Compilation

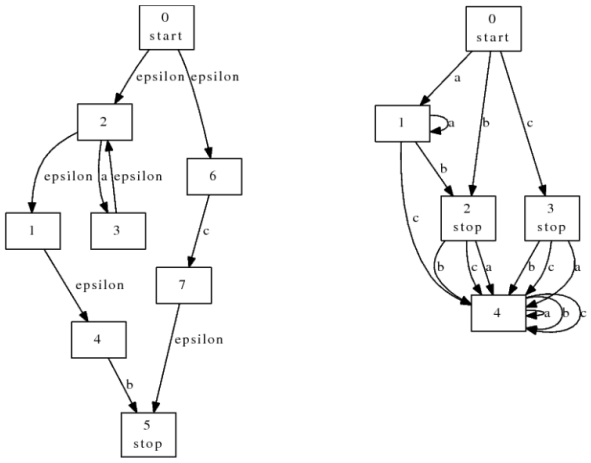
TP 1 : Orthographe

C. Alias & G. Iooss

La première étape d'un compilateur est de diviser le texte du programme en unités lexicales (*lexèmes*). L'écriture d'un analyseur lexical étant fastidieuse, il n'est pas question de tout faire à la main... Nous allons donc construire un compilateur (de compilateur) qui produit automatiquement le code C d'un analyseur lexical étant donnée une description lexicale.

Exercice 0. *Echauffement*

On considère l'expression régulière $a^*b \mid c$ sur l'alphabet $\Sigma = \{a, b, c\}$. La construction de Thompson et sa version déterminisée sont données dans la figure suivante:



Questions.

- **Vérifiez** que la construction de Thompson et sa version déterminisée proviennent bien de l'algorithme donné en cours.
- En appliquant l'algorithme du cours, **minimisez** l'automate déterministe. Notez que l'automate déterministe est aussi complet. Cette hypothèse est essentielle pour la minimisation. **Vérifiez-le** en minimisant l'automate déterministe privé de l'état 4.

Exercice 1. *Expressions régulières*

Recopiez le fichier `/home/calias/M1-Compilation/tp1/src0_automaton.tgz` dans votre répertoire et décompactez le avec `tar xvfz src0_automaton.tgz`. Notre générateur d'analyseur lexical est constitué des fichiers suivants:

- **Regexp.***: Expressions régulières
- **Automaton.***: Automates: construction de Thompson, déterminisation, minimisation.
- **Lexer.***: Génération d'analyseur lexical (fourni plus tard)

Manip.

- **Ouvrez le fichier** `Regexp.h`. Une expression régulière est un type somme qui se décrit naturellement avec des champs booléens (`is_epsilon`, `is_letter`, etc). Noter également les *pretty-constructors*. Allez voir l'implémentation dans `Regexp.cc`.
- **Créez le fichier** `main.cc`. Ecrivez le code pour **construire et afficher** $a^*b \mid c$. Pour compiler, retirez Automaton.o de la variable OBJS du Makefile.

Exercice 2. *Automates*

La première étape est de construire un automate déterministe, complet et minimal qui représente la description lexicale. Ces étapes sont abordées point par point dans cet exercice.

Manip.

- **Ouvrez le fichier** `Automaton.h`. Un automate comporte un ensemble d'états (ligne 37), une relation de transition (ligne 38), un état initial et un état final. Deux classes auxiliaires sont nécessaires: `State` (lignes 15–19) dont chaque instance représente un état différent, et `Label` (lignes 21–30) qui représente soit une lettre, soit ε.
- **Ouvrez le fichier** `Automaton.cc`. Complétez la construction de Thompson pour gérer l'union (ligne 83) en reproduisant le schéma donné en cours.
- **Analysez** les méthodes de déterminisation (`determinize()`, lignes 208–301) et de minimisation (`minimize()`, lignes 340–494), qui implémentent directement les algorithmes vus en cours.
- **Ouvrez le fichier** `main.cc`. Ajoutez la construction de Thompson, la déterminisation et la minimisation pour $a^*b \mid c$. Utiliser la méthode `print_dot()` pour **afficher l'automate graphiquement**. Pour cela, placez le script généré dans un fichier `test.dot`, et générez l'automate avec la commande `dot -Tps test.dot > test.ps`.
- Produire et afficher l'automate pour $\big($`while\1 | b\2 | ␣\3 | ([w|h|i|l|e|b])*\4`$\big)*$ où les caractères `\1`, `\2`, `\3` et `\4` sont des *marqueurs*. Que reste-t-il à faire pour construire un analyseur lexical?

Exercice 3. *Analyseurs lexicaux*

Une fois l'automate produit, il faut (i) produire des tables (table des transition et table des lexèmes reconnus pour chaque état), et (ii) générer le code C de l'analyseur.

A) Production des tables

Dans le fichier `Automaton.cc`, la méthode `transition_table` retourne trois données:

- La **table de transition** de l'automate, avec en ligne les états, et en colonne les lettres de l'alphabet.
- Un **mapping lettre → colonne** pour s'y retrouver dans la table.
- Un **mapping état → lexème reconnu**, qui retourne le code du lexème reconnu sur l'état. Le code du lexème est le code ASCII du marqueur associé. Dans l'exercice 2, le code de `while` est 1, celui de `b` est 2, etc. Si aucun lexème n'est reconnu sur l'état, on lui associe le code 0.

Manip.

- **Ouvrez le fichier** `main.cc`. Appliquez `get_transition` à l'automate minimisé. Affichez les données calculées (la table et les deux mappings). Comment sont traités les états ambigus (sur lesquels on peut reconnaitre plusieurs lexèmes)?
- **Sur feuille**, écrivez la boucle principale de l'analyseur lexical.

B) Production du code de l'analyseur lexical

Recopiez le fichier `/home/calias/M1-Compilation/tp1/src1_lexer.tgz` dans votre répertoire et décompactez le avec `tar xvfz src1_lexer.tgz`. Le fichier `Lexer.cc` implémente la génération d'analyseur lexical. Plus précisement:

- Le **constructeur** prend en argument un alphabet (sans les marqueurs) et une description lexicale. La description lexicale est implémentée avec un vecteur d'expressions régulières `Regexp`, le *rang* d'une expression régulière indiquant *sa priorité*. Le constructeur produit un automate minimal pour la description lexicale passée en argument. Les marqueurs utilisés sont les caractères de code ASCII 1, 2, etc.

- La méthode **print_code** génére le code C de l'analyseur lexical. Elle fait d'abord appel à `transition_table`, puis produit la table (tableau statique), et les fonctions nécessaires. Le code produit suppose l'existence des fonctions suivantes:
  - **reset_input_flow()** initialise le flot d'entrée (ouverture d'un fichier, par exemple).
  - **read_next()** lit le caractère suivant sur le flot d'entrée.
  - **unread()** revient un caractère en arrière dans le flot d'entrée.
  - **Eof** est le caractère de fin du flot (0 pour un flot chaine de caractère, EOF pour un fichier).
  - **accept(token,string)** est invoqué par l'analyseur chaque fois qu'un lexème ("token" en anglais) est reconnu. Typiquement, l'implémentation d'accept pourra l'afficher.
  - **error(car_number)** est invoqué par l'analyseur dès qu'une erreur est rencontrée (caractère non valide, car pas dans l'alphabet d'entrée ; ou bien token non reconnu).

Manip.

- **Créez le fichier main.cc**. Ajoutez le code qui produit l'analyseur lexical pour la description lexicale vue dans l'exercice 2. Redirigez la sortie de votre exécutable vers le fichier `lex.c`. Notez qu'un fichier `automaton.dot` est également produit, avec l'automate minimal.
- **Ouvrez le fichier lex.c** et comparez avec votre algorithme.
- **Créez un fichier main.c** (en C, donc) et implémentez les fonctions manquantes pour l'analyseur lexical. Pour simplifier, on lira les caractères dans une chaine (char*). On ajoutera un entier qui indique la position courante dans la chaine. Testez.
- Poussez l'analyseur dans ses retranchements... Qu'est ce qui prend le plus de temps? Que pourrait-on optimiser ?

Exercice 4. *Flex*

`flex` est un générateur d'analyseur lexical très efficace (en plus d'être libre) et largement utilisé dans la communauté des développeurs de compilateurs.

A) Fonctions de base

Recopiez le fichier `/home/calias/M1-Compilation/tp1/src0_flex.tgz` dans votre répertoire et décompactez le avec `tar xvfz src0_flex.tgz`.

Le fichier `lexer.l` contient la description lexicale de notre compilateur C. Si vous avez le temps, vous pourriez vous amuser à la faire passer dans l'analyseur de l'exercice 3 (je suis preneur). Un fichier de spécification flex est constitué de trois parties séparées par le lexème `%%`.

- La première partie contient du **code C/C++ à copier/coller** directement au début du code de l'analyseur (entre {% et %}). Typiquement, le #include "parser.h" (ligne 32) qui importe les symboles des lexèmes à reconnaître.
- La deuxième partie contient la **description lexicale** sous la forme d'une suite de couples expression régulière, action à effectuer quand on la rencontre. Typiquement, transmettre le lexème reconnu à l'analyseur syntaxique *via* un return.
- La troisième partie contient du **code C/C++ à copier/coller** directement à la fin du code de l'analyseur. Typiquement, une fonction `main()` quand on veut juste tester l'analyseur lexical, comme c'est le cas dans cet exercice.

L'analyseur produit par `flex` exporte une fonction `yylex()` qui retourne le **premier lexème reconnu** dans le fichier `FILE* yyin` (qu'il faut donc ouvrir). L'invocation suivante de `yylex()` retourne le lexème suivant, etc. Lorsqu'il n'y a plus de lexèmes à lire, `yylex()` retourne 0.

Manip.

- **Ouvrez le fichier lexer.l**. Ajoutez une fonction main() qui ouvre le fichier passé en argument (dans argv[1]) et qui affiche tous les lexèmes reconnus. Testez sur le fichier `bd.c`.

- Modifiez la description lexicale pour **afficher le lexème** quand c'est un identificateur (TK_ID). On utilisera la variable `char* yytext`. Testez.

B) Start conditions

`flex` permet de construire plusieurs automates et de passer d'un automate à un autre. Chaque automate est appelé *start condition*.

- Un automate (start condition) se **déclare** en première partie avec `%x nom_de_l_automate`.
- Ensuite, la **description lexicale** de l'automate se déclare en préfixant chaque expression régulière concernée par `<nom_de_l_automate>`.
- On ordonne à l'analyseur de **changer d'automate** en appelant la macro `BEGIN(nom_de_l_automate)`.
- L'automate par défaut s'appelle `INITIAL`. Donc, **on revient à l'automate par défaut** par l'appel `BEGIN(INITIAL)`.

Manip.

- Ajoutez la gestion des commentaires // et /* ... */

Exercice 5. *Bonus: compaction de la table de transition*

Les tables de transition produites peuvent être énormes. Par souci d'économie de mémoire, il peut être utile de les compacter. **Inventez un algorithme de compaction**. On pourra s'aider des tables produites par notre générateur d'analyseurs lexicaux pour tester sa performance.
Remarque: souvent, les expressions régulières décrivent des lexèmes qui n'ont que peu de lettres en commun. Elles n'utilisent donc pas (ou presque pas) les mêmes colonnes.