Projet de Compilation: Implémentation d'un générateur d'analyseurs lexicaux

ÉNSIIE, semestre 3

1 Informations pratiques

Ce projet est à effectuer en binômes. Les binômes constitués enverront un mail à `guillaume.burel@ensiie.fr` avant le 5 décembre pour vérification.

Le code rendu comportera un Makefile, et devra pouvoir être compilé avec la commande `make`. **Tout projet ne compilant pas se verra attribuer un 0** : mieux vaut rendre un code incomplet mais qui compile, qu'un code ne compilant pas. Votre code devra être **abondamment commenté et documenté**. Les bonus ne seront pris en compte que si le reste du projet fonctionne parfaitement, ne les commencez pas avant.

Votre projet sera écrit au choix en OCaml ou en C. Indépendamment, le code produit par votre générateur pourra être de l'OCaml ou du C.

Des fichiers d'entrées pour tester votre code seront disponibles depuis l'adresse : `http://www.ensiie.fr/~guillaume.burel/compilation/` . Vous attacherez un soin particulier à ce que ces exemples fonctionnent.

Votre projet est à déposer **avant le 3 janvier 2012 à 17h** sur le serveur `lunix31` dans le répertoire IIE2-ICO-1. Vous n'oublierez pas d'inclure dans votre dépôt un rapport (PDF) précisant vos choix, les problèmes techniques qui se posent et les solutions trouvées.

2 Sujet

Le but de ce projet est d'implémenter un générateur d'analyseur lexicaux, c'est-à-dire de réécrire `lex`. Comme indiqué un cours, `lex` peut être vu comme un compilateur, qui prend en entrée des fichiers au format lex, et qui produit du code C. On distinguera les phases suivantes :

2.1 Front-end

L'objectif de cette phase est de reconnaître un fichier du format lex et de produire l'arbre de syntaxe abstraire correspondant. On ne considérera en fait qu'un sous-ensemble du langage reconnu par lex. Le format des fichiers reconnu sera le suivant :

```
%{
    entête
%}
%%
    règles
%%
    trailer
```

L'entête contiendra n'importe quels caractères hormis la suite %} et devra être recopié intégralement en début du fichier généré.

Le *trailer* contiendra n'importe quels caractères et devra être recopié tel quel en fin du fichier généré.

Les règles seront chacune de la forme *expression_régulière* {*action*} (avec au moins un espace ou une tabulation entre *expression_régulière* et {*action*}) où

- *action* est composé de n'importe quels caractères hormis } et sera recopié tel que dans le fichier généré à l'endroit où l'action doit être déclenchée.
- *expression_régulière* est une expression formée de la manière suivante :
  Les caractères spéciaux sont \ [ ] ^ - ? . * + | ( ) { }. Un caractère non-spécial reconnaît le caractère en question. Pour reconnaître un caractère spécial, il faut le protèger par \ .
  Les expressions sont construites de la façon suivante :

expr.	signification
c	reconnaît le caractère non-spécial c
\c	reconnaît le caractère (spécial ou non) c
ef	reconnaît l'expression e puis l'expression f
.	reconnaît n'importe quel caractère
$[S]$	reconnaît l'ensemble de caractères S, où S est une concaténation de caractères ou d'intervalles de caractères (par exemple a-z)
$[\hat{}S]$	reconnaît l'ensemble des caractères n'appartenant pas à S
e?	reconnaît optionnellement l'expression e
e+	reconnaît l'expression e une ou plusieurs fois
e*	reconnaît l'expression e zéro, une ou plusieurs fois
$e \mid f$	reconnaît l'expression e ou l'expression f
(e)	reconnaît e (utile pour changer la priorité des opérateurs)

| est le moins prioritaire des opérateurs ; puis vient la concaténation ; puis les autres opérateurs.

Bonus Rajouter la possibilité de définir des alias pour les expressions régulière : entre l'entête et les règles (avant %%), on peut écrire des lignes de la forme

identifiant *expression_régulière*

où *identifiant* commence par une lettre et est composé de lettres, de chiffres et d'*underscores* ; on peut ensuite réutiliser ces alias dans les expressions régulières des règles avec la syntaxe {*identifiant*} .

2.2 Middle end

L'objectif de cette phase est de transformer l'arbre de syntaxe abstraite petit à petit pour se rapprocher du code à produire tout en effectuant des optimisations. On utilise pour cela une suc-

cession de représentations intermédiaires. Dans notre cas précis, les représentations intermédiaires seront les automates non-déterministes avec ϵ-transitions, puis éventuellement non-déterministes sans ϵ-transitions puis déterministes. Sur cette dernière représentation intermédiaire, on effectuera comme optimisation la minimisation de l'automate.

Il pourra être utile, notamment à des fins de débogage, d'implémenter une fonction d'affichage pour chacune de ces représentations intermédiaires.

Remarque : À chaque état final de l'automate on devra associer l'action correspondante dans le fichier d'entrée. On rappelle qu'en cas de conflit, c'est l'action associée à l'expression régulière donnée en premier qui prévaut.

2.3 Back end

L'objectif de cette phase est de produire le code proprement dit. L'automate minimisé obtenu à la fin de la phase précédente sera transformé en du code C ou OCaml (selon le cas). Cette fonction lira l'entrée à analyser dans un *buffer* lexical. Pour utiliser tous le même type, on impose le type suivant pour ce *buffer* :

- en C :

```
struct buffer_b {
  char *content;
  int pos;
};
typedef struct buffer_b *buffer;
```

- en OCaml :

```
type buffer = {
  content : string;
  pos : int
}
```

Les prototypes des fonctions à produire seront donc :

- en C :

```
void* lexer(buffer);
```

- en OCaml :

```
val lexer: buffer -> buffer * t
```

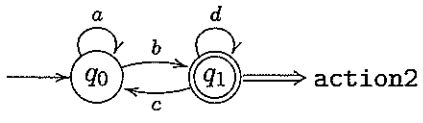
où `t` est le type des actions (qui sera inféré automatiquement par OCaml).

L'argument de la fonction (qui pourra être utilisé dans les actions) aura comme identifiant `lex_buf`.

La fonction `lexer` reconnaîtra le plus grand mot possible à partir de la position indiquée dans le *buffer* lexical, et retournera l'action correspondante. Le *buffer* pointera alors sur la position immédiatement après ce mot : en C, il sera mis à jour sur place ; en OCaml, on retournera le *buffer* ainsi modifié.

Pour produire le code correspondant à l'automate, vous pourrez au choix :

- utiliser une table de transition comme dans le code produit par lex : un tableau indique dans quel état aller en fonction du caractère lu ;
- faire des fonctions mutuellement récursives ; par exemple, l'automate



peut être transformé en OCaml

```
let rec lexer lex_buf = q0 lex_buf

and q0 lex_buf =
  match lex_buf.content.[lex_buf.pos] with
  'a' -> q0 { lex_buf with pos = lex_buf.pos + 1 }
| 'b' -> q1 { lex_buf with pos = lex_buf.pos + 1 }
| _ -> raise Lexing_error

and q1 lex_buf =
  match lex_buf.content.[lex_buf.pos] with
  'c' -> q0 { lex_buf with pos = lex_buf.pos + 1 }
| 'd' -> q1 { lex_buf with pos = lex_buf.pos + 1 }
| _ -> lex_buf, action2
```

Cette deuxième solution n'est pas forcément la meilleure, mais l'efficacité du code produit ne sera pas prise en compte.

Bonus *Bootstrap*er votre générateur d'analyseur lexicaux : écrire l'analyseur lexical utilisé dans votre projet à l'aide de programme produit par votre projet.

3 Résumé des tâches à accomplir obligatoirement

- Écrire un analyseur syntaxique qui construit l'arbre de syntaxe abstraite des fichiers d'entrée.
- Écrire une fonction qui transforme ces arbres de syntaxe abstraite en automate non déterministe avec ϵ-transition.
- Écrire une fonction qui déterminise ces automates.
- Écrire une fonction qui minimise les automates déterministes.
- Écrire une fonction qui produit le code correspondant à un automate déterministe.

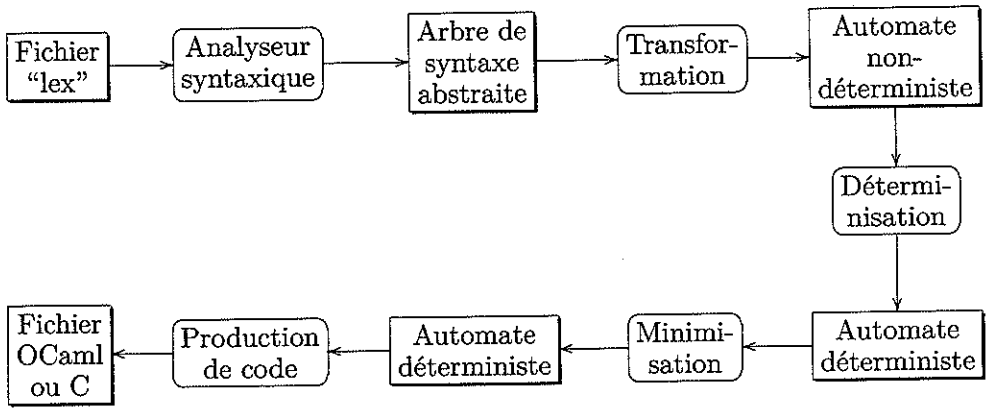


FIGURE 1: Architecture de votre compilateur