Polytech'Paris-Sud **4ème année Informatique**

TP Compilation

Le but de ce TP est d'introduire les outils Flex et Bison qui serviront en projet. Ils sont ici mis en œuvre sur un exemple de portée très réduite, un interpréteur d'un langage de programmation 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 et des quatre opérateurs binaires habituels (on ne considère par les `'+'` et `'-'` unaires dans un premier temps). 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. Dans un premier temps on ne considère pas les connecteurs logiques « et », « ou » et « non »; les expressions de comparaison ne sont autorisées que dans la partie « condition » du `if`.

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: Une expression ne peut référencer que des variables déjà définies. Une variable ne doit pas être définie plusieurs fois dans la séquence. 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`.

Dans ce TP, la réalisation de cet interpréteur se fera en deux étapes[1] :

- Lors de la première étape vous devez construire un simple « reconnaisseur » qui se limitera aux aspects lexicaux et syntaxiques du langage. Votre interpréteur se contentera donc de reconnaître les programmes bien formés vis-à-vis de ces deux aspects uniquement.
- Dans la seconde partie vous devez associer des règles de définitions d'attributs pour implémenter les règles de vérifications contextuelles propres à ce langage de programmation ainsi que pour simuler l'exécution des programmes corrects.

1Diverses extensions sont proposées pour ceux qui termineraient rapidement les premières étapes.

1ère partie : réalisation d'un « reconnaisseur »

Pour débuter cette première étape vous disposez des éléments suivants :

- un fichier `tp.h` qui contient différentes macros (qui serviront ultérieurement pour la construction d'arbres de syntaxe abstraite) et des définitions de types qui peuvent vous être utiles.
- un fichier `tp.y` qui contient un squelette de fichier pour Bison. Ce fichier est **à compléter** pour obtenir un analyseur syntaxique complet et correct, cohérent avec l'analyseur lexical. Dans un premier temps il s'agit de lister les unités lexicales nécessaires et d'écrire une grammaire correcte du langage.
- un fichier `tp.l` qui contient quelques éléments de l'analyseur lexical à produire. Ce fichier est **à compléter** pour obtenir un analyseur lexical complet et correct pour le langage défini.
- un fichier `test_lex.c` pour aider à mettre au point votre analyseur lexical: il se contente d'appeler l'analyseur produit par Flex et d'imprimer des messages selon ce qu'il reçoit. Sa fonction `main` reconnaît l'option `-v` (« verbose », désactivée par défaut) et dans ce cas imprime au fur et à mesure les unités reconnues. **Ce programme doit être complété par vos soins s**i vous modifiez la liste d'unités reconnues à l'analyse lexicale. Son seul intérêt est de faciliter le test de votre analyseur lexical; il n'est pas utilisé dans la suite du TP ni dans le projet.
- Un fichier `print.c` qui contient différentes fonctions pour imprimer une version complètement parenthésée des arbres abstraits, **si** vous associez des actions à vos productions grammaticales pour construire des arbres abstraits (à faire dans un second temps). La fonction `pprint` de ce fichier est à compléter au fur et à mesure des ajouts à la grammaire et à l'analyseur lexical.
- Un fichier `tp.c` qui sert à lancer l'analyseur syntaxique et, ultérieurement, l'interprète. La fonction `main` du fichier `tp.c` reconnaît l'option `-v` dont le but est d'imprimer les arbres abstraits (à vous d'insérer les appels aux fonctions définies dans `print.c`).
- Un fichier `Makefile` pour produire les différents exécutables (dont `tp` et `test_lex`).
- Un répertoire `test` avec quelques fichiers d'exemples, corrects et incorrects.

Travail à réaliser *:*

- Compléter la partie grammaticale, i.e. compléter la description des déclarations et des expressions, (`tp.y, tp.c`) pour obtenir un analyseur LaLR(1) via Bison. Vérifier l'absence de conflits ou à défaut leur bonne résolution (fichier `tp_y.output`).
- Compléter l'analyseur lexical en ajoutant les opérateurs et constructions manquants (`tp.l, tp.h, test_lex.c`).
- Compiler (`make test_lex`) l'analyseur lexical et le tester sur les fichiers de test.

Une fois que les analyseurs lexical et syntaxique semblent corrects :

- Ajouter des actions sémantiques aux productions grammaticales pour construire des arbres de syntaxe abstraite et les afficher (voir les fonctions disponibles dans `print.c`), afin de vérifier la bonne gestion des priorités d'opérateurs
- Compiler (`make tp`) l'analyseur syntaxique et le tester sur quelques uns des fichiers de test.

Avertissement : les fichiers fournis (`tp.l`, `tp.y`, `tp.c`, etc.) sont **incomplets** mais permettent :

- De compiler une ébauche d'analyseur lexical
  `make test_lex`
- De visualiser le résultat produit sur un des fichiers de test :
  `./test_lex -v test/enonce1.txt`
- De produire un reconnaisseur complet et de le tester :
  `make`
  `./tp` *`fichier-contenant-le-programme`*

2ème partie : réalisation de l'interpréteur

Cette seconde partie consiste à étendre le « reconnaisseur » précédent afin d'obtenir un **interprèteur** pour le langage source (parties « vérifications contextuelles » et « évaluation »). Vous disposez:

- du fichier `tp.c` qui contient la fonction `main` de l'interprète ainsi que le code de certaines fonctions utiles dont les appels proviendront directement ou indirectement des actions que vous ajouterez aux productions grammaticales. La fonction `main` du fichier `tp.c` implémente l'option **`-v`** décrite précédemment et l'option **`-e`** (pour « `noEval` ») dont le but est de bloquer l'évaluation des expressions du programme source (par exemple pour ne s'occuper que des vérifications contextuelles)[2].
- du fichier `tp.h` avec ses macros et ses définitions de types (dont `YYSTYPE`).
- du répertoire `test` avec des exemples. Le résultat attendu est indiqué en commentaire en fin de fichier.

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 abstrait; l'interpréteur recherche dynamiquement les valeurs des variables dans l'environnement courant.

3ème partie: extensions (une fois l'interprète de base achevé)

- Ajouter des opérateurs `'+'` et `'-'` **unaires**, en plus de leurs homologues binaires.
- Dans la partie expression logique du `if`, autoriser les opérateurs `not`, `or` et `and`, munis des priorités et associativités du langage C. La conjonction et la disjonction doivent avoir un comportement « paresseux », comme en langage C. Ces opérateurs ne peuvent apparaître que pour combiner ds conditions dans un `if` (i.e. il n'y a pas de booléean dans le langage).
- autoriser les séquences d'opérateurs de comparaison. On autorise maintenant des expressions telles que `a <= b + c = d + e < g`, dont l'interprétation devra être équivalente à celle de l'expression `a <= b + c and b + c = d + e and d + e < g`. De même pour des expressions telle que `a = b = c != d.`
- Ajouter des chaînes de caractères au langage (au format des chaînes en C) ainsi qu'un nouveau type d'expression: `put(str, int)` qui prend en paramètre une chaîne de caractères et une expression entière : `put` évalue tout d'abord l'expression, puis imprime ses deux paramètres de gauche à droite et renvoie en résultat la valeur de l'expression. Le seul contexte où une chaîne de caractères peut apparaître c'est comme premier argument du `put`.
- Traiter les déclarations de variables au fur et à mesure qu'on les rencontre. L'expression qui définit la valeur d'une variable ne dépend pas de la suite du programme et peut donc être évaluée dès qu'on rencontre la fin de la déclaration et il est inutile de conserver l''arbre syntaxique correspondant.

---

2 Le code actuel positionne la variable `noEval` à `TRUE`, sa valeur par défaut étant `FALSE`. Cette variable sert à bloquer les évaluations d'expressions dans les fonctions que vous ajouterez.