Grammaires locales étendues
mise en œuvre

Dans ce chapitre nous présentons la mise en œuvre du formalisme des grammaires locales étendues qui a été introduit au chapitre chapitre 4. Leur développement s'inscrit dans le cadre du moteur linguistique Unitex, qui comme a été évoqué au chapitre 3 est un outil open source pour le traitement de corpus textuels à l'aide de grammaires locales. Dorénavant, nous utilisons le terme manuel pour faire référence au « Manuel d'utilisation d'Unitex » (Paumier, 2016).

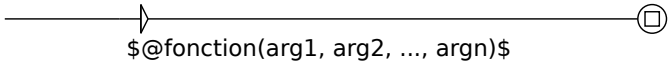
Rappelons qu'une des particularités qui différencie une grammaire locale étendue d'une grammaire locale classique est la capacité d'associer des fonctions arbitraires aux transitions. Considérer des fonctions arbitraires suppose d'une part qu'il n'existe pas de contrainte spécifique au moment où on associe une fonction à une transition, d'autre part, cela implique que les fonctions sont liées à la grammaire mais ne sont pas implémentées dans celle-ci. Par exemple, si *reverse* est le nom d'une fonction placée sur une des transitions d'une grammaire, il n'en existe aucune définition dans la grammaire relative à cette fonction, mise à part la syntaxe d'appel. Une telle caractéristique fournie, du point de vue des traitements, une grande souplesse, mais impose à la fois une réflexion sur la façon de mettre en œuvre les fonctions d'une grammaire locale étendue.

5.1 Anatomie de l'appel à une fonction étendue

Dans notre implémentation une fonction étendue est appelée en suivant différents types de conventions, dont la plus simple est :

$$\texttt{\$@fonction(}arg_1\texttt{, }arg_2\texttt{, ..., }arg_n\texttt{)\$}$$

Sous forme de graphe :



Graphe 5.1 – Convention pour appeler une fonction étendue

Le symbole dollar (`$`) comme préfixe et suffixe sert à délimiter la partie de la *sortie étendue* qui doit être évaluée, autrement dit, les séquences en dehors de l'appel à la fonction étendue sont interprétées séparément soit comme des symboles terminaux (des chaînes de caractères littérales), soit comme des appels à d'autres fonctions. S'il s'agit des symboles de l'alphabet de sortie et si la *sortie est satisfaite*, alors ces symboles seront concaténés à la valeur retournée par la fonction. Dans le cas où une sortie étendue comporte plus d'une fonction, l'évaluation est effectuée de la fonction la plus à gauche vers celle la plus à droite. Dans tous les cas, l'évaluation est effectuée seulement si la fonction qui la précède est aussi satisfaite.

Le nom d'une fonction est précédé par le symbole arobase (@). Nous avons choisi cette convention puisque l'arobase est communément associé à la préposition « à » (« at », en anglais). En effet, cela sert à évoquer le fait que la définition de la fonction, c'est-à-dire, le code source contenant les instructions à exécuter se trouve dans un fichier externe (que nous appelons extension) du même nom et se terminant par `.upp` (dans l'exemple, `fonction.upp`).

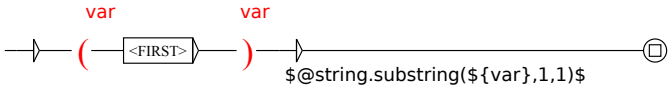
Le nom d'une fonction est valide s'il est composé uniquement de caractères appartenant à la classe `[a-zA-Z0-9_]`, soit des lettres latines minuscules ou majuscule, des chiffres et des tirets bas. Cette restriction suit celle déjà utilisée dans Unitex pour créer des noms de variables (d'entrée ou de sortie, manuel, p. 105) et permet ainsi de garantir une cohérence vis-à-vis de la création des identifiants associés aux grammaires.

Dans certains cas il est plus propice de définir plus d'une fonction étendue dans une seule extension `.upp`, par exemple, afin de créer des bibliothèques de fonctions qui partagent un but ou une sémantique commune. Il est alors admis de faire appel à une fonction en utilisant la convention suivante :

$$\texttt{\$@extension.fonction(}arg_1\texttt{, }arg_2\texttt{, ..., }arg_n\texttt{)\$}$$

Dans le graphe 5.2, *string* est le nom du fichier `.upp` contenant la fonction étendue du nom *substring*. Le nom d'une extension doit respecter les mêmes caractéristiques que celles établies pour les noms de variables et de fonctions étendues, en d'autres termes,

être composé uniquement de lettres latines minuscules ou majuscules, de chiffres et de tirets bas. En outre, aussi bien les noms de variables, de fonctions étendues et d'extensions doivent avoir une longueur inférieure ou égale à 127 caractères et sont sensibles à la casse, c'est-à-dire, les noms avec des lettres minuscules sont considérés comme étant différents de ceux contenant des lettres majuscules.



Graphe 5.2 – Convention pour appeler une fonction étendue avec un nom différent du fichier où elle est définie

Immédiatement après le nom de la fonction étendue, il est requis d'écrire une parenthèse ouvrante qui délimite avec une autre parenthèse fermante les arguments qui sont passés à la fonction étendue. Le nombre des arguments situées entre parenthèses doit normalement correspondre au nombre des paramètres attendus par la définition de la fonction. Cependant, il n'existe pas de restriction de cardinalité, si un appel est réalisé avec un nombre d'arguments différents que celui de paramètres, alors les arguments en excès sont ignorés et les paramètres manquants sont assignés à la valeur nulle.

Comme évoqué précédemment, une fonction étendue peut recevoir un ensemble d'arguments, séparés par des virgules, en tant que paramètres d'entrée. Tandis que les paramètres sont les variables utilisées pour définir l'implémentation de la fonction étendue (dans un fichier `.upp` externe à la grammaire), les arguments sont les valeurs qui sont passées à la fonction lorsqu'elle est appelée dans un graphe.

Une liste d'arguments peut être vide. En d'autres termes, la fonction étendue peut avoir zéro paramètres d'entrée ou se voir affectée la valeur nulle à chacun de ses paramètres manquants. Par contre, si la fonction étendue reçoit un ou plusieurs paramètres, chaque argument transmis peut correspondre à une valeur nulle, à une valeur booléenne (vraie ou fausse), à une chaîne de caractères, au contenu d'une variable d'entrée (manuel, p. 105) ou de sortie (manuel, p. 147), ou à une référence des variables précitées (entrée ou sortie).

5.1.1 Nuls et booléens

Pour passer un argument de valeur nulle il est nécessaire d'utiliser le mot-clé `nil`. Un argument de valeur nulle peut être compris comme un argument dépourvu de valeur. Pour la valeur booléenne vraie, le mot-clé est `true` et pour la valeur fausse le mot-clé est `false`. Nous avons adopté cette convention en accord avec les mot-clés `nil | false | true` qui peuvent être utilisés avec une connotation similaire dans la définition des fonctions. Par ailleurs, il est utile de remarquer que ces trois mot-clés doivent toujours être écrits en respectant la casse.

5.1.2 Chaînes de caractères littérales

Pour construire un argument contenant une chaîne littérale de caractères, telle que « chaîne » ou « autre chaîne », il est juste requis d'écrire la suite de caractères, sans aucun autre symbole particulier l'entourant. Autrement dit, une chaîne littérale est uniquement délimitée par les virgules ou par les parenthèses qui séparent les arguments. Afin d'utiliser les caractères qui ont un sens spécial dans la syntaxe d'appel dans une fonction étendue, tels que le symbole dollar (`$`) ou « et » commercial (`&`), l'implémentation courante permet de les déspécialiser (mode d'« échappement »). Ainsi, pour utiliser ces symboles dans une chaîne littérale, il suffit de les redoubler : `$$` → `$`, `&&` → `&`. En outre, dans une liste d'arguments non-vide, une chaîne littérale est vide si elle ne contient pas de caractères et dans ce cas elle prend la valeur nulle. Finalement, les chaînes littérales peuvent se concaténer aux variables que nous décrivons ensuite. Il est aussi utile de remarquer que la longueur maximale d'un argument du type chaîne littérale est de 4095 caractères.

5.1.3 Variables d'entrée et de sortie

Il est possible de passer un argument à une fonction étendue contenant la valeur d'une variable d'entrée ou de sortie. Contrairement à la convention utilisée dans Unitex pour faire référence aux variables en encadrant leur nom avec le caractère `$`, par exemple `$variable$`, dans les arguments d'une fonction étendue les valeurs des variables sont indiquées en utilisant la convention :

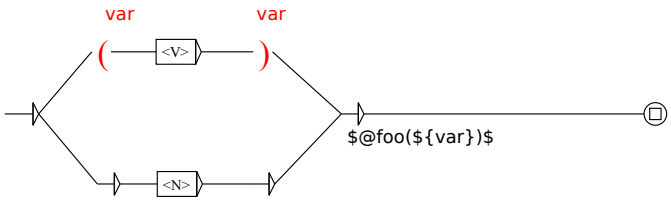
```
${variable}
```

Le nom de la variable est entouré par des accolades (`{` et `}`) et préfixé par le symbole dollar `$`. Lorsqu'une variable est retrouvée, sa valeur (une chaîne de caractères) est assignée à l'argument passé à la fonction. Si la variable fait partie d'un argument du type chaîne littérale, alors sa valeur est concaténée à la chaîne dans laquelle se trouve.

En outre, les variables non assignées, qui ne font pas partie d'un argument du type chaîne, sont assignées par défaut à la valeur nulle. Dans le traitement des arguments des fonctions étendues, ce comportement est plus pertinent que celui d'utiliser une politique unique (manuel, p. 154) lorsqu'une variable n'est pas définie. En effet, cela permet d'autoriser des chemins qui définissent une variable ainsi que d'autres chemins qui ne la définissent pas sans considérer ces derniers comme des erreurs (voir le graphe 5.3). Il est alors de la responsabilité de la fonction étendue (l'implémentation) de prendre en charge les deux cas. Nous avons trouvé cette approche utile dans la réalisation de certaines tâches qui sont prises en charge par des fonctions étendues :

- D'une part, pour générer des sorties de graphes dictionnaires (manuel, p. 69) avec une fonction étendue qui met en forme des ensembles de variables éventuellement non définies.

- D'autre part, pour passer des variables, éventuellement non définies, à des fonctions étendues qui ont la charge de faire des requêtes à une ressource externe telle qu'une base de données.



Graphe 5.3 – Variable défini par un seul chemin

5.1.4 Chaînes de caractères non-littérales

Il est possible de construire des arguments qui combinent des chaînes littérales et de variables dans un seul argument, par exemple : `littéral ${variable}`. L'argument qui est passé à la fonction étendue est alors composé de parties littérales et de variables concaténées.

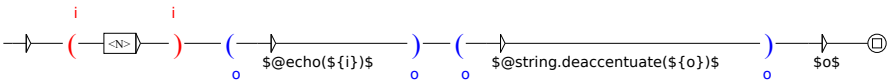
Les règles utilisées pour obtenir les valeurs des variables sont quasiment les mêmes que celles citées dans la partie *variables d'entrée et sortie* (cf. sous-section 5.1.3). Elles ne diffèrent que dans le cas du traitement des variables non assignées. Lorsqu'une variable isolée n'est pas définie, elle se voit affecter la valeur nulle. Par contre, lorsqu'elle fait partie d'un argument combinant chaînes littérales et variables, elle est remplacée par une chaîne vide (contenant zéro caractères) avant d'être concaténée. Par exemple, si `${variable}` est non défini, alors l'argument `littéral ${variable}` sera égal à `littéral`.

5.1.5 Références aux variables d'entrée et de sortie

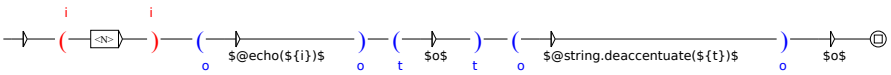
Nous avons vu qu'en utilisant la convention `${variable}`, il est possible de passer un argument à une fonction étendue contenant la valeur d'une variable d'entrée ou de sortie. Une des tâches pour appeler une fonction étendue est donc de résoudre d'abord, pour chaque argument, les variables utilisées avant de construire l'argument final qui sera passé pour évaluer la fonction. Par résoudre nous entendons la transformation d'un nom de variable en une chaîne de caractères. Considérons à présent deux types de scénario où la notion de référence à une variable est utile :

- Il est nécessaire de réaliser une opération sur une variable de sortie et de stocker le résultat dans la même variable, par exemple, mettre la variable de sortie en majuscules, la convertir en une sous-chaîne, lui supprimer les accents, la translittérer, etc. En utilisant le mode de passage par valeur, la démarche pour y parvenir consisterait à appeler la fonction étendue qui réalise le processus : `$@fonction(${variable})$` et à stocker le résultat de l'appel dans la même variable. Cependant, cette stratégie ne s'avère pas satisfaisante. Il n'est pas possible de résoudre l'ancienne valeur d'une

variable lorsqu'elle est redéfinie (voir le graphe 5.4). Comme alternative, il est possible de créer une variable temporaire afin d'éviter de perdre la valeur affectée avant la réassignation (voir le graphe 5.5). Une approche plus simple, et moins coûteuse, consisterait à établir une notion de passage par référence des variables.

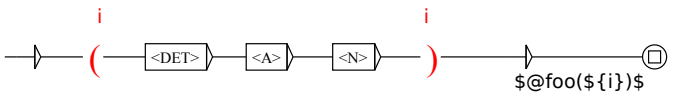


Graphe 5.4 – Variable redéfinie et passage par valeur



Graphe 5.5 – Variable temporaire et passage par valeur

- Il est requis de connaître ou de traiter directement les *tokens* qui composent une variable d'entrée. Rappelons qu'en utilisant les variables d'entrée il est possible de stocker des parties du texte reconnu (manuel, p. 105). Lorsqu'un argument contient une variable d'entrée, c'est la valeur de la variable, c'est-à-dire, la chaîne de caractères composée de la concaténation de chacun des *tokens*, séparés par des espaces, qui est utilisée. Étant donnée l'entrée « a deep scar on his brow », le graphe graphe **??** affecte la variable *i* à la séquence « a deep scar ». Il faut noter alors que pour la construction des arguments d'une fonction étendue, une fois la variable convertie en un chaîne de caractères, elle perd le lien avec les *tokens* qui la composent. Si dans une grammaire locale ce comportement peut être suffisant, dans une grammaire locale étendue il pourrait être souhaitable de réaliser une opération dépendant ou utilisant une partie des *tokens* d'une variable d'entrée. Comme pour le scénario précédent, une approche pour y parvenir consisterait à établir une notion de passage par référence des variables.



Graphe 5.6 – Variable composée par plusieurs *tokens*

Le passage par référence des variables implique que l'argument d'une fonction ne prenne pas la valeur de la variable mais une référence. La convention que nous avons adoptée pour passer une variable par référence est la suivante :

```
&{variable}
```

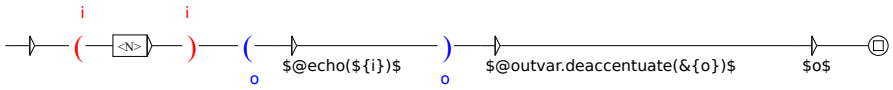
Ainsi, le fait de remplacer le symbole dollar par « et » commercial (`&`), indique que nous souhaitons obtenir non pas la valeur mais la référence de la variable. Les références aux variables d'entrée ou de sortie sont de nature différente.

Références à des variables d'entrée : Une référence à une variable d'entrée est passée comme une liste de constantes numériques $r_i = \{n_1, n_2, n_3, n_4\}$ indiquant le rang absolu $[n_1, n_2[$ des *tokens* utilisés pour construire la valeur référencée. Chacune de valeurs se rapporte à la version du texte tokenisé (manuel, p. 38) et correspond à :

n_1 = Position du premier *token*
n_2 = Position après le dernier *token*
n_3 = Position de début (en caractères) dans le premier *token*
n_4 = Position de fin (en caractères) dans le dernier *token*

Comme exemple, dans le graphe (inputvarref1), tandis que `${i}` est transformé en « a deep scar », `&{i}` est converti en la liste $\{0, 3, 0, -1\}$. Notons en effet que a=0 ; depp=1 ; scar=2. Autrement dit, n_1 prend la position du premier *token* (0), n_2 la position après le dernier *token* $(2 + 1)$, n_3 la position de début (en caractères) dans le premier *token* (0), n_4 la position de fin (en caractères) dans le dernier *token*, où -1 signifie prendre en compte tout le *token*. Nous avons adopté cette convention des 4 entiers car il correspond à la modélisation interne de variables dans UNITEX.

Références à des variables de sortie : Une référence à une variable d'entrée est passée comme un argument du type référence `elg.ustring`, `ustring` (*Unitex String*) est une bibliothèque de traitement de chaînes de caractères spécialement conçue pour être utilisée avec les grammaires locales étendues. Ainsi, lorsqu'un paramètre d'une fonction prend une référence du type `elg.ustring`, les changements réalisés sur le paramètre sont affectés directement à la variable de sortie. Le graphe 5.7 constitue ainsi une alternative au graphe 5.5.



Graphe 5.7 – Variable passée par référence

Finalement, il est pertinent de préciser que contrairement aux variables passées par valeur, une variable passée par référence ne peut pas être combinée à une chaîne de caractères. Ceci est dû naturellement à l'incompatibilité entre les deux types.

5.1.6 Constantes numériques

Dans l'implémentation actuelle, les arguments du type constantes numériques, telles que 9, 98.4 ou $0.2e17$ sont passées en tant que chaînes de caractères. Cette méthode de passage ne répond à aucune contrainte particulière et pourrait évoluer dans des versions à venir. Entre temps, il est possible de convertir un paramètre `p` en un nombre en utilisant l'expression `n = tonumber(p)` dans la définition de la fonction étendue.

5.2 Un langage interprété comme infrastructure des fonctions étendues

Lorsque nous avons envisagé l'option d'utiliser un langage interprété comme infrastructure des fonctions étendues, nous avons passé en revue plusieurs langages de programmation, entre autres : C++, Java, Python, Haxe, Javascript et Lua.

Nous avons finalement porte notre choix par Lua (Ierusalimschy et al., 1996, 2007), un langage de programmation de scripts extensible et portable. Cette décision a été motivée par plusieurs aspects :

- Lua fournit une syntaxe concise et simple pour décrire des données ce qui le rend utile comme un langage de configuration et d'échange d'information. Cette simplicité est aussi présente lors de la manipulation des tables dynamiques, la seule structure de données disponible en Lua et qui sert à représenter tout autre type abstrait tel que les chaînes de caractères.
- Le langage offre un bon niveau d'abstraction en rapport à l'architecture matérielle où le langage est interprété.
- Il permet le prototypage et les tests rapides.
- L'interprète du langage dispose d'un sous-système de gestion automatique de la mémoire. Ce sous-système, non-intrusif, permet d'écrire des programmes plus simples du fait qu'il libère le développeur de la tâche de gestion manuelle de la mémoire.

Dont une particulièrement attractive afin de mettre en œuvre le formalisme des grammaires locales étendues sur Unitex. Il s'agit de la capacité de Lua d'être appelé et de communiquer avec d'autres logiciels écrits en C et C++, ce qui font de lui un langage propice pour le développement de briques logiciels modulaires, chacune regroupant des fonctionnalités bien définies.

En outre, Lua peut être aussi utilisé comme langage de configuration, par exemple dans Roman et al. (2002), il est employé pour paramétrer et coordonner des dispositifs informatiques dans un environnement informatique ubiquitaire. Il peut aussi prendre le rôle de langage d'extension, tel que dans Ernst et al. (2009), où il sert à étendre les fonctionnalités d'un framework destiné à la détection des gestes et l'analyse des visages. Finalement, Une autre caractéristique importante est que l'interprète de Lua est rapide et a une faible empreinte mémoire [1].

Dans le cadre des grammaires locales étendues (ELGs), Lua est utilisé à deux fins : comme extension du moteur d'analyse des grammaires locales et comme langage pour implémenter des fonctions étendues.

1. Pour d'autres détails techniques, consultez le livre de référence de Lua (Ierusalimschy, 2016)

5.2.1 Procédure d'exécution d'une fonction étendue

- Création d'un λ-state vide
- Définition des fonctions autorisées dans le λ-state
- Chargement du fichier .upp contenant la fonction étendue
- Compilation de la fonction étendue
- Empilement (push) des paramètres de la fonction étendue dans la λ-pile
- Exécution du code compilé de la fonction étendue
- Désempilement (pop) des valeurs de retour qui se trouvent dans la λ-state

5.3 Opérations et événements

- Opérateurs : il s'agit des opérateurs disponibles sur les chaînes, sur les *tokens*, sur les états et sur les ressources.

- Événements : il s'agit des événements `init_event`, `load_event`, `unload_event`, `token_event`, `slide_event`, `onenter_event`, `onexit_event`, `onbacktrack_event`,`onmatch_event`, `onfail_event`.

5.4 Chaîne de traitement

5.5 Application d'une grammaire locale étendue

Les étapes d'application d'une grammaire locale étendue sont issues de la stratégie de représentation des *tokens* proposée par Paumier (2003b) et mise en œuvre dans Unitex. La démarche se distingue notamment par la façon de traiter et construire les sorties, par conséquent dans la façon d'analyser et évaluer les fonctions étendues, lors de l'application de la grammaire.

Comme illustré sur la figure 5.1, le processus global est divisé en trois étapes : Découpage en *tokens*, Optimisation de la grammaire et Application finale de la grammaire.

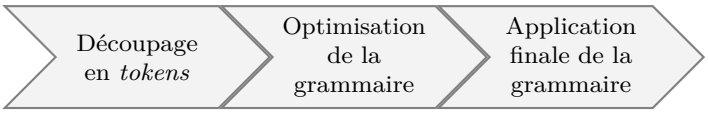


Figure 5.1 – Étapes d'application d'une grammaire locale étendue à un texte

5.5.1 Découpage en *tokens*

Pour les langues comportant une séparation précise de mots, un *token*, est soit une séquence de lettres, soit un caractère non-alphabétique. Étant donné un texte d'exemple :

« *The incident took place on Tesday*[1] *the 2nd of May of 1969* », telle qu'illustrée à la figure 5.2, nous découpons le texte en *tokens* en leur associant un entier unique[2]. Ce processus est réalisé en un seul passage et en un temps proportionnel à la taille du texte, soit $\mathcal{O}(n)$. Il est également indépendant des types ou du nombre de grammaires qui seront appliquées par la suite. Une fois le texte balayé, une liste indexée de *tokens* comme celle de la figure 5.3 est finalement constituée. Le texte peut alors être représenté comme une succession d'entiers chacun correspondant à l'indice du *token* qui lui est associé. Nous présentons un exemple dans la figure 5.4.

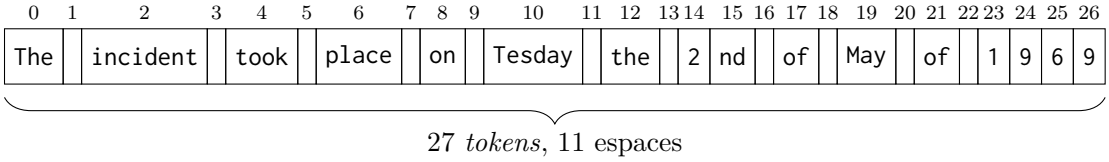


Figure 5.2 – Découpage : *tokens*

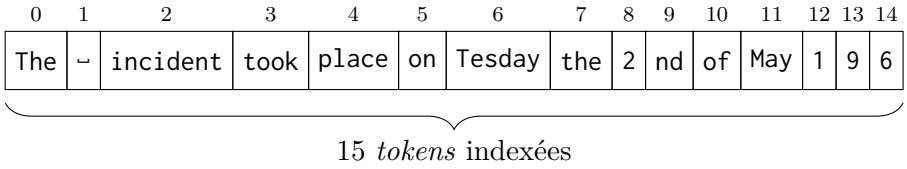


Figure 5.3 – Découpage : *tokens* indexées

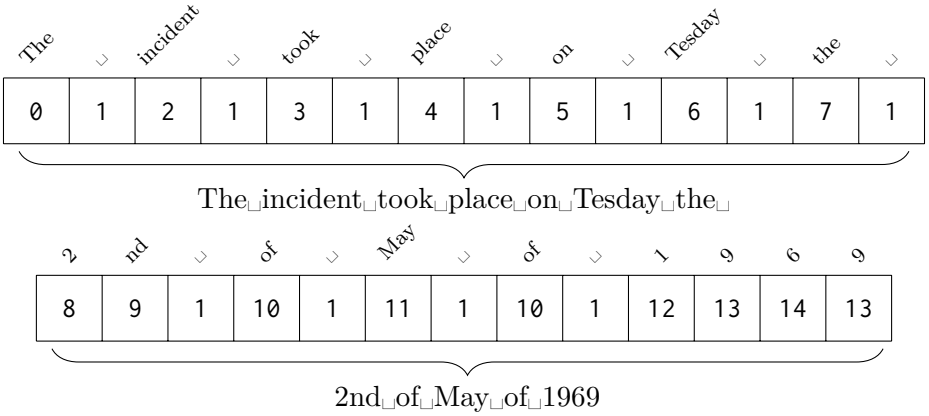


Figure 5.4 – Découpage : représentation sous forme d'indices

1. *Tuesday : La faute d'orthographe est fournie à titre d'illustration*
2. Ceci est l'approche utilisée par Unitex où les *tokens* sont représentées par des entiers, ainsi que les grammaires lors de leur phase d'application

5.5.2 Optimisation de la grammaire

Étant donné une grammaire dont les étiquettes de transition sont des *tokens* conformes à la règle de découpage utilisée au cours de la première étape, les trois techniques d'optimisation proposées par Paumier (2003b) sont mis en place :

Remplacer les *tokens* par des entiers

Chaque transition marquée par un *token* est remplacé par la liste d'entiers à laquelle il peut correspondre. Dans le cas exemplifié, le token *the* sera remplacé par 0_{the} et par 12_{The}. La complexité de cette étape est de $\mathcal{O}(N)$, où N est la taille de l'alphabet d'entrée.

Créer des références à des classes de mots :

Les classes de mots, qui établissent un lien avec les ressources linguistiques, sont référencées dans les transitions de la grammaire au moyen des masques lexicaux (cf.tableau II). Lors de l'application de la grammaire au texte, vérifier si une unité lexicale correspond à un masque lexical exige la consultation répétitive des ressources linguistiques ce qui rend la grammaire non-déterministe du fait que plusieurs masques pourraient reconnaître la même unité lexicale.

Unités lexicales simples : Pour pallier ce problème, en ce qui concerne les unités lexicales simples, la stratégie adoptée est basée tout d'abord sur le comptage du nombre d'unités lexicales qui correspondent à un masque lexical. Ensuite, le masque est remplacé par la liste complète des entiers représentant les unités lexicales auxquels il peut correspondre. Dans le cas contraire, le masque est indexé par un entier et pour enregistrer le fait que l'unité lexicale correspond au masque lexical une structure auxiliaire de données est utilisée.

Cette structure est un tableau de bits associé à chaque *token*. Le masque d'indice n vérifie une unité lexicale lorsque le $n^{\text{ième}}$ bit de leur tableau est mis à 1. Ainsi, prouver si une unité lexicale quelconque correspond à un masque lexical, se réalise en deux temps : le premier par le numéro de l'unité lexicale, le deuxième par le numéro du masque lexical.

Unités lexicales composées : S'agissant des unités lexicales composées, la stratégie suivie marche comme une extension de la précédente. Dans un premier temps, tous les motifs qui peuvent correspondre à une unité lexicale composée sont vérifiés, ensuite un arbre est construit à partir de la liste de leur composants qui vérifient au moins un masque lexical. Chaque nœud de l'arbre correspond à l'entier qui est associé à l'unité lexicale. Chaque nœud terminal mémorise également les numéros des masques qui correspondent à l'unité lexicale composée.

Savoir si un masque lexical quelconque correspond à une ou plusieurs unités lexicales composées peut se réaliser en explorant simultanément l'arbre et la représentation

indexée du texte. L'hypothèse est alors vérifiée, si lors de cette exploration l'algorithme atteint un nœud final et si parmi les numéros des masques stockés se trouve celui recherché.

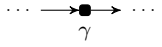
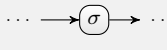
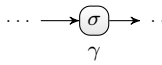
Trier les listes des entiers :

Finalement, à l'issue des optimisations précédentes, les transitions sont de deux types, soit des listes d'entiers représentant les tokens (transitions-*token*), soit des masques représentant les classes de mots (transitions de classe).

Étant donné un état de la grammaire, vérifier si la transition peut correspondre à un *token* du texte équivaut à comparer leur numéro avec ceux de la transition. Pour améliorer le temps de ce traitement, les transitions de lexique sont fusionnées en une liste unique ordonnée de n entiers, qui par recherche dichotomique, pourra être consultée en $\mathcal{O}(log(n))$.

5.5.3 Construction des sorties

Lorsque une fonction étendue fait partie d'une sortie, des politiques supplémentaires à celles d'ignorer les sorties, les remplacer ou les insérer à gauche des séquences manuel, p. 146 peuvent être mises en place, il s'agit d'un mode *mettre à jour* avec priorité, leur construction est décrit au tableau 5.1.

Cas	Σ_{input}	Γ_{output}	ELGs					
			LGs					
			$\mathcal{I}_{gnore}$	$\mathcal{R}_{eplace}$	$\mathcal{M}_{erge}$	$\mathcal{U}_{pdate_\sigma}$	$\mathcal{U}_{pdate_\gamma}$	$\mathcal{U}_{pdate_\varepsilon}$
	ε	ε	ε	ε	ε	ε	ε	ε
	ε	γ	ε	γ	γ	γ	γ	γ
	σ	ε	σ	ε	σ	σ	σ	σ
	σ	γ	σ	γ	$\gamma \cdot \sigma$	γ	σ	ε

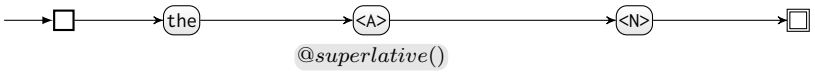
Cas	Σ_{input}	Γ_{output}	ELGs					
			LGs					
			$\mathcal{I}_{gnore}$	$\mathcal{R}_{eplace}$	$\mathcal{M}_{erge}$	$\mathcal{U}_{pdate_\sigma}$	$\mathcal{U}_{pdate_\gamma}$	$\mathcal{U}_{pdate_\varepsilon}$

Tableau 5.1 – Politiques pour la construction des sorties des ELGs

Afin de comprendre leur utilité prenons comme exemple la phrase (13) :

(13) *The crazy things*

Ainsi que la grammaire locale étendue du graphe 5.8 ci-après :



Graphe 5.8 – Exemple de la construction des sorties avec une ELG

L'appel à une fonction étendue `superlatif` renvoie le superlatif de l'étiquette d'entrée courante qui est directement accessible à partir de la fonction étendue. Les possibles sorties sont énumérées dans le tableau 5.2.

Nom	Description	Sortie
$\mathcal{I}_{gnore}$	Le résultat de la concaténation des étiquettes d'entrée appartenant à chacun des chemins réussis stockées (c'est-à-dire ignorant les étiquettes de sortie)	*The crazy things*
$\mathcal{M}_{erge}$	Le résultat de la concaténation consistant à fusionner les étiquettes de sortie (à gauche) avec les étiquettes de sortie appartenant à chacun des chemins réussis stockés (c'est-à-dire les sorties sont insérées à gauche des séquences reconnues)	*The craziest crazy things*
$\mathcal{R}_{eplace}$	Le résultat de la concaténation consistant à remplacer les étiquettes d'entrée par les étiquettes de sortie appartenant à chacun des chemins réussis stockés (c'est-à-dire en remplaçant les étiquettes d'entrée par des étiquettes de sortie)	*craziest*
$\mathcal{U}_{pdate_\sigma}$	Le résultat de la concaténation des étiquettes d'entrée avec les étiquettes de sortie étendues appartenant à chacun des chemins réussis stockés (c'est-à-dire le remplacement des étiquettes d'entrée par des étiquettes de sortie étendues)	*The craziest things*

Tableau 5.2 – Construction de sorties pour le graphe 5.8