Programmation fonctionnelle **Licence 3, Informatique**

Projet — Simplification d'expressions arithmétiques

Le but de cet exercice est d'écrire un simplificateur (très naïf, le problème de la simplification étant très complexe) d'expressions arithmétiques ordinaires. Pour pouvoir disposer confortablement d'expressions arithmétiques suffisamment complexes, on utilisera l'une des méthodes les plus simples pour les représenter, le langage de Łukasiewicz postfixé (encore appelé, notation polonaise inversée), dans lequel, on représente les opérandes d'un opérateur avant l'opérateur lui-même. L'intérêt de ce langage est qu'il ne nécessite pas l'introduction des priorités relatives entre les opérateurs, et que l'on peut exprimer toute expression arithmétique en n'utilisant aucune parenthèse. L'inconvénient est que chaque opérateur doit avoir une arité bien déterminée (pas question d'utiliser le signe moins pour la soustraction et pour l'opposé). Voici une expression dans ce langage : 13 2 5 × 1 0 / − + qui représente l'expression habituelle $13 + 2 \times 5 - 1/0$. Comme on l'a dit plus haut, les symboles représentent des opérateurs ayant une arité bien définie : si l'on veut un opérateur de soustraction de deux nombres (binaire, $x - y$) et un opérateur d'opposé (unaire, $-x$) alors il faut les représenter par deux signes distincts. Dans notre cas, on supposera que les opérateurs possibles sont

- ×, +, −, / tous quatre d'arité 2 ;
- ~ d'arité 1, représentant le moins unaire ;
- par ailleurs, dans ce langage, il faut considérer que les constantes et les variables sont des opérateurs d'arité 0 (ce qui est tout à fait cohérent).

Analyse lexicale

C'est la partie réellement fastidieuse du travail. Tout d'abord, il faut définir des fonctions permettant de transformer une suite de caractères (chiffres, les cinq symboles d'opérateurs ci-dessus, les lettres minuscules représentant 26 variables possibles, des espaces, et le point-virgule indiquant la fin d'une expression) en suite de valeurs (des lexèmes) d'un type somme permettant la représentation abstraite d'un opérateur (cinq constructeurs sans argument permettant de représenter les cinq opérateurs), la représentation d'un entier positif ou nul (seuls nombres que l'on manipule, un constructeur avec un argument entier), et des variables (un constructeur ayant un argument qui est une lettre minuscule). Imaginons que le type somme en question s'appelle *token*, alors on a besoin de deux fonctions :

— *input_to_token_list* de type *unit* → *token list* qui lit l'entrée standard (le clavier, ou bien un fichier, par redirection) et construit une liste de lexèmes représentant les symboles de l'expression arithmétiques fournie sur l'entrée standard ;

— *string_to_token_list* de type *string* → *token list* qui analyse une chaîne de caractères représentant une expression postfixée et construit la liste de lexèmes associée.

La première fonction, *input_to_token_list* sera utilisée dans le programme final, alors que la fonction *string_to_token_list* pourra être utilisée pour faire des tests sous l'interpréteur. On ne peut pas mélanger facilement les entrées-sorties (surtout les entrées de l'utilisateur...) et l'interpréteur car les deux sont en concurrence d'accès sur le clavier.

On notera que le seul point délicat dans l'analyse lexicale est l'identification des nombres (qui peuvent avoir plusieurs chiffres) ; on prêtera une attention toute particulière au fait qu'un nombre peut être immédiatement suivi d'un opérateur, sans espace entre les deux. Les noms de variables étant limités à une seule lettre ne posent pas de problème particulier. Il est également important de savoir que l'analyseur lexical s'arrêtera à la moindre erreur lexicale (typiquement, caractère interdit) en levant une exception *Lexical_Error* qui précisera le caractère en cause.

Pour donner un exemple, l'appel de fonction : *string_to_token_list* `"34␣56␣2+␣x␣*␣-;"` vous fournira la liste [*Number*(34) ; *Number*(56) ; *Number*(2) ; *Add* ; *Variable*('x') ; *Multiply* ; *Subtract* ; *End*] (à titre d'information, l'expression postfixée représente ici l'expression infixée : $34 - ((56 + 2) \times x) = 34 - (56 + 2)x$). Toute expression postfixée sera terminée par un point-virgule.

Remarque importante

L'analyseur lexical vous est fourni dans un module qui s'appelle *Expression_scanner*, donc, pas la peine de programmer cette partie. Non-non, ne nous remerciez pas ! Voici la signature de ce module :

```
type token =
  | Add    (* Binary operator for addition (+) *)
  | Subtract    (* Binary operator for subtraction (-) *)
  | Multiply    (* Binary operator for multiplication [*] *)
  | Divide    (* Binary operator for division (/) *)
  | Minus    (* Unary operator for opposite (~) *)
  | Variable of char    (* Variable names are lowercase unaccented letters *)
  | Number of int    (* Only nonnegative integer numbers *)
  | End    (* End of expression (;) *)
val input_to_token_list : unit -> token list
val string_to_token_list : string -> token list
exception Lexical_Error of string
```

Pour pouvoir l'utiliser, vous devez charger le module, via une directive `# load "expression_scanner.cmo"` suivie, éventuellement, d'une directive **open** *Expression_scanner* pour avoir pleine visibilité sur le module. Le fichier `expression_scanner.mli` vous est fourni pour votre information, vous y trouverez le type exact dans lequel vous seront fournis les résultats de l'analyse lexicale (c'est ce fichier que l'on a listé ci-dessus). Quant à eux, les fichiers `expression_scanner.cmi` (version semi-compilée de la spécification du module) et `expression_scanner.cmo` (version semi-compilée de l'implémentation du module) vous sont également fournis et doivent impérativement être accessibles depuis l'interpréteur pour pouvoir utiliser le module *Expression_scanner* (regardez la documentation de l'interpréteur de CAML pour voir comment les rendre accessibles) ; ils sont également indispensables si vous voulez produire un programme exécutable au lieu d'une collection de fonctions utilisables sous l'interpréteur.

Analyse syntaxique et construction de l'arbre

Pour toute la suite (votre travail), il faut partir de la suite de tokens construite par l'analyseur lexical ; si, d'aventure, les tokens que fournit l'analyseur lexical ne vous conviennent pas, vous pouvez toujours écrire une fonction de traduction qui les transforme en ce que vous voulez, mais vous noterez que le type *token* plus haut transporte exactement l'information voulue à la sortie d'un analyseur lexical pour ce langage.

L'analyse syntaxique consiste à construire, à partir de la liste de lexèmes (les tokens), un arbre de syntaxe abstraite (c'est comme ça que ça s'appelle) représentant l'expression arithmétique et c'est sur cet arbre que seront effectuées toutes les opérations de simplification. L'intérêt du langage de Łukasiewicz est précisément que l'analyseur syntaxique est extrêmement simple.

On démontre, en théorie des langages, qu'un mot du langage de Łukasiewicz postfixé est bien formé si, lorsque l'on attribue aux opérateurs d'arité i la valeur $i - 1$ (comme on l'a dit plus haut, les constantes ou les variables sont considérées comme des opérateurs d'arité 0), alors la somme des valeurs de tous les composants d'un mot est -1 et les sommes

partielles sur un suffixe strict du mot ont toutes une valeur > -1. Si vous le souhaitez, vous pouvez écrire une fonction qui détermine si une liste de tokens est un mot de Łukasiewicz postfixé (ce n'est pas obligatoire, l'analyse syntaxique décrite ci-dessous faisant elle aussi cette vérification).

Le principe de l'analyse d'une expression de Łukasiewicz est le suivant : on utilise une pile et on procède de la manière suivante en examinant la liste de tokens :

— lorsqu'on rencontre une variable ou une constante (par exemple, $\mathit{Variable}(x)$ ou $\mathit{Number}(32)$), on empile l'élément de type *tree* correspondant (c'est-à-dire [confer plus bas la définition du type *tree*] une valeur $\mathit{Var}(x)$ ou bien $\mathit{Cst}(32)$) ;

— lorsqu'on rencontre un opérateur d'arité i (ici quatre opérateurs sont d'arité 2 et un opérateur est d'arité 1), on extrait i items de la pile, et on assemble un arbre constitué de l'opérateur à la racine et des i items comme sous-arbres, puis on empile l'arbre qui vient d'être construit (cette règle implique que lorsqu'on rencontre une constante ou une variable — qui sont des opérandes, donc, des opérateurs d'arité 0 —, on se borne à les empiler, ce qu'on a écrit juste avant).

Si vous ne voyez pas bien, faites un essai d'évaluation à partir d'une expression sans variables : on empile les constantes, et quand on voit un opérateur, on dépile ce qu'il faut pour évaluer l'opérateur, et on empile le résultat de l'application de l'opérateur à ses opérandes ; à la fin de ce procédé, dans la pile, se trouve la valeur de l'expression.

Écrire une fonction *parse* dont la signature est *token list* $\to$ *tree* qui, à partir d'une suite de tokens représentant une expression arithmétique, construit un arbre de syntaxe abstraite la représentant. Vous aurez préalablement défini soigneusement le type des arbres de syntaxe abstraite que vous allez manipuler ensuite. L'analyse lexicale, outre vérifier les aspects théoriques (c'est fait automatiquement lorsque l'on effectue l'analyse syntaxique comme précisé plus haut) devra s'assurer qu'une expression se termine par un point-virgule (le token *End*) et signalera une erreur dans le cas contraire. Toutes les erreurs seront fatale (aucun traitement d'erreur n'est demandé), ce qui signifie que la première erreur de structure de l'expression termine le programme sans aucun rattrapage possible...

Voici le type des arbres de syntaxe abstraite, épuré pour ne contenir que ce qui est vraiment indispensable :

```
type operator = | Plus | Minus | Mult | Div
type tree =
  | Var of char
  | Cst of int
  | Unary of tree
  | Binary of operator * tree * tree
```

À partir de l'instant où l'on a construit l'arbre de syntaxe abstraite, on peut commencer à travailler sur l'expression qu'il représente : un arbre est la représentation la plus concrète et la plus fidèle qui soit d'une expression arithmétique (ou plus généralement d'un programme écrit dans un langage de programmation quelconque).

Simplification sur l'arbre

La simplification de l'expression arithmétique à partir de l'arbre de syntaxe abstraite se fait en appliquant les règles qui sont présentées ci-dessous (plus, éventuellement, d'autres si vous avez du courage à revendre). Ces règles sont peu nombreuses, et correspondent à des simplifications élémentaires ; dans ce qui suit, x désigne une sous-expression quelconque :

— une sous-expression constituée exclusivement de constantes entières sera évaluée ;

— une expression de la forme $1 \times x$ ou $0 + x$ sera simplifiée en x ; de même l'expression $0 \times x$

sera simplifiée en 0 ;
— une sous-expression de la forme $x - x$ sera simplifiée en 0 ;
— de manière analogue, une sous-expression de la forme x/x sera simplifiée en 1 (bien que ce soit un peu abusif).

Attention, on ne demande pas de simplifier une expression de la forme $x + y - x$ en y, car cela fait intervenir la commutativité de l'addition, et peut conduire facilement à des programmes qui bouclent... enfin, disons qu'il faut faire un peu de théorie avant de s'attaquer à ces choses-là.

Affichage du résultat

Enfin, une fois les simplifications terminées, il faudra afficher l'expression sous la forme simplifiée ; cet affichage devra éliminer le plus possible de parenthèses inutiles — et au moins les parenthèses d'associativité, ce qui signifie, par exemple, qu'une expression comme $((a \times b) \times c) \times (e + f)$ sera rendue par $a \times b \times c \times (e + f)$. On n'exige pas que soient supprimées toutes les parenthèses rendues inutiles par la priorité relative des opérateurs (par exemple, afficher $a + (b \times c)$ comme $a + b \times c$, mais si le cœur vous en dit...).

Programme final

Le programme final devra être compilé ou semi-compilé de manière à pouvoir être exécuté en dehors de l'interpréteur et permettre la simplification d'une suite d'expressions postfixées (chacune terminée par un point-virgule) ; à défaut, on pourra se limiter à n'analyser que des chaînes de caractères sous l'interpréteur (dans ce cas, pas de flot d'entrée si on ne veut pas s'exposer à de petits problèmes de concurrence d'accès au flot d'entrée).

Voyons cela un peu plus en détail ; la suite d'expressions que l'on fournit au programme doit subir les traitements que voilà :
— transformation de chaque expression en flot de tokens ; et pour chaque expression :
— transformation de la suite de tokens en arbre de syntaxe abstraite ;
— affichage de l'expression (infixée habituelle) sans aucune simplification ;
— application des simplifications sur l'arbre de syntaxe abstraite ;
— affichage du résultat qui est l'expression arithmétique infixée simplifiée.

Comme on l'a dit avant, l'idéal est d'avoir un programme qui fait tout ça et qui tourne en dehors de l'interpréteur CAML, mais si vous n'y arrivez pas, on tolèrera une série de fonctions parfaitement spécifiées utilisables sous l'interpréteur à partir d'expressions fournies dans des chaînes de caractères.

Prenons un exemple un peu développé : on part de l'expression arithmétique suivante :

$$(x + 3 + (5 + 7))/(3 \times 4/(1 + 3))$$

L'expression correspondante en notation polonaise inversée que l'on pourra fournir au programme est:

- `x 3 + 5 7 + + 3 4 * 1 3 + / /`

Une fois cette expression analysée et transformée en arbre de syntaxe abstraite, le premier affichage avant simplifications sera

- `((x + 3) + (5 + 7)) / ((3 * 4) / (1 + 3))`

Et l'affichage après les simplifications (évaluations de sous-arbres formés de constantes, ici, rien d'autre, je crois, si ce n'est l'utilisation de l'associativité gauche de l'addition), on obtiendra finalement :

- `(x + 3 + 12) / 3`

Difficile d'aller plus loin, je crois (pas impossible ! difficile!).