Petit guide pour générer vos exercices de révision

1 lexeur

Ci-dessous, une liste (non exhaustives) de types de lexèmes pouvant être utilisés à l'examen classés en trois classes de difficultés.

Choisissez-en quelques-uns (de préférence avec des structures similaires affin de favoriser les conflits) et :

— donnez leur expressions régulières,
— construisez un automate non-déterministe pour chaque types de lexèmes,
— construisez un lexeur reconnaissant tous les types de lexèmes choisis,
— exécutez votre lexeur sur quelques exemples,
— faites des exécutions avec et sans le maximum-munch.

Lexèmes standards

— numéro de téléphone,
— verbe du 1er/2ieme groupe
— les commentaires multilignes de *C* (uniligne et/ou multiligne),
— les commentaires uniligne de *C*,
— nom de variable correcte en *C*,
— les flottants de la forme `12.54`, `-2.42` ou `12.`,
— les entiers,
— strings (sans échappement),
— les adresses emails (version standard),
— les listes de mots entre crochets (par exemple `[]`, `[foo]` ou `[foo,bar]`).
— les codages d'images bitmap 256 couleurs (chaque ligne contient un nombre pair de pixels et chaque pixel contient 3 chiffres en hexadécimal) contenant un pixel noir,
— les adresses postales (avec une précision choisie...),
— numéro d'étudiant ou de sécu,
— liens (html, ftp...),
— caractère UTF8 (ex : ,
— dates (choisir un format),
— les hexadécimaux selon un (ou plusieurs) systèmes de notation au choix : `0xA42` (C), `&hA42` (Basic), `$A42` (pascal), `#A42` (Lisp), `0hA42` (Texas), `X'A42'` (Cobol), `A42h` ou `A42H` (intel), `A42(16)` (math), `&#A42` (XML), `16rA42` (Smaltalk), `16rA42` (Algol), `16#A42` (postcript), `&#A42` (unicodes en html) .

Lexèmes difficiles

— argument valide de la commande scp,
— adresse IP,
— instruction de la mini-JSMachine

— dates valides (européenne, américaine, les deux, avec bisextile, sans),
— heure valide
— les chemins d'accès légaux en linux (et/ou windows),
— les flottants virgule fixe de la forme `12.54`, `.54`, ou `12.`, mais pas `.`, et flottants en notation scientifique scientifique (`12.5e-5`),
— les entiers de OCaml ou JS, avec des underscores dedans (ex : 2_000 ou 06_23_21), mais pas en fin ni en début de nombre, ni deux à la suite,
— strings python : commençant et finissant par `"`, `"""` ou `'`, mais le même symbole pour le début et fin, enfin, il ne peut pas y avoir de retour à la ligne sauf dans les strings entre triple guillemets, (par contre on ne considère pas les symboles échappés)
— strings standards avec symboles échappés (attention `\"` est échappé, mais pas `\\"`)
— les sommes d'entiers et de cases de tableur qui peuvent être discrètes (par exemple `"5+A22+264+OMG2"`) ou continues (par exemple `"SUM(C42,C56)"`) ou un mixe des deux (par exemple `"35+SUM(C42,C56)+B512"`)
— les noms propres qui sont des groupes de mots chacun commençant par une majuscule ou par un 'd' minuscule, par exemple "Jacobé de Naurois" est un seul nom.
— les adresses emails (aussi précisément que possible...).

Lexèmes triviaux (pour compléter les lexeurs)
— un ou plusieurs mots clef (ex : `for` ou `Var`),
— un ou plusieurs symboles simples (ex : `=`, `%` ou `[`),
— un ou plusieurs symboles agrégés (ex : `>=`, ou `++`),
— un ou plusieurs symboles invisibles (ex : , `\t` ou `\n`) typiquement utilisés pour les séparateurs.

2 Parseur

2.1 Grammaire symbolique

Écrivez une grammaire quelconque avec $a, b...$ comme terminaux et $S, T, E, F...$ comme non terminaux. Par exemple

$$S := aESb \mid Fb$$
$$E := EaE \mid F$$
$$F := \epsilon \mid bE$$

Une telle grammaire sera probablement ambigüe, mais ce n'est pas grave : le but ici est de trouver les conflits.
— Établissez le parseur non-déterministe,
— écrivez le parseur LR_0 psedo-déterministe,
— identifiez les conflits,
— essayez de voire si certains conflit correspondraient à des soucis de priorité et/ou d'associativité,

— calculez les firsts et follows,
— écrivez le parseur SLR psedo-déterministe,
— s'il y a toujours des conflits, vous pouvez alors essayer de calculer le parseur LR_1 ou bien chercher une exécution qui exploite ses conflits et montrerait que la grammaire est ambigüe. [1]

2.2 grammaire réelle

Sélectionnez le langage de votre choix (de préférence que vous connaissez un peu) et essayez d'écrire la grammaire d'un de ses fragment, pour ça :
— vous pouvez utiliser vos connaissances, une description en ligne du langage, voir une grammaire plus riche trouvée en ligne,
— ne sélectionnez que 4-8 opérateurs/structures,
— n'hésitez pas à trivialiser certains non terminaux (par exemple un fragment où toute expression est un entier) pour se concentrer sur les autres non-terminaux.

Une fois fait,
— faites comme pour les grammaires symboliques. Normalement vous aurez beaucoup moins de conflit (mis à part les priorités/associativités),
— exécutez votre parseur sur quelques exemples (vous pouvez utiliser l'optimisation linéaire où on se souvient des états visités).

3 Assembleur virtuel

3.1 Compilation

Écrivez un code JS et :
— écrivez-en l'arbre syntaxique,
— choisissez des conditions de typages pour ne pas avoir à écrire tout le code du typage dynamique (sauf si vous voulez vous concentrer sur cet aspect),
— générez du code,
— vous pouvez aussi voire à écrire des optimisations.

3.2 Décompilation

Demandez à l'un de vos camarade un code compilé (à la main sans optimisation ou à l'aide de son projet) et essayer de retrouver le code JS de départ.

3.3 Interprétation

Sur un code compilé (à la main ou à l'aide d'un projet), exécutez la machine étape par étape. À l'examen, on ne vous le demandera pas, mais on peut vous demander une description de la machine après exécution sur un code donné.

1. Ce genre de questions sont hors programme mais peuvent apparaître comme bonus.

3.4 Petit programme assembleur ?

Ça ne tombera pas à l'exam, mais vous devriez être capable d'écrire de petits programmes directement dans l'assembleur.

4 Compréhension global

4.1 Question de cours

Vous devriez être capable de synthétisé en 5-6 lignes ce que vous savez des différents concepts rencontrés dans le cours. Exemples : interprétation, frontend, lexème, GC, ConJmp, machine virtuelle, Clôture, ambiguité, ect...

4.2 QCM de cours

Posez des petites questions rapide à choix multiples sur le cours.

4.3 Analyse d'erreur de compilation/exécution

Dans le style du TP3, le but est de décrire l'approche plus que trouver le bug (sur papier ce n'est pas forcément évident).

5 Lien avec le projet

N'omettez pas de travailler sur le projet : les TPs étant liés au projet, leur contenu peut apparaître à l'examen. Par exemple, on peut vous demander d'écrire l'un des document suivant (dans le langage de votre choix)

- un fichier de génération de lexeur,
- un fichier de génération de parseur,
- un encodage possible d'AST,
- un programme capable de générer un code d'assembleur à partir de n'importe quel AST sur un petit fragment de JS,
- ...