Projet du cours « Compilation »
Jalon 1 : Analyse lexicale et syntaxique de Hopix

version Tue Oct 9 15:27:50 CEST 2018

1 Spécification de la grammaire

1.1 Notations extra-lexicales

Les commentaires, espaces, les tabulations et les sauts de ligne jouent le rôle de séparateurs. Leur nombre entre les différents symboles terminaux peut donc être arbitraire. Ils sont ignorés par l'analyse lexicale : ce ne sont pas des lexèmes.

Les commentaires sont entourés des deux symboles « `(*` » et « `*)` ». Par ailleurs, ils peuvent être imbriqués.

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, d'étiquettes, 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_con ou int). La classification des identificateurs est définie par les expressions rationnelles suivantes :

alien_infix_id	≡ ‘ [a-z 0-9 + - * / = _ ! ?]$^+$ ‘	*Identificateur d'opérateurs infixes*
alien_prefix_id	≡ ‘ [a-z 0-9 + - * / = _ ! ?]$^+$	*Identificateur d'opérateurs préfixes*
var_id	≡ [a-z] [A-Z a-z 0-9 _]* | alien_prefix_id	*Identificateur de variables préfixe*
all_var_id	≡ var_id | alien_infix_id	*Identificateur de variables de toutes sortes*
constr_id	≡ [‘ A-Z] [A-Z a-z 0-9 _]*	*Identificateur de constructeurs de données*
label_id	≡ [a-z] [A-Z a-z 0-9 _]*	*Identificateur d'étiquettes d'enregistrement*
type_con	≡ [‘ A-Z] [A-Z a-z 0-9 _]*	*Identificateur de constructeurs de type*
type_variable	≡ [a-z] [A-Z a-z 0-9 _]*	*Identificateur de variables de type*
int	≡ -$^?$[0-9]$^+$ | 0x[0-9 a-f A-F]$^+$ | 0b[0-1]$^+$ | 0o[0-7]$^+$	*Littéraux entiers*
char	≡ 'atom'	*Littéraux caractères*
atom	≡ \000 | ... | \255 | \0[x][0-9 a-f A-F]^2 | [printable] | \\ | \' | \n | \t | \b | \r	
string	≡ " (atom | ['']) −{ ", \' } | \")* "	*Littéraux chaîne de caractères*

Autrement dit, les identificateurs de valeur, de variable de type et de champs d'enregistrement 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 et de constructeurs de type peuvent comporter les mêmes caractères, mais doivent commencer par une majuscule ou un guillemet arrière. Par ailleurs, un identificateur d'opérateur est formé de symboles et de caractères alphanumériques. S'il est entouré de deux ‘, il est dit *infixe*. Enfin, notez que le caractère ␣ représente un espace.

Les constantes entières sont constituées de chiffres en notation décimale, en notation hexadécimale, en notation binaire ou en notation octale. Les entiers utilisent une représentation binaire sur 32 bits en complément à deux. Les constantes entières sont donc prises dans $[-2^{31}; 2^{31} - 1]$.

Les constantes de caractères sont écrites entre guillemets simples (ce qui signifie en particulier que les guillemets simples doivent être échappés dans les constantes de caractères). On y trouve tous les symboles ASCII affichables (voir la spécification de ASCII pour plus de détails). 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.

Programme Un programme est constitué d'une séquence de définitions de types et de valeurs.

p ::=	{ *definition* }	*Programme*
definition ::=	**type** type_con [`<` type_variable { `,` type_variable } `>`] [`=` *tdefinition*]	*Définition de type*
|	**extern** all_var_id `:` *type_scheme*	*Valeurs externes*
|	*vdefinition*	*Définition de valeur(s)*
tdefinition ::=	[`|`] constr_id [`(` *type* { `,` *type* } `)`]	*Type somme*
	{ `|` constr_id [`(` *type* { `,` *type* } `)`] }	
|	`{` label_id `:` *type* { `;` label_id `:` *type* } `}`	*Type produit étiqueté*
vdefinition ::=	**val** var_id [`:` *type_scheme*] `=` *expr*	*Valeur simple*
|	**def** *fundef* { **and** *fundef* }	*Fonction(s)*
fundef ::=	all_var_id [`:` *type_scheme*] `(` [var_id { `,` var_id }] `)` `=` *expr*	

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

type ::=	type_con [`<` *type* { `,` *type* } `>`]	
|	*type* { `*` *type* } `->` *type*	
|	type_variable	
|	`(` *type* `)`	
type_scheme ::=	[**forall** type_variable { `,` type_variable } `.`] *type*	

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

expr ::=	int	*Entier positif*
|	char	*Caractère*
|	string	*Chaîne de caractères*
|	all_var_id [`<` [*type* { `,` *type* }] `>`]	*Variable*
|	constr_id [`<` [*type* { `,` *type* }] `>`] [`(` *expr* { `,` *expr* } `)`]	*Construction d'une donnée*
|	`{` label_id `=` *expr* { `;` label_id `=` *expr* } `}` [`<` [*type* { `,` *type* }] `>`]	*Construction d'un enregistrement*
|	*expr* `.` label_id	*Accès à un champ*
|	*expr* `;` *expr*	*Séquencement*
|	*vdefinition* `;` *expr*	*Définition locale*
|	**fun** `(` [var_id { `,` var_id }] `)` `=>` *expr*	*Fonction anonyme*
|	*expr* `(` [*expr* { `,` *expr* }] `)`	*Application*
|	*expr binop expr*	*Application infixe*
|	**case** *expr* `{` *branches* `}`	*Analyse de motifs*
|	**if** *expr* **then** *expr* [**else** *expr*]	*Conditionnelle*
|	**ref** *expr*	*Allocation*
|	*expr* `:=` *expr*	*Affectation*
|	`!` *expr*	*Lecture*
|	**while** *expr* `{` *expr* `}`	*Boucle non bornée*
|	**for** var_id `=` *expr* **to** *expr* [**by** *expr*] `{` *expr* `}`	*Boucle bornée*
|	`(` *expr* `)`	*Parenthésage*
|	`(` *expr* `:` *type* `)`	*Annotation de type*

Voici la grammaire des définitions auxiliaires utilisées par la grammaire des expressions :

binop ::= `+` | `-` | `*` | `/` | `&&` | `||` | `=?` | `<=?` | `>=?` | `<?` | `>?` | alien_infix_id	*Opérateurs binaires*
branches ::= *[* `|` *]* *branch* *{* `|` *branch* *}*	*Liste de cas*
branch ::= *pattern* `=>` *expr*	*Cas d'analyse*

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

pattern ::= var_id	*Motif universel liant*
| `_`	*Motif universel non liant*
| `(` *pattern* `)`	*Parenthésage*
| *pattern* `:` *type*	*Annotation de type*
| int	*Entier*
| char	*Caractère*
| string	*Chaîne de caractères*
| constr_id *[* `<` *[* *type* *{* `,` *type* *}* *]* `>` *]* *[* `(` *pattern* *{* `,` *pattern* *}* `)` *]*	*Valeurs étiquetées*
| `{` label_id `=` *pattern* *{* `;` label_id `=` *pattern* *}* `}` *[* `<` *[* *type* *{* `,` *type* *}* *]* `>` *]*	*Enregistrement*
| *pattern* `|` *pattern*	*Disjonction*
| *pattern* `&` *pattern*	*Conjonction*

Remarques Notez bien que la grammaire spécifiée plus haut est ambiguë ! Vous devez fixer des priorités entre les différentes constructions ainsi que des associativités aux différents opérateurs. *In fine*, c'est la batterie de tests en ligne qui vous permettra de valider vos choix. Cependant, il est fortement conseillé de poser des questions sur la liste de diffusion du cours pour obtenir des informations supplémentaires sur les règles de disambiguation associées à cette grammaire.

3 Code fourni

Un squelette de code vous est fourni, il est disponible sur le dépot GIT du cours.

```
git@moule.informatique.univ-paris-diderot.fr:Yann/compilation-m1-2018.git
```

Vous devez vous connecter sur le Gitlab disponible ici :

```
http://moule.informatique.univ-paris-diderot.fr:8080
```

et vous créer un dépot par branchement (*fork*) du projet `compilation-m1-2018`.

L'arbre de sources contient des `Makefile`s ainsi que des modules O'Caml à compléter.

La commande `make` produit un exécutable appelé `flap`. 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.

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.

Le projet est à rendre **avant le** :

21 octobre 2018 à 23h59

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

```
2018-10-09  yrg <yrg@irif.fr>

* Corrige ‘alien_infix_id‘ et ‘alien_prefix_id‘
* Corrige les variantes hexa, binaire et octale de ‘int‘
* Corrige la définition de la règle des déclarations externes
* Corrige la syntaxe du while

2018-09-14  yrg <yrg@irif.fr>

* Version initiale
```