  {EspaceChar}+     {}
  {Ch}+ "e"         {intbuff=Integer.parseInt(chop(yytext()));
                     yybegin(EXPONENT);}
}
<EXPONENT> {
  {Ch}+ {yybegin(YYINITIAL);
         return new IntToken(Sym.INT,
                             expo(intbuff,Integer.parseInt(yytext())));}
}
```

Mots clefs d'un langage de programmation

Solution naïve : une règle par mot clefs.

```
%%
%type Token
EspaceChar = [ \n\r\f\t]
Letter     = [a-zA-Z]
%%
"begin"        {return token(Sym.BEGIN)}
"end"          {return token(Sym.END)}
"class"        {return token(Sym.CLASS)}
{Letter}+      {return token(Sym.IDENT, yytext());}
{EspaceChar}   {}
```


Attention à l'ordre des règles

Entrée : `beg begin beginner`

- Premier appel à `yylex()` : seulement la quatrième règle s'applique ⇒ token IDENT.
- Deuxième appel à `yylex()` : les règles (2) et (4) s'appliquent au même lexeme `begin`, c'est donc la première parmi ces deux qui gagne ⇒ token BEGIN.
- Troisième appel à `yylex()` : les règles (2) et (4) s'appliquent mais la dernière reconnaît un lexeme plus long ⇒ token IDENT.

Mais regardez la taille de l'automate engendré !

Contrôler la taille de l'automate

- Techniques utilisés par le générateurs :
  - Utiliser des classes de caractères dans la représentation de l'automate.
  - Minimiser l'automate engendré à partir des expressions régulières.
- Optimisation dans la spécification : Éviter de créer une nouvelle classe lexicale pour chaque mot clef
  (Java : 46 mots clefs.)

Comment reconnaître les mots clefs sans catégories dédiées ?

- En Java (et pareil dans les autres langages de programmation) : tous les mots clefs sont des séquences de lettres en minuscules.
- Mettre une seule catégorie pour les identificateurs.
- Dans l'action associé, on cherche (par ex. dans une table de hachage) si le lexeme est un mot clefs, et crée un jeton en fonction.

Tables de hachage : pour faire quoi ?

- Représenter des fonctions partielles finies $f : D_A \rightsquigarrow D_B$
- Cas d'usage : Le domaine *potentiel* D_A est très grand ou même infini ; par contre f est définie seulement pour un "petit" nombre de valeurs.
- (1) On souhaite un coût mémoire plus au moins linéaire dans la taille du domaine plus la taille du co-domaine de f.
- (2) On souhaite une complexité constante pour accéder à la valeur de la fonction appliquée à un argument.
- (3) Fonctions modifiables : possibilité d'ajouter ou de supprimer des paires (argument, résultat)

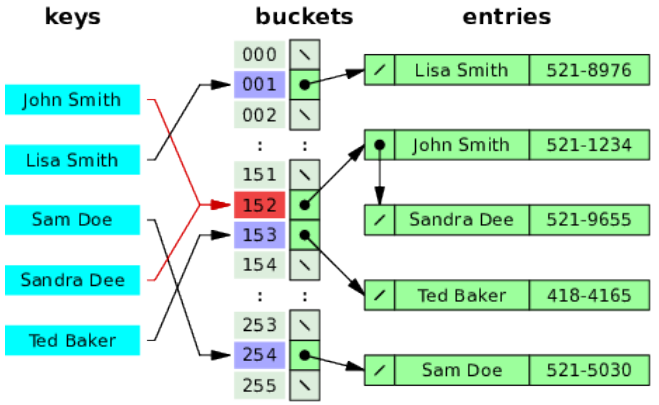
Tables de hachage : comment ça fonnctionne

- Les tableaux répondent aux objectifs 2 (complexité constante) et 3 (fonctions modifiables), à part du fait qu'il faudrait utiliser des valeurs d'un type D_A en tant d'indices.
- On utilise une fonction de hachage h: $D_A \rightarrow \mathit{int}$ pour mapper les arguments de la fonction f vers des entiers (indices du tableau).
- Cela nous permet aussi de répondre à l'objectif 1 : la fonction de hachage h est non-injective, et on s'arrange pour que l'image de h soit un intervalle $[0, \ldots, d-1]$.
- Le tableau peut donc avoir la taille d.

Tables de hachage : le problème des conflits

- On ne peut pas exclure des conflits de la fonction de hachage : $x \neq y$ et $h(x) = h(y)$, même si on essaye de les éviter par un bon choix de la fonction de hachage.
- Pour gérer les conflits, les entrées du tableau ne sont pas des valeurs de D_B, mais encore des fonctions partielles $D_A \rightsquigarrow D_B$. Ces fonctions devront avoir un domaine vraiment petit (quelques éléments seulement), on peut donc les représenter par une liste d'association par exemple.

Table de hachage avec listes chaînées



Le fichier keys.flex I

```
import java.util.HashMap;
class Keys extends HashMap<String,Sym> {
      public Keys() {
            super();
            this.put("end",Sym.END);
            this.put("begin",Sym.BEGIN);
            this.put("class",Sym.CLASS);
      }
}

%%
%public
%type Token
%class Lexer
%unicode
EspaceChar = [ \n\r\f\t]
Letter     = [a-zA-Z]
```


Le fichier keys.flex II

```
%{
 private Keys keys = new Keys();
 private Token ident_or_keyword(String lexeme) {
    Sym s = keys.get(lexeme);
    if (s == null) { /* not a keyword */
       return new StringToken(Sym.IDENT, lexeme);
    } else {  /* keyword */
       return new Token(s);
    }
 }
%}

%%
{Letter}+      {return ident_or_keyword(yytext());}
{EspaceChar}   {}
```