Projet de compilation : mini-langage orienté objets
Partie 1: Analyse syntaxique

Dans cette première partie du projet, on s'intéresse à l'étape d'analyse syntaxique d'un langage à objets assez simple proche du langage décrit en cours.

Nous donnons dans un premier temps la spécification des unités lexicales ainsi que de la syntaxe concrète (c'est-à-dire la grammaire permettant d'analyser un programme source). Nous donnons ensuite la spécification de la syntaxe abstraite (c'est-à-dire l'ensemble des types OCaml permettant de décrire les arbres de syntaxe abstraite). Nous terminons ce document par une liste des consignes pour la première partie du projet.

1 Syntaxe concrète

1.1 Analyse lexicale

Caractères ignorés. Les espacements, les tabulations et les retours à la ligne sont ignorés, c'est-à-dire qu'ils ne produisent aucune entité lexicale pour l'analyse grammaticale.

Identifiants. Toute chaîne de caractères commençant par une lettre et ne contenant que des lettres, des chiffres ou le caractère `_` est un identifiant valide.

Il existe deux classes d'identifiants. Les identifiants commençant par une lettre majuscule, notés Id, seront utilisés pour les noms de classes. Les identifiants commençant par une lettre minuscule, notés id seront utilisés pour les noms de variables, d'attributs et de méthodes.

Mots-clés. Les mots-clés du langage sont:

```
class extends var method new as print fail
```

Ils seront notés en gras dans la grammaire concrète.

Chaîne de caractères. Une chaîne de caractères est une séquence de caractères quelconques entre guillemets doubles, par exemple: `"ceci est une chaîne ~#@^{% 123"`. Les caractères spéciaux `\n` et `\"` sont autorisés dans les chaînes de caractères, et correspondent respectivement au caractère "guillemets" et au caractère de fin de ligne. Dans la suite, nous noterons str les chaînes de caractères.

Commentaires. Comme en OCaml, les commentaires commencent par `(*` et se terminent par `*)`. Ils peuvent faire plusieurs lignes. Les commentaires imbriqués sont autorisés. Par exemple le commentaire `(* commentaires (* je commente toujours mon code *) *)` est valide, mais `(* mauvais *) commentaire *)` ne l'est pas.

1.2 Analyse grammaticale

Nous utilisons une grammaire au format habituel et des priorités données en langage naturel pour décrire la syntaxe concrète du langage. La grammaire décrivant cette syntaxe est donnée ci-dessous. Elle correspond essentiellement à la grammaire du langage à objets présenté en cours.

program	::=	*class_decl** *expr*
class_decl	::=	**class** Id **extends** Id '=' '(' *var_decl** *method_decl** ')'
var_decl	::=	**var** id ':' Id
method_decl	::=	**method** id '(' *arg_list* ')' ':' Id '=' *expr*
arg_list	::=	id ':' Id ',' ... ',' id ':' Id
expr	::=	id
	|	**new** Id '(' *expr* ',' ... ',' *expr* ')'
	|	*expr* '.' id
	|	*expr* '.' id '(' *expr* ',' ... ',' *expr* ')'
	|	*expr* **as** Id
	|	*inst* ';' *expr*
	|	'(' *expr* ')'
	|	**fail**
inst	::=	*expr* '.' id ':=' *expr*
	|	*inst* ';' *inst*
	|	**print** str

Modifications.

La syntaxe du langage a été légèrement modifiée par rapport à celle du cours. En particulier, dans les règles

$$\textit{var\_decl} ::= \textbf{var}\ \text{id}\ \text{':'}\ \text{Id}$$
$$\text{et} \qquad \textit{method\_decl} ::= \textbf{method}\ \text{id}\ \text{'('}\ \textit{arg\_list}\ \text{')'}\ \text{':'}\ \text{Id}\ \text{'='}\ \textit{expr}$$

on a ajouté respectivement les mots-clés **var** et **method**, et dans la règle

$$\textit{class\_decl} ::= \textbf{class}\ \text{Id}\ \textbf{extends}\ \text{Id}\ \text{'='}\ \text{'('}\ \textit{var\_decl}^\star\ \textit{method\_decl}^\star\ \text{')'}$$

on a ajouté des parenthèses avant et après les déclarations d'attributs et de méthodes.

2 Syntaxe abstraite

La syntaxe abstraite définit un langage d'arbre qui correspond aux arbres de dérivation de la syntaxe concrète. La Figure 1 donne une définition de ce langage à l'aide d'un ensemble de types OCaml. Ce code vous est fourni (fichier `ast.ml`) et ne nécessite en principe aucune modification.

```
type id = string and class_name = string and method_name = string

(* Une variable et son type *)
type var = { var_name: id; var_type: class_name; }

(* Une expression *)
type expr =
  (* Variable *)
  | E_Var of id
  (* Nouvel objet *)
  | E_New of class_name * expr list
  (* Invocation d'attribut *)
  | E_Attr of expr * id
  (* Appel a une methode *)
  | E_Call of expr * method_name * expr list
  (* Downcasting *)
  | E_Cast of expr * class_name
  (* Instruction suivie d'une expression *)
  | E_Sequ of instr * expr
  (* Interruption du programme *)
  | E_Fail

(* Une instruction *)
and instr =
  (* Affectation d'attribut *)
  | I_Affect of expr * id * expr
  (* Suite d'instructions *)
  | I_Sequ of instr * instr
  (* Affichage *)
  | I_Print of string

(* Une methode *)
type method_ = {
  method_name : method_name;
  method_args: var list;
  method_return: class_name;
  method_body: expr;
}

(* Une classe *)
type class_ = {
  class_name: class_name;
  class_parent: class_name;
  class_attributes: var list;
  class_methods: method_ list;
}

type program = { program_classes: class_ list; program_body: expr; }
```

FIG. 1 – *Syntaxe abstraite du langage*

3 Exemple de programme

Voici un exemple de programme permettant de manipuler des booléens et des références vers des objets. Son code vous est fourni avec le reste des fichiers du projet (fichier `bool-ref.obj`).

```
class Bool extends Object = (
 method ite(x : Object, y : Object) : Object = new Object() )

class True extends Bool = (
 method ite(x : Object, y : Object) : Object = x )

class False extends Bool = (
 method ite(x : Object, y : Object) : Object = y )

class Ref extends Object = (
 var val : Object
 method read () : Object = this.val
 method write (x : Object) : Ref = this.val := x; this )

(new True()).ite(new True(), new False ()) as Bool
```

4 Consignes

1. Recopiez sur votre compte le répertoire `/info/ens/meyera/Public/Compil/projet`. Dans celui-ci, vous trouverez un squelette du code de la partie 1, un répertoire contenant l'exemple de code présenté au paragraphe 3 ainsi que ce document.
2. Complétez les fichiers `lexer.mll` et `parser.mly` en suivant la définition des unités lexicales, de la grammaire du langage et de la syntaxe abstraite présentées ci-dessus afin de créer un analyseur syntaxique pour le langage. En principe, il n'est pas nécessaire de modifier les autres fichiers fournis. Nous vous fournissons un afficheur afin de tester votre code (fichier `print.ml`).
3. Un certain nombre de conflits apparaissent lors de la compilation du fichier `parser.mly` avec `menhir`. Nous choisissons délibérément de ne pas les détailler et de ne pas imposer une solution. Résolvez ces conflits de la façon qui vous paraîtra la plus appropriée, en prenant soin de **garder une trace** de votre solution pour pouvoir l'expliquer ensuite.
4. Il est indiqué au paragraphe 1 un ensemble de modifications apportées à la syntaxe du langage (mots-clés **var** et **method** et parenthèses délimitant une définition de classes). Supprimez temporairement ces modifications et expliquez **en termes simples** les nouveaux conflits.
5. En vous inspirant de l'exemple de programme proposé, écrivez un programme permettant de manipuler des entiers codés en unaire (exercice 7.5 du cours).

Ce travail donnera lieu à une **pré-soutenance** à une date qui vous sera indiquée ultérieurement.