TP Compilation

Le but de ce TP est d'introduire les outils **ocamllex** et **menhir** qui serviront en projet. Ils sont ici mis en œuvre sur un exemple très simple, afin d'en comprendre les mécanismes généraux.

Description du langage source :

Aspects syntaxiques: Un programme dans le langage considéré est constitué d'une séquence éventuellement vide de définitions de variables, suivie d'une expression comprise entre un `begin` et un `end`. Le résultat du programme est le résultat de l'évaluation de cette expression finale, qui contiendra en général des références aux variables définies précédemment.

Une définition de variable est constituée du nom de la variable, suivi du symbole `:=`, d'une expression arithmétique **à valeurs entières** et est terminée par `';'`. Les expressions arithmétiques sont construites à partir de constantes et variables à valeurs entières, des quatre opérateurs arithmétiques binaires habituels et des `'+'` et `'-'` unaires. Le langage définit une **expression** `if then else` à valeurs entières. La partie « condition » du `if` consiste en deux expressions arithmétiques reliées par un des opérateurs de comparaison habituels (en notant par `=` l'égalité, et `<>` la non-égalité). Les mot-clefs `then` et `else` sont suivis d'une expression arithmétique. La partie `else` est obligatoire de façon que le `if` ait une valeur quelle que soit la valeur de la condition. **Les opérateurs de comparaison ne sont autorisées que dans la partie « condition » du `if`**. Dans un premier temps on ne considère **pas** les opérateurs logiques.

Les opérateurs arithmétiques ont leurs précédence et associativité habituelles. Toute expression, arithmétique ou de comparaison, peut être parenthésée.

Aspect lexicaux: les commentaires suivent les conventions du langage C. Le format des identificateurs et des constantes est le format habituel, sauf qu'on interdit l'usage du symbole `'_'` dans les identificateurs.

Aspects Contextuels: toutes les expressions étant entières, il n'y a pas besoin de contrôle de type ; une expression n'a le droit de référencer que des variables déjà définies. Une variable ne doit pas être définie plusieurs fois. Le contrôle de portée des identificateurs est fait statiquement, avant toute exécution.

Exemple de programme dans ce langage

```
x := 3;
y := 12 + x;
z := x + 2 * y;
t := if z < y then 1 else y + 3;
begin 1 * if t < z then x * y + z * t else 1 end
```

Le résultat de l'évaluation de ce programme doit donner `639 (soit 3*15 + 33*18)`.

Le TP comporte plusieurs parties (avec des extensions possibles pour ceux qui iraient plus vite) :

- Pour la première partie, vous devez produire les analyseurs lexical et syntaxique à l'aide de `occamlex` et `menhir` et construire des arbres de syntaxe abstraite (AST) que vous afficherez ensuite. Votre programme se contentera de reconnaître les programmes bien formés vis-à-vis des aspects lexicaux et syntaxiques uniquement et de construire leurs AST.
- Dans la seconde partie vous devez enrichir votre solution de façon à obtenir un **interprète** pour ce langage de programmation très simple
- Dans la troisième partie vous devez modifier votre programme pour obtenir un **compilateur** à destination de la machine abstraite du projet, plutôt qu'un interprète.

1ère partie : réalisation d'un « reconnaisseur »

Pour débuter cette première étape vous disposez des éléments suivants

- Un fichier `ast.ml` qui contient des définitions (**à compléter**) de type `ocaml` pour vos futurs AST.
- le fichier `tpParse.mly` contient un squelette de fichier pour `menhir`. Ce fichier est **à compléter** pour obtenir un analyseur syntaxique complet et correct, cohérent avec l'analyseur lexical.
- Le fichier `tpLex.ml` contient un analyseur lexical réalisé avec `ocamllex`. Dans cette première séance, ce fichier est mis à votre disposition.
- le fichier `test_lex.ml` pour vous aider à visualiser ce que renvoie l'analyseur lexical: il se contente d'appeler l'analyseur lexical et d'imprimer des messages selon les tokens qu'il reçoit. Utile pour vérifier ce que renvoie l'analyseur lexical (surtout quand vous aurez à réaliser le votre).
- le fichier `main.ml` lance l'analyse syntaxique et, ultérieurement, les autres phases de la compilation.
- le fichier `Makefile` produit les différents exécutables
- le répertoire `test` avec plusieurs fichiers d'exemples, corrects et incorrects.

Travail à réaliser pour la première séance

- Compléter la partie grammaticale, i.e. la description des déclarations et des expressions, (`tpParse.mly`) pour obtenir un analyseur syntaxique avec `menhir`. Vérifier l'absence de conflits ou leur bonne résolution. Votre grammaire n'a pas besoin d'action associée aux productions de la grammaire et il se contente de reconnaître si un programme est syntaxiquement correct ou non.
- Lorsque ceci est terminé et correct, compléter (si besoin) les fichiers `ast.ml` et `tpParse.mly` de manière à construire des AST représentant le programme source et à en imprimer une représentation textuelle **non ambiguë** afin de pouvoir vérifier si les précédences et associativités sont bien gérées.

Extension: opérateurs unaires '+' et '-' et opérateurs logiques

- Ajouter les opérateurs unaires '+' et '-' avec leurs priorités usuelles.
- Dans la partie expression logique du `if`, autoriser des opérateurs `not`, `or` et `and`, munis des priorités et associativités du langage C. Ces opérateurs ne peuvent apparaître que pour combiner des conditions dans un `if` (i.e. il n'y a **pas** de booléean dans le langage).

Pour cette extension il faudra utiliser l'analyseur lexical `tpLex2.ml` au lieu de `tpLex.ml` et mettre à `testeLex.ml` pour ajouter le traitement des nouveaux tokens.