Projet 1
ENS Lyon — L3IF
Analyse lexicale et syntaxique

Sébastien Briais

17 septembre 2007

Le but de cette partie est d'écrire un analyseur lexical et un analyseur syntaxique pour le langage `drei`. Cette partie est à réaliser individuellement. La construction de l'arbre de syntaxe abstraite sera la première étape à réaliser en binôme.

À rendre pour le 5 octobre minuit au plus tard.

1 Consignes générales

Votre compilateur devra pouvoir être invoqué à l'aide de la commande `dreic` dans un terminal. Il prendra en argument un paramètre indiquant la phase du compilateur testée suivi du nom de fichier `drei` à "compiler". Un exemple typique d'utilisation est :

```
./dreic -scanner Factorial.drei
```

Les noms des phases sont indiqués ci-après :

Nom de la phase	Nom de l'option
Analyse lexicale	`-scanner`
Analyse syntaxique	`-parser`
Construction de l'arbre	`-printer`
Analyse des types	`-analyser`
Génération de code	`-generator`

Vous pouvez tout à fait nommer votre compilateur `drei` comme bon vous semble puis ensuite écrire un script shell nommé `dreic` pour réaliser l'invocation de votre compilateur en suivant ce standard.

Pour ceux qui ont choisi `ocaml` comme langage d'implémentation, le module `Arg` de la bibliothèque standard pourra vous servir.

2 Analyse lexicale

Cette phase consiste en l'écriture d'un analyseur lexical pour le langage `drei`. Elle transforme le flux de caractères donné en entrée en un flux de lexèmes.

Concrètement, le test de cette phase consiste à écrire sur la sortie standard une représentation textuelle de la liste des lexèmes correspondant au fichier texte fourni en entrée. En cas d'erreur (caractère invalide, chaine non terminée, ...), votre programme pourra s'arrêter en signalant la cause et renvoyer un code d'erreur non nul (fonction `exit` en `ocaml`). Sinon, le code renvoyé sera 0.

À noter que pour pouvoir réaliser les phases futures, il sera utile d'avoir accès à la position (ligne, colonne) du token dans le texte source fourni.

La première chose à faire est de récupérer les deux fichiers décrivant la syntaxe concrète de `drei` :

```
wget http://perso.ens-lyon.fr/sebastien.briais/Projet2007/lexical-grammar.txt
wget http://perso.ens-lyon.fr/sebastien.briais/Projet2007/syntactic-grammar.txt
```

Vous pourrez remarquer que la micro-syntaxe de `drei` est incomplète. Votre première tâche est donc de la compléter en identifiant tous les tokens nécessaires. Il s'agit essentiellement de repérer les symboles terminaux de la macro-syntaxe de `drei`.

2.1 Indications

Voici quelques indications pour ceux qui ont choisi `ocaml` comme langage d'implémentation. Vous êtes bien entendu libre de vous en inspirer ou de les ignorer totalement.

2.1.1 Tokens

Vous allez devoir définir un type token.

```
type token =
 | EOF (* fin de flux *)
 | IDENT of string
 | NUMBER of Int32.t
 | STRING of string
 | LPAREN | RPAREN
 | ...
 | INT
 | CLASS | EXTENDS | ...
 | ...
```

Je vous conseille de définir ce type dans un module `Tokens` où vous regrouperez toutes les fonctions concernant les tokens.

Pour distinguer entre mot-clé et identificateur, une technique est d'avoir une table de hachage contenant la correspondance pour chacun des mots clé entre chaine de caractère et token associé.

Vous pourrez ensuite avoir une fonction `token_of_keyword: string -> token` qui renvoie le token associé à un mot clé ou une exception s'il ne s'agit pas d'un mot clé.

Enfin pour afficher un token, une fonction `string_of_token: token -> string` définira la représentation textuelle des tokens.

2.1.2 CharReader

Pour faciliter l'écriture de l'analyseur lexical, je vous conseille d'implémenter l'interface vue en cours pour représenter le flux de caractères fourni en entrée. Un module `CharReader` implémentant l'interface suivante pourra être utile :

```
(* le type des flux de caractères *)
type t

(* consomme le caractère courant *)
val next_char: t -> unit

(* donne le caractère courant ou None *)
val current_char : t -> char option

(* donne la position du caractère (ligne, colonne) *)
val get_position : t -> int * int

(* crée un flux à partir d'une chaine de caractères *)
val charReader_of_string : string -> t

(* crée un flux à partir d'un canal d'entrée *)
val charReader_of_in_channel : in_channel -> t
```

2.1.3 Scanner

Enfin, il vous restera à définir le module `Scanner` implémentant l'analyseur lexical. Une interface possible est la suivante :

```
(* le type des flux de lexèmes *)
type t

(* le token courant *)
val current_token : t -> Tokens.token

(* consomme le token courant *)
val next_token : t -> unit

(* donne la position du token courant *)
val get_position : t -> int * int

(* crée un flux de lexèmes *)
val new_scanner : CharReader.t -> t

(* la fonction de test *)
val main : in_channel -> unit
```

2.1.4 Report

Pour la gestion des erreurs, il pourra être utile de définir certaines fonctions dans un module `Report` qui se chargera d'afficher les messages d'erreur ou

d'avertissement sur la sortie d'erreur standard et tiendra le compte du nombre d'avertissements signalés. Une interface possible est

```
(* signale une erreur et quitte en renvoyant un code non nul *)
val error : int * int -> string -> 'a

(* idem en indiquant la position dans le fichier source *)
val error_at_pos : int * int -> string -> 'a

(* signale un avertissement *)
val fail : int * int -> string -> unit

(* idem en indiquant la position dans le fichier source *)
val fail_at_pos : int * int -> string -> unit
```

3 Analyse syntaxique

Une fois l'analyseur lexical implémenté, vous allez pouvoir passer à l'analyseur syntaxique.

Il faudra peut être légérement transformer la grammaire fourni pour que vous puissiez implémenter un analyseur syntaxique par descente récursive (éliminer les récursivités à gauche, factoriser à gauche là où c'est nécessaire).

La phase de test consiste simplement à renvoyer le code 0 si le texte fourni en entrée est un programme `drei` syntaxiquement correct et sinon, vous pourrez interrompre l'analyse syntaxique à la première erreur que vous signalerez par un message d'erreur et un code de retour non nul.

3.1 Indications

Pour ceux qui ont choisi `ocaml` comme langage d'implémentation, une interface minimale à implémenter dans un module `Parser` serait :

```
(* analyse un programme *)
val parse_program : Scanner.t -> unit

(* fonction de test *)
val main : in_channel -> unit
```

Sachant que le résultat de `parse_program` sera plus tard un arbre de syntaxe abstraite. L'interface sera donc après l'écriture de la deuxième phase :

```
(* analyse un programme *)
val parse_program : Scanner.t -> Ast.t

(* fonction de test *)
val main : in_channel -> unit
```