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

version 1.0

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. Par ailleurs, le symbole « `##` » introduit un commentaire d'une ligne : tout ce qui suit ce symbole est ignoré jusqu'à la fin de la ligne.

1.2 Symboles

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

Les mots-clés sont les mots réservés aux constructions du langage. Dans ce document, ils seront écrits avec une police à chasse fixe (*monotype* en anglais). Par exemple, **if** et **while** sont des mots-clé.

Les identificateurs sont les terminaux auxquels une information est associée. Ils se subdivisent en plusieurs classes : les identificateurs de variables, d'étiquettes, de constructeurs de données, de constructeurs de types, ainsi que les littéraux. Les littéraux comprennent les entiers, les caractères et les chaînes de caractères. Les identificateurs seront désignés en utilisant une police sans empattement (*sans serif* en anglais). Par exemple, les classes type_con ou int désignent respectivement les constructeurs de type et les entiers littéraux. Les identificateurs sont définis par les expressions rationnelles ci-dessous.

var_id	≡ [a-z] [A-Z a-z 0-9 _]*	*Identificateur de variables préfixe*
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*
string	≡ " ((atom | ' | \") −{ " })* "	*Littéraux chaîne de caractères*
atom	≡ \000 | ... | \255 | \0x[0-9 a-f A-F]^2	
	| [printable] −{ ' } | \\ | \' | \n | \t | \b | \r	

Autrement dit, les identificateurs de valeurs, de constructeurs 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 peuvent comporter les mêmes caractères, mais doivent commencer par une majuscule. Les variables de type doivent débuter par un guillemet oblique.

Les entiers littéraux sont constituées de chiffres en notation décimale, en notation hexadécimale, en notation binaire ou en notation octale. Les entiers utilisent une représentation binaire sur 64 bits en complément à deux. Les constantes entières sont donc prises dans $[-2^{63}; 2^{63} - 1]$.

Les caractères littéraux sont écrits 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 ; les séquences d'échappement de trois chiffres décrivant le code ASCII du caractère en notation décimale ; les séquences d'échappement de deux chiffres décrivant le code ASCII en notation hexadécimale.

Les chaînes de caractères littérales 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). On peut y faire figurer un caractère par son code ASCII décrit via le même mécanisme que pour les caractères littéraux.

La ponctuation, comme les identificateurs, sera notée avec une police à chasse fixe. Par exemple « `(` » et « `=` » appartiennent à la ponctuation.

Symboles non-terminaux Les symboles non-terminaux seront notés à l'aide d'une police italique, 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 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 ::= let var_id [ : type_scheme ] = expr                           Valeur simple
              | fun fundef { and fundef }                                     Fonction(s)

fundef ::= [ : type_scheme ] var_id pattern = expr
```

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

```
type ::= type_con [ < type { , type } > ]      Application d'un constructeur de type
       | type -> type                          Fonctions
       | type * type { * type }                N-uplets (N > 1)
       | type_variable                         Variables de type
       | ( type )                              Type entre parenthèses

type_scheme ::= [ [ 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
       | var_id [ < [ type { , type } ] > ]                                   Variable
       | constr_id [ < [ type { , type } ] > ] [ ( expr { , expr } ) ]        Construction d'une donnée
       | ( )                                                                  Construction d'un 0-uplet
       | ( expr , expr { , expr } )                                           Construction d'un n-uplet (n > 1)
       | { label_id = expr { , label_id = expr } } [ < [ type { , type } ] > ]   Construction d'un enregistrement
       | expr . label_id [ < [ type { , type } ] > ]                          Projection d'un champ
       | expr ; expr                                                          Séquencement
       | vdefinition ; expr                                                   Définition locale
       | \ pattern -> expr                                                    Fonction anonyme
       | expr expr                                                            Application
       | expr binop expr                                                      Application infixe
       | match ( 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
       | do { expr } until ( expr )                                           Boucle non bornée et non vide
       | for var_id from ( expr ) to ( expr ) { expr }                        Boucle bornée
       | ( expr )                                                             Parenthésage
       | ( expr : type )                                                      Annotation de type
```

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

binop ::= `+` | `-` | `*` | `/` | `&&` | `||` | `=?` | `<=?` | `>=?` | `<?` | `>?`	*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{ , pattern }]* `)`	*N-uplets ou 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 cette grammaire de Hopix 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 dépôt Git du cours.

https://gaufre.informatique.univ-paris-diderot.fr/aguatto/compilation-m1-2023

Vous devez vous connecter à ce serveur Gitlab et vous créer un dépôt par branchement (*fork*) du projet `compilation-m1-2023`. **Il doit être privé**. L'arbre de sources contient les modules OCaml à compléter.

La commande `dune build` produit un exécutable appelé `flap.exe` et situé dans le répertoire `src`. On peut aussi l'exécuter avec la commande `dune exec ./src/flap.exe -- OPTIONS` depuis la racine de 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** :

25 octobre 2023 à 19h59

Le rendu est automatique si vous avez suivi la procédure décrite ci-dessus.

Pour finir, vous devez vous assurer des points suivants :

— Le projet contenu dans ce dépôt **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

10-04-2023 Version initiale.