Projet du cours « Introduction à la compilation »
Partie 1 : Analyse lexicale et syntaxique

version numéro f03fcf6cc5c64e988daf1185aa566d4b08ea4e4e

Le sujet du projet du cours d'introduction à la compilation a pour objectif l'implémentation d'un langage de programmation. Cette année, le langage est un langage fonctionnel muni de types sommes, produits et récursifs. Le projet est constitué de deux parties. La première est l'implémentation de l'analyse lexicale et syntaxique, la seconde est l'implémentation d'un interprète et d'un vérificateur de types.

Dans la suite, on présente la grammaire formelle du langage. On vous fournira une série de fichiers qui constituent un squelette du compilateur pour ce langage. Vous devez compléter ce squelette.

1 Spécification de la grammaire

1.1 Notations extra-lexicales

Les espaces, les tabulations et les sauts de ligne jouent le rôle de séparateurs. Leur nombre entre les différents symboles terminaux peut donc être arbitraire.

Les commentaires sont notés comme en OBJECTIVE CAML, c'est-à-dire entourés des deux symboles « `(*` » et « `*)` ». Par ailleurs, ils peuvent être imbriqués.

On peut aussi introduire un commentaire à l'aide du symbole « `**` » : tout ce qui le suit jusqu'à la fin de la ligne est interprété comme un commentaire.

1.2 Symboles

Symboles terminaux Les terminaux sont répartis en trois catégories : les mots-clés, les identificateurs et la ponctuation.

Les mots-clés sont les noms réservés aux constructions du langage. Ils seront écrits avec des caractères de machine à écrire (comme par exemple les mots-clés `if` et `while`).

Les identificateurs sont constitués des identificateurs de variables, de constructeurs de données et de types ainsi que des littéraux, comprenant les constantes entières, les caractères et les chaînes de caractères. Ils seront écrits dans une police sans-serif (comme par exemple type_id ou int). La classification des identificateurs est définie par les expressions rationnelles suivantes :

var_id	≡ [a-z] [A-Z a-z 0-9 _]*	*Identificateur de variables*
constr_id	≡ [A-Z _] [A-Z a-z 0-9 _]*	*Identificateur de constructeurs de données*
type_id	≡ [a-z] [A-Z a-z 0-9 _]*	*Identificateur de types*
int	≡ [0-9]$^+$ | 0x[0-9 a-f A-F]$^+$ | 0b[0-1]$^+$	*Littéraux entiers*
char	≡ 'atom'	*Littéraux caractères*
atom	≡ \000 | ... | \255 | \0x[0-9 a-f A-F]^2 | [printable] | \\ | \' | \n | \t | \b | \r	
string	≡ " atom* "	*Littéraux chaîne de caractères*

Autrement dit, les identificateurs de variables et de types commencent par une lettre minuscule et peuvent comporter ensuite des majuscules, des minuscules, des chiffres et le caractère souligné `_`. Les identificateurs de constructeurs de données peuvent comporter les mêmes caractères, mais doivent commencer par une majuscule ou par un caractère `_`.

Les constantes entières sont constituées de chiffres en notation décimale, en notation hexadécimale ou en notation binaire.

Les constantes de caractères sont décrites entre guillemets simples (ce qui signifie en particulier que les guillemets simples doivent être échappés dans les constantes de caractères). On y trouve en particulier les symboles ASCII affichables. Par ailleurs, sont des caractères valides : les séquences d'échappement usuelles, ainsi que les séquences d'échappement de trois chiffres décrivant le code ASCII du caractère en notation décimale ou encore les séquences d'échappement de deux chiffres décrivant le code ASCII en notation hexadécimale.

Les constantes de chaîne de caractères sont formées d'une séquence de caractères. Cette séquence est entourée de guillemets (ce qui signifie en particulier que les guillemets doivent être échappés dans les chaînes).

Les symboles seront notés avec la police "`machine à écrire`" (comme par exemple « `(` » ou « `=` »).

Symboles non-terminaux Les symboles non-terminaux seront notés à l'aide d'une police légèrement inclinée (comme par exemple *expr*).

Une séquence entre crochets est optionnelle (comme par exemple « *[* `ref` *]* »). Attention à ne pas confondre ces crochets avec les symboles terminaux de ponctuation notés `[` et `]`. Une séquence entre accolades se répète zéro fois ou plus, (comme par exemple « `(` *arg* *{* `,` *arg* *}* `)` »).

2 Grammaire en format BNF

La grammaire du langage est spécifiée à l'aide du format BNF. Les priorités et associativités des opérateurs doivent être choisies de façon à mimer celles du langage O'Caml. Le jeu de tests vous aidera à les déterminer correctement.

Programme Un programme est constitué d'un ensemble de définitions de types, de valeurs et de fonctions.

p ::= *{ definition }*
definition ::= `type type_id` *[* `<` `type_id` *{* `,` `type_id` *}* `>` *]* `=` *type*
| *vdefinition*
vdefinition ::= `val` *binding* `=` *expression*
| `def var_id` *{* `(` *binding* `)` *}* `:` *type* `=` *expression* *{* `with var_id` *{* `(` *binding* `)` *}* `:` *type* `=` *expression* *}*
binding ::= *arg_id* `:` *type* | *arg_id*
arg_id ::= `var_id` | `_`

Types de données La syntaxe des types est donnée par la grammaire suivante :

type ::= `type_id` *[* `<` *type* *{* `,` *type* *}* `>` *]*
| *type* `->` *type*
| `{ }`
| `{` `constr_id` *[* *type* *]* *{* `+` `constr_id` *[* *type* *]* *}* `}`
| `{` `constr_id` *[* *type* *]* *{* `*` `constr_id` *[* *type* *]* *}* `}`
| `rec type_id is` *type*
| `(` *type* `)`

Expression La syntaxe des expressions du langage est donnée par la grammaire suivante.

expr ::=	`int`	*Entier*
	`char`	*Caractère*
	`string`	*Tableau de caractères*
	`var_id`	*Variable*
	`constr_id` *[* `at` *type* *]* *[* `[` *expr* `]` *]*	*Construction d'une somme*
	[`at` *type* *]* `{` `constr_id` *[* `<-` *expr* *]* *{* `,` `constr_id` *[* `<-` *expr* *]* *}* `}`	*Construction d'un produit*
	`(` *expr* `)`	*Parenthésage*
	`(` *expr* `:` *type* `)`	*Annotation de type*
	expr `;` *expr*	*Séquencement*
	vdefinition `in` *expr*	*Définition locale préfixe*
	expr `where` *vdefinition* `end`	*Définition locale postfixe*
	expr expr	*Application*
	expr `.` *expr*	*Application postfixe*
	expr op expr	*Opérations binaires*
	unop expr	*Opérations unaires*
	`case` *[* `at` *type* *]* `{` *[* `|` *]* *branch* *{* `|` *branch* *}* `}`	*Analyse de motifs*
	`if` *expr* `then` *expr* *[* `else` *expr* *]*	*Expression conditionnelle*
	`fun` *{* `(`*binding*`)` *}* *[* `:` *type* *]* `=>` *expression*	*Fonction*
	`do {` *expr* `}`	*Bloc*
op ::=	`+` | `-` | `*` | `/` | `%` | `=` | `:=` | `&&` | `||` | `<=` | `>=` | `<` | `>` | `!=`	*Opérateurs binaires*
unop ::=	`-` | `~`	*Opérateurs unaires*
branch ::=	*pattern* `=>` *expr*	*Cas d'analyse*

Motifs Les motifs (*patterns* en anglais), utilisés par l'instruction d'analyse de motifs, ont la syntaxe suivante :

```
pattern ::=
          | constr_id [ at type ] [ [ pattern ] ]                                  Somme
          | [ at type ] { constr_id [ -> pattern ] { , constr_id [ -> pattern ] } }  Produit
          | pattern or pattern                                                     Disjonction
          | pattern and pattern                                                    Conjonction
          | not pattern                                                            Négation
          | ( pattern )                                                            Parenthèsage
          | 0                                                                      Motif vide
          | var_id                                                                 Motifs universels
          | _
```

3 Code fourni

Un squelette de code vous est fourni, il est disponible sur la page web du cours. Il contient des `Makefile`s ainsi que des modules O'Caml à compléter. Voici une description rapide de ces modules (les modules à compléter sont précédés d'une étoile) :

- `Clap` : analyse les arguments fournis en ligne de commande et lance les processus correspondants.
- `AST` : définit le type de l'arbre de syntaxe abstraite.
- ⋆ `Lexer` : implémente l'analyse lexicale à l'aide de l'outil `ocamllex`.
- ⋆ `Parser` : implémente l'analyse syntaxique à l'aide de l'outil `menhir`.
- `Sugar` : définit les sucres syntaxiques du langage.
- `Operator` : définit les opérateurs binaires et unaires du langage.
- `Printer` : implémente un afficheur d'arbre de syntaxe.
- `Position` : définit des types et des opérations pour décrire un emplacement dans un fichier.
- `Error` : définit des opérations pour arrêter le programme en cas d'erreur et afficher un message informatif.

La commande `make` produit un exécutable appelé `clap`. On doit pouvoir l'appeler avec un nom de fichier en argument. En cas de réussite (de l'analyse syntaxique), le code de retour de ce programme doit être 0. Dans le cas d'un échec, le code de retour doit être 1.

Par ailleurs, la commande `make check` lance une batterie de tests sur votre programme. Peu de tests sont fournis, vous devez en écrire vous-même ! (Il suffit de rajouter de nouveaux fichiers dans les répertoires idoïnes, ils seront automatiquement pris en compte.)

4 Travail à effectuer

La première partie du projet est l'écriture de l'analyseur lexical et de l'analyseur syntaxique spécifiés par la grammaire précédente. Pour cela, vous devez compléter les parties du code fourni de la forme :

```
failwith "Students, this is your job."
```

Avant toute chose, vous devez définir la variable `$(STUDENTS)` dans le `Makefile.local` du code fourni et compléter le fichier `AUTEURS` à la racine du répertoire en suivant le format suivant :

```
# nom1, prenom1, numero-etudiant1, email1
# nom2, prenom2, numero-etudiant2, email2
```

La compilation s'effectue par la commande « `make` ».

Vous pouvez lancer une batterie de tests sur votre projet à l'aide de la commande « `make check` ».

Le projet est à rendre **avant le** :

19 avril 2013 à 23h59

Votre travail, à effectuer par groupe de 2, devra être construit par la commande « `make dist` ». Cette commande produit une archive de la forme `clap-$(STUDENTS)-13.1.tar.gz`. Vous devez le déposer à l'URL `http://www.regis-gianas.org/~clap2013`. Le login est `p7`. Le mot de passe est `compilation` [1]

Votre projet sera traité de façon automatique : un mail de confirmation vous sera envoyé contenant, en particulier, le résultat de votre travail sur notre propre batterie de tests. Il n'est pas interdit de soumettre votre travail plusieurs

1. Ce mot de passe est donné en clair dans ce document car les exigences de sécurité sont très faibles dans notre situation.

fois (le système de soumission sera mis en place à partir du 5 avril 2013). Attention, si vous n'avez pas respecté le format du fichier `AUTEURS` décrit plus haut, la correction automatique échouera, et vous aurez 0.

Pour finir, vous devez vous assurer des points suivants :

- Le projet contenu dans cette archive **doit compiler**.
- Vous devez **être les auteurs** de ce projet.
- Il doit être rendu **à temps**.

Si l'un de ces points n'est pas respecté, la note de 0 vous sera affectée.

5 Log

```
commit f03fcf6cc5c64e988daf1185aa566d4b08ea4e4e
Date:   Thu Apr 11 10:32:22 2013 +0200

    * Précision sur les échappements dans les chaînes et les caractères.
    * Étend la syntaxe des types avec le type vide {}.
    * Rend optionnel les définitions de champs dans les expressions
      de construction de produits.

commit d3841baadfc86a554cb0a46c33718c173d2e3fed
Date:   Fri Apr 5 10:06:53 2013 +0200

    - Spécification :
      * '<' et '>' dans la syntaxe des déclarations de types.
      * Des virgules pour séparer les champs des types produits.
      * La partie '-> pattern' des motifs de projection est optionnelle.

    - Implémentation :
      * Printer: Corrige quelques erreurs.
      * Operator: Nouveau module déclarant les opérateurs unaires et binaires.

commit 9609c840c1206b1f3c9fd1275e389384b1c45e1f
Date:   Tue Mar 19 22:14:10 2013 +0100

    * Version initiale.
```