Projet Compilation – TP Introductif

Avant-Propos : le but des premières séances de TP est d'introduire les outils Flex et Bison qui serviront ultérieurement dans le cadre du projet. Ils seront ici mis en œuvre sur un exemple de portée beaucoup plus réduite, afin d'en comprendre les mécanismes généraux.

Flex: générateur d'analyseur lexical à partir d'un fichier d'abréviations, d'expressions régulières et d'actions à exécuter lorsqu'on reconnaît une occurrence d'expression régulière.

Bison: générateur d'analyseur syntaxique à partir d'une grammaire LaLR(1).

Flex et Bison sont conçus pour s'intégrer facilement.

Description du langage :

Un programme est constitué d'une séquence éventuellement vide de déclarations 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 déclarées précédemment.

Une déclaration de variable est constituée du nom de la variable, suivi du symbole `:=`, d'une expression arithmétique à valeurs entières et est terminée par `';'`. L'expression ne doit référencer que des variables définies au préalable. Une variable ne peut être déclarée qu'une fois dans la séquence de déclarations.

Toutes les variables de l'expression finale doivent avoir été déclarées au préalable.

Les expressions peuvent comporter des variables, des constantes entières, les opérateurs arithmétiques et de comparaison habituels (en notant `=` l'égalité, et `<>` la non-égalité), une construction `if then else` qui a la valeur d'une expression (ie. il y a toujours une partie `else` de façon que le `if then else` renvoie toujours une valeur), ainsi qu'une instruction `get()` qui permettra lors de l'exécution du programme de lire une valeur entière dans un fichier de données passé en paramètre au lancement de l'interprète. Les opérateurs ont leurs précédence et associativité habituelles et le parenthésage permet de grouper les opérandes si besoin.

Il existe aussi une construction `put(str, int)`, qui prend en paramètre une chaîne de caractères (au format des chaînes en C) et une expression entière : `put` évalue l'expression, puis imprime ses deux paramètres de gauche à droite et renvoie la valeur de l'expression.

Enfin, on peut écrire des commentaires, comme dans le langage C.

Exemple de programme dans ce langage

```
x := 3;
y := 12 + x;
/* La valeur retournée par get ne sera connue qu'à l'exécution */
z := x + 2 * get();
t := if z > y then put("Then:", x + 3) else y + 3;
t2 := if get() <> 0 then 1 / 0 else 1 ;
begin
   put("Résultat", if t = z then x * y + z * t else 1 * z)
end
```

1ère partie : réalisation d'un reconnaisseur (analyseur lexical + syntaxique)

Pour débuter cette première étape vous disposez des éléments suivants :

- un fichier `tp.y` qui contient un squelette de fichier pour Bison. Ce fichier est à compléter pour avoir un analyseur syntaxique complet et correct, cohérent avec l'analyseur lexical.
- un fichier `tp.h` qui contient différentes constantes symboliques (qui serviront ultérieurement pour la construction d'arbres de syntaxe abstraite) et des définitions de types.
- 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 pour le langage défini.
- un fichier `tp_lex.c` qui se contente d'appeler l'analyseur produit par Flex et d'imprimer des messages selon ce qui a été reconnu par l'analyseur lexical. La fonction `main` du fichier `tp_lex.c` reconnaît l'option `-v` (verbose) et dans ce cas imprime au fur et à mesure les unités reconnues. Par défaut l'option est désactivée dans le programme de test.
- 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) et que vous respectez le format prévu pour ces arbres.
- 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` (verbose) dont le but devrait être d'imprimer les arbres abstraits (à vous d'insérer les appels aux fonctions définies dans `print.c`).
- Un fichier `makefile` pour lancer Flex et produire différents exécutables (dont `tp` et `tp_lex`).

- Un répertoire `test` avec des fichiers d'exemples (et des sous-répertoires pour les extensions à réaliser ultérieurement).

Travail à réaliser :

- Compléter les symboles que l'analyseur lexical doit retourner (abréviation `Symbol` dans `tp.l`).
- Ajouter les opérateurs et constructions manquants pour l'analyse lexicale (`tp.l`, `tp.h`, `tp_lex.c`).
- Prendre en compte les délimiteurs (abréviation `Delim` dans `tp.l`) et les caractères erronés.
- Éventuellement (car c'est sans incidence sur les étapes qui suivent) gérer le contenu des chaînes de caractères pour qu'elles s'affichent correctement.
- Compiler (`make tp_lex`) l'analyseur lexical et le tester sur quelques uns des fichiers de test.

Une fois que l'analyseur lexical semble correct :

- Compléter la partie grammaticale : `if`, `put`, `get`, opérateurs absents, etc. (`tp.y`, `tp.c`).
- Compiler (`make tp`) l'analyseur syntaxique et le tester sur quelques uns des fichiers de test.
- Ajouter des actions sémantiques aux productions grammaticales pour construire des arbres abstraits et les afficher (voir les fonctions disponibles dans le fichier `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 bien sûr incomplets mais vous permettent dès le départ :

- De compiler une ébauche d'analyseur lexical
  `make tp_lex`
- De visualiser le résultat produit sur un des fichiers de tests :
  `./tp_lex -v test/enonce1.txt`

2ème partie : réalisation de l'interprète et extensions

Cette seconde partie consiste à étendre le « reconnaisseur » précédent afin d'obtenir un **interprète** pour le langage source (partie évaluation, précédée d'une partie « vérifications contextuelles »).

- Le fichier `tp.c` 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` (verbose) décrite précédemment et prévoit l'option `-e` (noEval) dont le but devrait être de bloquer l'évaluation des expressions du programme source (par exemple pour ne s'occuper dans un premier temps que des vérifications contextuelles)[1].
  Le fichier `tp.c` contient une fonction `getValue()` qui doit être appelée à chaque exécution de `get()`. Cette fonction lit la prochaine donnée dans un fichier dont on aura fourni le nom au lancement de l'interprète. Ce lancement se fait de l'une des deux façons suivantes

  ```
  ./tp fichier-du-programme.txt
  ```

  ou

  ```
  ./tp fichier-du-programme.txt fichier-des-donnees.dat
  ```

  selon que le programme source contient ou non des `get()`. Le fichier de données contient des constantes entières en nombre au moins égal au nombre de `get` à exécuter, **chaque constante étant sur une ligne séparée.**

- Le répertoire `test` avec des exemples (et des sous-répertoires pour les extensions).
- Un fichier shell `test.sh` d'exécution automatique de tests (à utiliser ultérieurement).

  Appel :

  ```
  ./test.sh programme.txt
  ./test.sh repertoire-de-test
  ```

  Le fichier `test.sh`, permet l'enchaînement automatique et le contrôle des résultats des fichiers de test du répertoire précédent. Il est muni de diverses options (faire `./test.sh -h`). Pour que cette procédure de test fonctionne correctement, les conventions suivantes sont à respecter :
  - le programme source est dans un fichier d'extension `.txt`. Les impressions et le résultat final iront dans un fichier d'extension `.out`. Il est aussi possible de préparer les résultats attendus (par exemple pour tester les messages affichés) dans un fichier d'extension `.res` qui servira de « témoin » pour valider le contenu du fichier `.out`.
  - si le programme source est valide et s'exécute correctement, l'interprète doit émettre la chaîne `result:` suivie du résultat du programme et d'un retour-charriot.
  - Si le programme est incorrect pour une raison quelconque, l'interprète doit terminer correctement (sauf dans le cas où le programme source boucle à l'exécution), ne pas émettre ce message `result:` mais un message commençant par `error:` suivi d'un texte indiquant la nature de l'erreur.
  - Les fichiers des répertoires de test terminent avec un commentaire indiquant le résultat attendu ou un commentaire décrivant l'erreur. Par exemple

    ```
    begin if 1 = 0 then 1 else if 1 = 2 then 2 else 1 end
    /*
    result: 1
    */
    ```

Travail à effectuer:

1. Compléter les fichiers `tp.h`, `tp.c` et `tp.y` pour obtenir un interprète du langage.
2. Réaliser des extensions au langage, une fois compris le principe des différentes parties. Ces extensions sont de nature lexicale, syntaxique, sémantique ou mixtes ! Elles sont de difficulté croissante.
   a) Ajouter les opérateurs `and`, `or` et `not` de manière à pouvoir combiner des expressions logiques. L'opérateur `not` a la plus forte priorité, suivi de `and`, puis de `or`. Les opérateurs binaires associent à gauche. Ils sont "séquentiels", à la C : ils n'évaluent leur second opérande que si le premier ne suffit pas à obtenir le résultat.
   b) Deux opérateurs unaires : `+` et `-`, prioritaires sur les autres opérateurs.

---

1 Le code actuel se contente de positionner la variable `noEval` à `TRUE`, sa valeur par défaut étant `FALSE`. Cette variable pourra être testée pour bloquer les évaluations d'expressions dans les fonctions que vous ajouterez

c) Une construction `let id := E1 in E2` qui introduit une variable locale `id` utilisable dans `E2` et initialisée à la valeur associée à `E1`. La nouvelle variable peut éventuellement masquer une variable de même nom, déjà visible. Cette construction a la plus faible précédence.

Exemple de programme avec les extensions

```
x := 3;
x1 := if not x > 0 and x < 12 then 1 else x * 2;
x2 := if not x > 0 and x < 12 then 1 else let y := (x - 1) in y * x;
x3 := put("Cas0: ", if 3 < 2 then x1 else - x2 - - x2);
k := if not x <> 0 or x1 < 7 and x3 = x2 then 1 else 0;
k1 := put("Cas1: ", let x := 3 * x in x * let x := x + 1 in x * 2);
k2 := put("Cas2: ", let x := 3 * x in (x * let x := x + 1 in (x * 2)));
k3 := put("Cas3: ", (let x := 3 * x in x) * let x := x + 1 in (x * 2));
z := 2;
begin
    z
end
/*
result: 2
*/
```