Ecriture d'un algorithme pour la conjugaison de l'imparfait de l'indicatif.
Stéphane Clément, Jean-Paul Berroir, Jean-Baptiste Civet, groupe MATHICE

Ce document présente un exemple de démarche algorithmique pour la génération de conjugaisons. Une des difficultés des conjugaisons est le nombre très important d'exceptions à gérer. Il paraît pertinent de sélectionner le temps ou une langue les plus réguliers possibles. Ici, nous proposons l'imparfait de l'indicatif en français.

Phase 1: langage naturel, on cherche des règles de génération de l'imparfait, elles mêmes exprimées en français. Puis on essaye de découper en sous-problèmes de plus en plus petits.

Source Wikipédia "imparfait de l'indicatif en français": ce n'est pas la meilleure source, de nombreux détails y sont omis, mais c'est un point de départ classique.

> **Formation de l'imparfait** [modifier | modifier le code]
>
> Pour former l'imparfait on utilise le radical du verbe à la première personne du pluriel au présent de l'indicatif. On ajoute à ce radical les terminaisons de l'imparfait.
>
> - p.ex. : nous **aim**ons donner J' **aim**-ais, -ais, -ait, -ions, -iez, -aient.
> - exception : le verbe **être**

Une seule exception, c'est bien, on peut se concentrer sur le reste, et traiter le verbe être à part, plus tard.
Le problème se décompose donc en :
- former le radical de la 1ère personne du pluriel au présent;
- former les 6 successions pronom / radical / terminaison, les pronoms étant la liste "je tu il nous vous ils", les terminaisons "ais ais ait ions iez aient".

Ça a l'air simple, mais la génération de la première personne du pluriel au présent est très compliquée. Rien que pour le premier groupe on doit gérer les verbes en -ger (qui intercalent un e, nous mangeons) et en -cer (qui change le c en ç, nous commençons). Le deuxième groupe semble plus régulier, mais le problème, c'est de les reconnaître. Ainsi "finir" est du 2° groupe (nous finissons les miettes) mais pas "mourir" (nous mourons de faim), la terminaison ne suffit pas à détecter ce groupe. On ne parle même pas du 3° groupe !

Il faut donc simplifier.
Une solution qui évite le problème, et permet un projet de taille acceptable: demander à l'utilisateur de former la première personne du pluriel. Il ne reste plus qu'à enlever le "ons" final. Sauf que:
"nous mangeons" donne un radical de "mange": je mangeais est ok, mais pas nous mangeions.
"nous commençons" donne un radical de "commenç", je commençais est ok mais pas nous commençions. Il faut donc tester la lettre avant le "ons".

Algorithme en langage naturel
demander la première personne du pluriel au présent
enlever le ons final, créer la liste des radicaux en dupliquant 6x le résultat
si la dernière lettre est 'e', enlever le e final des 4° et 5° éléments de la liste
si la dernière lettre est 'ç', remplacer le 'ç' par 'c' dans les 4° et 5° éléments de la liste
créer la liste des pronoms: 'je tu il nous vous ils' (éventuellement j' à la place de je si la première lettre est une voyelle)
créer la liste des terminaisons : 'ais ais ait ions iez aient'
afficher les 6 successions pronom/radical/terminaison.

Phase 2: passer au codage en pseudo-langage
L'algorithme en langage naturel précise la structure, mais n'est pas directement traduisible sur machine. Il faut maintenant détailler les opérations que scratch ne sait pas directement faire.

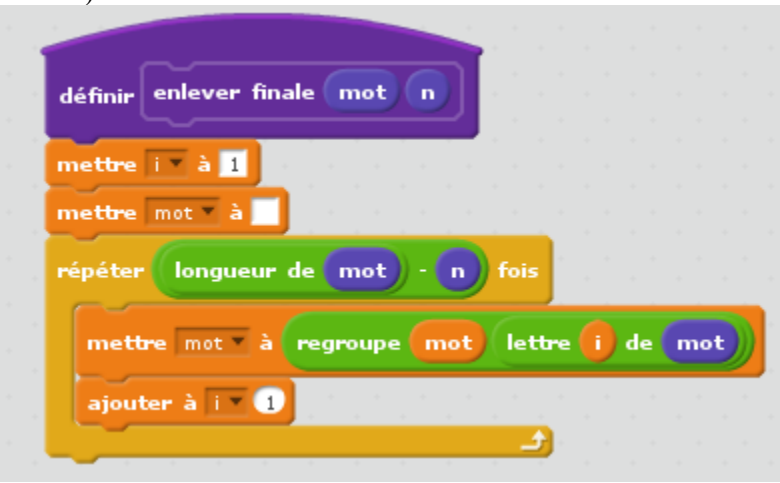
Les limites de scratch: ce logiciel est avant tout pensé pour créer des animations graphiques, la manipulation de lettres et de mots n'est pas son fort. Ainsi, supprimer les 3 dernières lettres du verbe saisi va demander des manipulations un peu pénibles, alors qu'en python, on écrirait simplement: radical=verbe[:-3]

Une bonne pratique est (avec Scratch v2) de créer des blocs de commandes correspondant aux tâches auxiliaires de l'algorithme en langage naturel, et ainsi de ne préciser l'algorithme que sur des parties bien délimitées. Ainsi, on peut créer un bloc "supprimer les 3 dernières lettres".

Par exemple, voici un bloc générique permettant de supprimer les *n* dernières lettres d'un mot:

bloc supprimer les n dernières lettres de mot
i vaut 1
résultat vaut: chaine vide
répéter longueur de mot – n fois :
    ajouter à résultat la lettre numéro i de mot
    ajouter 1 à i

et son implémentation dans scratch (nécessitant l'existence d'une variable "mot" qui stockera le résultat et du compteur "i"):

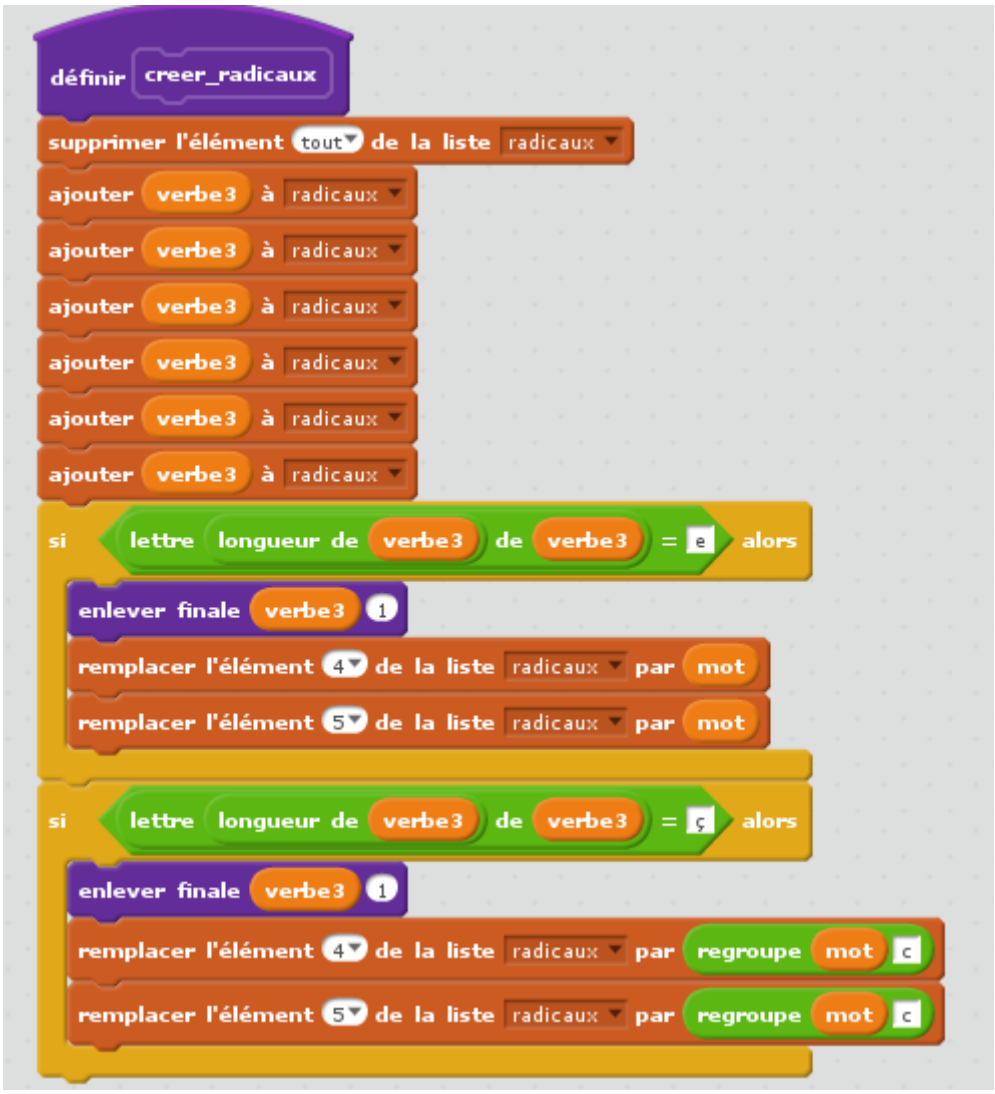


Remarques pour ceux qui ont des notions de programmation:
- il n'y a pas de notion de variable locale au bloc de commande, hormis les paramètres de ce bloc (en bleu), qui ne sont utilisables que dans ce bloc, et peuvent porter le même nom que les variables globales: comparer les variables "mot" en bleu (paramètre) et en orange (variable globale).
- on peut préciser à la création d'une variable ou d'une liste qu'elle n'est connu que de son lutin, ou de tous. Ici, on n'utilise qu'un lutin pour éviter de gérer la communication entre lutins.

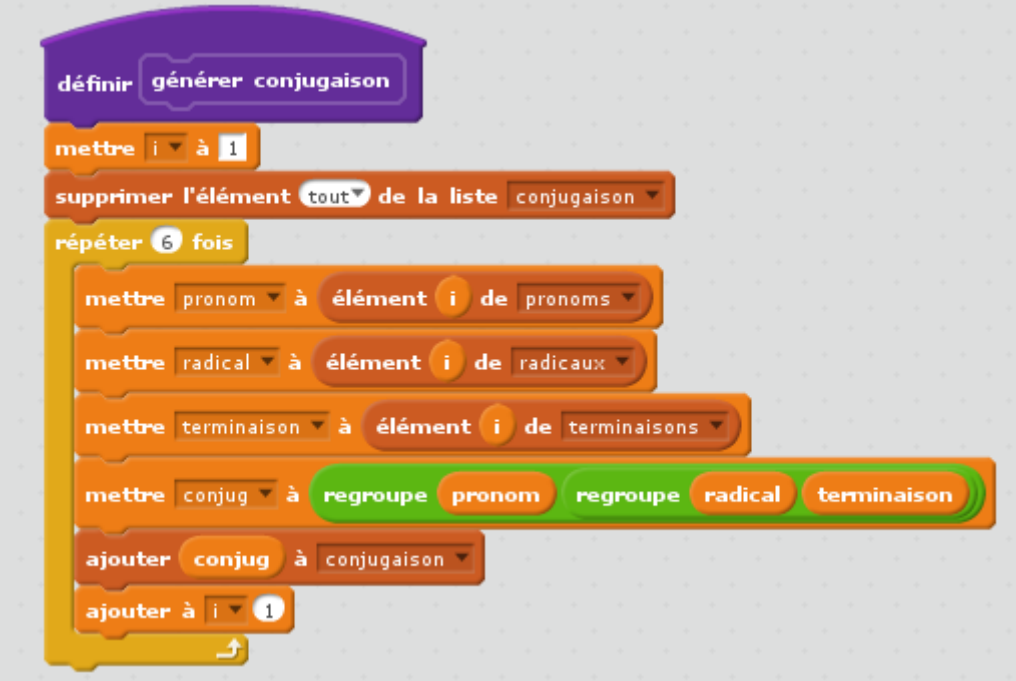
On développe alors bloc de commande par commande, chaque bloc étant testé isolément.

Avec l'utilisation de blocs, le script principal est très lisible, et du coup, facilement améliorable: dans cette version, le problème du pronom "je" élidé ou non n'est pas (encore) traité:

Nous donnons ci-dessous les deux blocs les plus compliqués, directement en Scratch: créer radicaux et générer conjugaison.



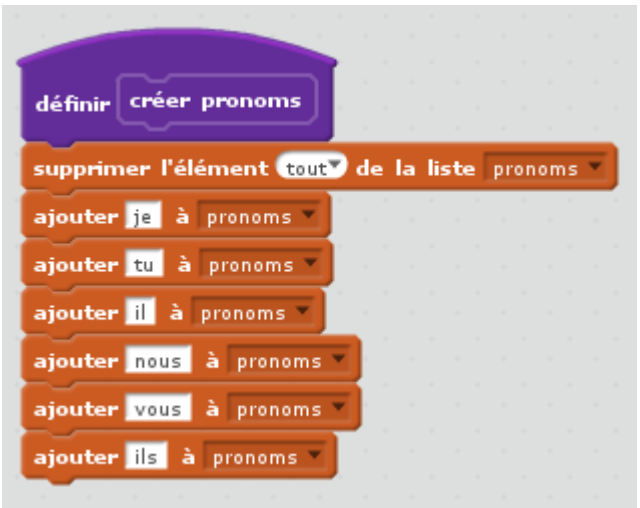
Une bonne pratique est de créer des blocs pas trop longs, et faciles à lire. Plus facile à dire qu'à faire!

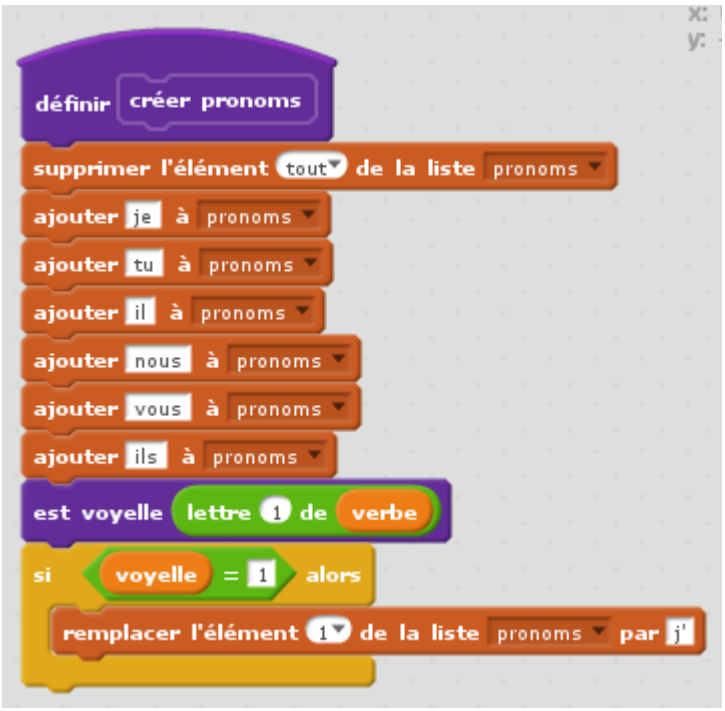


Dans cette version, 'générer pronoms' crée la liste 'je tu il nous vous ils': la première amélioration à envisager est de gérer le 'je' élidé, et de donner la possibilité de conjuguer au féminin.

Le fait d'avoir utilisé des blocs de commande permet que ces futures modifications se fassent uniquement à l'intérieur du bloc 'générer pronoms', sans impacter le programme principal.

Voici la modification, qui est faite ultérieurement quand on est content du reste:





cette modification nécessite la définition d'une variable voyelle et d'un bloc de détection de voyelles, non détaillé.

Quelques remarques pour finir :

- éviter une excessive rigueur dans les pseudo langage, c'est souvent inutile, surtout si on a pris soin de bien découper le problème en sous problèmes associés à des blocs: l'algorithme contient alors beaucoup d'appels de blocs et est souvent suffisant en "langage naturel".
- une syntaxe "à la python" est beaucoup plus souple qu'une syntaxe "à la algobox", comparer:

```
Pseudo langage à la python
si lettre=e :
        supprimer dernière lettre
si lettre=ç:
        changer dernière lettre en c
```

```
Pseudo langage à la algobox
Si lettre=e Alors
        supprimer dernière lettre
Fin Si
Si lettre=ç Alors
        changer dernière lettre en c
Fin Si
```

- ne pas essayer de résoudre toutes les difficultés d'un coup ! Elles sont tellement nombreuses qu'on s'y perd vite.
-et pour finir :