Feuille de travaux pratiques n° 4
Utilisation de bison

Yacc (ou **bison** sur Linux) est un générateur d'analyseurs syntaxiques. Il prend un fichier en argument contenant une description d'une grammaire algébrique, en fait la sous-classe des grammaires algébriques reconnues par les analyseurs $LALR(1)$ (Look-Ahead Left Recursive). Il produit en sortie un fichier contenant la definition d'une table (en C ou C++) permettant de créer un analyseur syntaxique d'un flux de caractères (ou de tokens si on utilise aussi **flex**).

Le fichier en entrée de **bison** permet aussi d'associer chaque règle de la grammaire à une action qui est un bout de code C ou C++. Au cours de l'analyse syntaxique, l'analyseur va construire un arbre de dérivation et appliquer, pour chaque nœud produisant un non-terminal, l'action correspondant à la règle qui a produit ce non-terminal.

Premier programme

Bison est un générateur d'analyseur syntaxique pouvant générer du code C ou C++ suivant l'extension du fichier d'entrée. Avec l'extension **.y** du C sera produit et avec l'extension **.yxx**, cela sera du C++. Ici, nous utiliserons la version C++. Pour utiliser **bison**, vous devez saisir un petit fichier (avec votre éditeur de texte préféré) contenant la description de la grammaire et des actions associées aux règles. Voici un premier exemple que vous sauvegarderez sous le nom **exemple1.yxx** :

```
// Programme exemple1.yxx
//   bison exemple1.yxx
//   g++ -o exemple1 exemple1.tab.cxx
//   ./exemple1
//     1 0 1 + +<Entrée>
//      --> 2
//     01-1-<Entrée>
//      --> -2
//     3<Entrée>
//      erreur lors de l'analyse : syntax error

// ****** Prologue C++ ******
%{
       #define YYSTYPE int
       #include <iostream>
       int yylex (void);
       void yyerror (char const *);
%}

// ****** Grammaire ******
%%
input:  //
        | input ligne
;

ligne:   '\n'
        | exp '\n'      { std::cout << " --> " << $1 << std::endl;}
;
```

```
exp:      '0'           { $$ = 0 ;       }
        | '1'           { $$ = 1 ;       }
        | exp exp '+'   { $$ = $1 + $2; }
        | exp exp '-'   { $$ = $1 - $2; }

// ****** Épilogue C++ ******
%%

// Appelé par l'analyseur en cas d'erreur
void yyerror (char const *s){
       std::cout << "erreur lors de l'analyse : " << s << std::endl;
}

// Fait une analyse syntaxique caractère par caractère
int yylex (void){
       char c;
       std::cin >> c;
       while (c == ' ' || c == '\t') {
             std::cin >> c;
       }
       return c;
}

// Le programme principal
int main (void){
       std::cin.unsetf(std::ios_base::skipws);
       return yyparse ();
}
```

Ce fichier doit alors être traité par le programme **bison** pour qu'il le transforme en un programme C++. On peut utiliser la commande (tapée dans un terminal) :

```
bison exemple1.yxx
```

Si tout va bien (c'est-à-dire si **bison** n'a pas détecté d'erreur), vous devez trouver le fichier **exemple1.tab.cxx** dans le répertoire où vous avez lancé **bison**. On peut alors compiler ce programme en utilisant un compilateur C++ :

```
g++ exemple1.tab.cxx -o exemple1
```

Le résultat (l'exécutable) porte le nom **exemple1** (cela correspondant à l'option **-o exemple1**). Cet exécutable lit les caractères sur son entrée standard et écrit le résulat des actions sur sa sortie standard. La commande suivante lance ce programme avec pour entrée standard ce que vous tapez au clavier :

```
./exemple1
1 0 1 + +
01-1-
```

Le programme lit l'entrée standard ligne par ligne. Il évalue les expressions composées des caractères **0**, **1**, **+** et **-** en notation postfixée (voir la grammaire). Ce programme doit afficher **–> 2** pour la ligne **1 0 1 + +** et **–> -2** pour la ligne **01-1-**. Avec une ligne ne correspondant pas à la grammaire, le programme affiche **erreur lors de l'analyse : syntax error**.

Définition des fichiers sources de yacc/bison

Un fichier source **bison** est composé (comme pour **flex**) de trois parties délimitées par les caractères `%%` :

Définition
`%%`
Règles
`%%`
Fonctions et routines

Toutes les lignes précédant le premier délimiteur `%%` sont considérées comme une définition par **bison** ou bien, pour les lignes comprises entre `%{` et `%}`, comme du code C ou C++ copié tel quel dans le fichier généré par **bison**. Une définition sert à introduire les symboles terminaux qui ne sont pas de simple caractère (**%token**), les symboles non-terminaux avec leur type associé (**%type**), la priorité (**%precedence**) et l'associativité des opérateurs (**%left** et **%right**).

La seconde partie, délimitée avant et après par `%%` comporte une liste de règles grammaticales de la forme :

Résultat : Composant1 Composant2 ... { Actions C/C++ } | ... ;

Par exemple, **S : 'a' S {i++ ;} | 'b' 'c' S | 'd' {j++ ;} ;** correspond à la grammaire $S \rightarrow aS|bcS|d$ qui définit le langage rationnel $(a|bc)^*d$. Dans cet exemple, en plus, les actions associées comptent le nombre de a (dans la variable i) et le nombre de d (dans la variable j). Une règle peut être mise sur plusieurs lignes, le caractère **;** termine la définition de la règle courante.

Finalement, la dernière partie d'un source Lex, après le second `%%`, est recopiée à la fin du programme généré par **bison**. Il sert à définir des fonctions utiles à l'analyseur. Dans le premier programme, nous y avions défini la fonction `main` pour obtenir un programme C++ complet plus deux autres fonctions nécessaires à l'analyseur : la fonction **yylex** permettant de lire le prochain symbole terminal (doit retourner le code du caractère lu ou bien le code du token défini avec la déclaration **%token**) et la fonction **yyerror** appelée lorsque l'analyseur détecte une erreur (comme une erreur de syntaxe).

Exercice 4.1

1. A partir de l'exemple 1, modifier la grammaire pour qu'elle puisse accepter la grammaire ci-dessus :
   S : 'a' S {i++ ;} | 'b' 'c' S | 'd' {j++ ;} ;
   Il faudra penser à déclarer les variables **i** et **j** dans le préambule. Le programme devra terminer en affichant les valeurs de ces variables **i** et **j**.
2. Ecrire une grammaire qui reconnaisse le langage $\{a^n b^n \mid n > 0\}$ puis à partir du programme de l'exemple 1 écrire un analyseur syntaxique qui accepte les lignes contenant le mot du langage ci-dessus et qui affiche la valeur de n (pour chaque ligne).

Exercice 4.2

Dans l'exemple 1, les actions contiennent des macros **$$**, **$1**, **$2**, etc. Les macros **$1**, **$2**, etc, correspondent à la valeur retournée par l'action qui engendre un non-teminal. La macro **$$** désigne la valeur que retournera cette action. Le type de ces valeurs est défini par la macro **YYSTYPE** et peut aussi être précisé pour chaque non-terminal dans la partie déclaration (**%type**). Ce principe permet de calculer des valeurs associées à chaque nœud de l'arbre de dérivation à partir des valeurs des fils de ce nœud. On peut ainsi faire un petit calculateur. Pour cela, on peut reprendre l'exercice 3.2 du TD 3 :

$$G : (\quad \{a, b, ..., z, +, (,)\}, \{S, somme, produit, facteur, terme\}, S, \left\{ \begin{array}{lcl} S & \rightarrow & somme \\ produit & \rightarrow & produit\ facteur \mid facteur \\ facteur & \rightarrow & (\ somme\ ) \mid terme \\ somme & \rightarrow & somme + produit \mid produit \\ terme & \rightarrow & a \mid b \mid c \mid \cdots \mid y \mid z \end{array} \right\})$$

Dans un premier temps, on pourra utiliser les chiffres de 0 à 9 à la place des variables $a, ..., z$. On pourra ensuite ajouter des variables représentées par une lettre, ajouter une opération permettant de modifier une de ces variables, ajouter des nombres (et pas simplement des chiffres), d'autres opérations et fonctions mathématiques, utiliser des nombre flottant plutôt que des entiers (en changeant la définition de la macro **YYSTYPE**), etc.