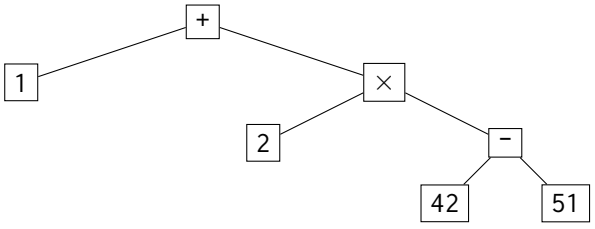
TP nº 3

Préliminaires

Il vous est demandé de lancer votre page avec Google Chrome. Le plus simple est de lancer Chrome puis « Ctrl-O » et d'ouvrir le fichier HTML directement. On suppose que votre répertoire de travail est un dépôt `git`. Il vous appartient maintenant de faire des `add`/`commit` judicieux pour sauvegarder l'historique de vos modifications.

Formules

On s'intéresse ici à l'analyse lexicale et syntaxique d'une formule, c'est à dire au processus permettant de transformer une chaîne de caractères, telle que `"1 + 2 × (42 - 51)"` en un objet javascript ayant la structure suivante :



1 Objets pour l'AST

On souhaite modéliser d'abord l'arbre de syntaxe abstraîte (*i.e.* la représentation interne d'une formule, sous la forme d'un objets Javascripts. On se limite dans un premier temps aux formules contenant les quatres opérations et les constantes numériques.

1. Récupérer le fichier `formula.js` sur la page du cours, le placer avec le reste des fichiers Javascript du projet et rajouter un élément `script` à la page HTML permettant de charger ce fichier `formula.js`. Ce dernier doit être chargé avant le fichier `cell.js`.
2. Lire attentivement la déclaration du constructeur `Formula` ainsi que ses méthodes de prototype, de même que la sous-classe `Const` représentant les constantes numériques. Compléter le code des quatres sous-classes restantes : `Add`, `Sub`, `Mul`, `Div` représentant les 4 opérations. Chacun de ces objets devra contenir deux méthodes de prototype :

   `eval()` renvoyant la valeur de la formule

   `toString()` renvoyant une chaîne de caractères représentant la formule.

   Dans le cas des quatres opérations, on fera un usage judicieux de `evalChildren()` et `childrenToString()` afin d'évaluer/afficher récursivement les nœuds fils de la formule.

2 Analyse Syntaxique

L'analyse syntaxique est découpée en deux phases. En premier lieu, on découpe la chaîne de caractères donnée en entrée en un tableau de *tokens* (unités lexicales). En deuxième lieu, on essaye de grouper ensemble les *tokens* selon les règles de l'arithmétique afin de créer un arbre syntaxique.

1. Récupérer le fichier `lexer.js` sur la page du cours, le placer avec le reste des fichiers Javascript du projet et rajouter un élément `script` à la page HTML permettant de charger ce fichier `lexer.js`. Ce dernier doit être chargé avant le fichier `formula.js`.

2. L'objet `Lexer` défini dans le fichier `lexer.js` permet de découper une chaîne de caractères en tableau d'objets. L'interface pour utiliser un `Lexer` est la suivante :

```
var actions = [ { re : /[0-9]+/, action : function (s, i, j) {
                                            return (s - 0);
                                          } },
                { re : /[A-Z]+/, action : function (s, i, j) {
                                            return s;
                                          } } ];

var lexer = new Lexer (actions);

var tokens = lexer.scan ( "A B 123 DEDE 24090, !, 49");
console.log(tokens);
//affiche : ["A", "B", 123, "DEDE", 24090, 49]
```

Comme on le voit, le tableau `actions` utilisé pour initialiser le *lexer* est un tableau d'objets. Chaque objet possède une propriété `re` contenant une expression régulière et une propriété `action` contenant une fonction. Supposons que le *lexer* a été initialisé avec un tableau d'objets :

```
[
  { re: re_1, action: f_1,  },
  . . .,
  { re: re_n, action: f_n,  }
]
```

alors effectuer `lexer.scan(input)` correspond à effectuer le pseudo code suivant :

(a) si à partir de la position courante i, il existe une sous-chaîne s qui satisfait une certaine expression régulière $\texttt{re}_k$:

i. on calcule $\texttt{res = f}_k\texttt{(}s\texttt{, }i\texttt{, }i + s\texttt{.length)}$ et on ajoute `res` au tableau des résultats.

ii. si on n'est pas à la fin de `input`, avancer de s`.length` caractères dans `input` et reprendre en (a).

(b) sinon on avance de un caractère

Par exemple, avec le code ci-dessus, on commence à la position `0`. La deuxième expression régulière fonctionne, pour une longueur de `1` caractère (`A` et on exécute la deuxième action (qui ne fait que renvoyer son entrée) et on place donc `"A"` à la première position du tableau de résultat. On avance ensuite d'un caractère. Aucune expression régulière ne commence par `" "`, on avance donc de un caractère (cas (b)). On avance ainsi en trouvant successivement `"B"` puis `"123"`, ...et en ignorant les caractères qui ne sont pas exactement reconnus.

Compléter dans le fichier `formula.js` les expressions régulières aux endroit demandés. Attention, si vous souhaitez grouper des expressions régulières, il faut absolument utiliser la syntaxe : `(?:`*regex*`)` afin de ne pas créer de sous-groupes d'expressions.

3. Lire attentivement la fonction `Formula.parse` et y implémenter, à l'endroit indiqué, l'algorithme de Dijkstra pour la reconnaissance d'expressions arithmétiques[1]. Dans le pseudo-code ci-dessous, on dit que l'on `réduit` un opérateur sur `output` pour dire que l'on effectue l'opération `output.reduce(t)` (pour un opérateur `t`, cf. le fichier `formula.js`). Dans votre programme, vous pourrez signaler une erreur de syntaxe avec la construction :

```
throw "Message d'erreur";
```

qui lève une exception.

(a) Initialiser deux tableaux `output` et `stack` vides.

(b) Phase I : pour chaque *token* présent dans l'entrée :

1. Voir : `https://en.wikipedia.org/wiki/Shunting-yard_algorithm` pour la version complète, on n'en donne ici qu'une version simplifiée pour nos expressions

i. Soit *t* le *token* courant

ii. Si *t* est un nombre (objet `Const`) alors le réduire sur `output`

iii. Si *t* est un opérateur (`Add`, `Sub`, `Mul`, `Div`))
   — tant qu'il existe un autre opérateur *t'* au sommet de `stack`, dont la précédence est inférieure ou égale à celle de *t* ; retirer *t'* de `stack` et le réduire sur `output`.
   — Placer ensuite *t* sur `stack`.

iv. Si *t* est une parenthèse ouvrante, la placer sur `stack`.

v. Si *t* est une parenthèse fermante alors :
   — Dépiler les opérateurs (`Add`, `Sub`, `Mul`, `Div`)) de `stack` et les réduire sur `output` jusqu'à trouver une parenthèse ouvrante
   — Dépiler la parenthèse ouvrante (sans la placer dans `output`)
   — Si `stack` est vide, lever une erreur (expression mal parenthésée)

(c) Phase II : tant qu'il reste des opérateurs qui ne sont pas des parenthèses sur `stack`, les réduire sur `output`. Si on trouve une parenthèse, lever une erreur (expression mal parenthésée)

(d) Si le tableau `output` ne contient pas exactement `1` entrée, lever une erreur (erreur de syntaxe, par exemple `123 + 25 36`

(e) Renvoyer le contenu de la seule case de `output`.