

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. On ne considère **pas** les connecteurs logiques « et », « ou » et « non »; **les opérateurs de comparaison ne sont autorisés que dans la partie « condition » du if.**

Les opérateurs arithmétiques ont leurs précédence et associativité habituelle. 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 visibilité 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 **ocamllex** 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.
- Diverses extensions sont possibles pour ceux qui finiront avant la fin de la séance.

1ère partie : réalisation d'un « reconnaisseur »

Pour débiter cette première étape vous disposez des éléments suivants

- Un fichier `ast.ml` qui contient des définitions 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.mll` qui contient une ébauche de description pour construire votre futur analyseur lexical avec `ocamllex`.
- 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 tester votre propre analyseur lexical.
- 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 :

- 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.
- Compléter la partie lexicale pour obtenir votre analyseur lexical (`tpLex.mll`) et testez-le. Vérifiez la combinaison lexicale+syntaxique sur les exemples mis à disposition.
- 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.

2ème partie : réalisation d'un « interprète »

Étendre le « reconnaisseur » précédent afin d'obtenir un **interprète** pour le langage source (parties « vérifications contextuelles » et « évaluation »). Le travail réalisé consiste principalement à gérer « l'environnement » qui représente les couples (*nom de variable, valeur de la variable*) et à écrire une fonction qui réalise les vérifications contextuelles ainsi qu'une fonction d'évaluation d'une expression arithmétique représentée par un arbre de syntaxe abstraite; l'interprète recherche dynamiquement les valeurs des variables dans l'environnement courant.

3ème partie : réalisation d'un compilateur

Dans cette partie modifiez le parcours de vos AST pour engendrer du code pour la machine virtuelle, en stockant le code dans un fichier (le code mis à disposition dans le fichier `compil.ml` prévoit l'écriture du code dans un fichier nommé `out`). Vous pourrez ensuite exécuter le code à l'aide du simulateur de la machine virtuelle mis à disposition.