# Compilation

## Flex & Bison: analyse lexicale et syntaxique

C. Alias

Le but des TPs de compilation est de vous faire écrire un compilateur simple pour architectures x86. Vu la limite de temps, il n'est bien sûr pas question de tout vous faire écrire. Vous aurez donc à récupérer des squelettes de programme, et à en compléter les parties importantes.

Ce premier TP à pour but de vous faire découvrir les (méta-)outils flex et bison, largement utilisés pour construire des compilateurs.

### Exercice 1. *Echauffement: la calculatrice*

Le but de cette partie est d'implanter une calculette, qui saisie une expression arithmétique, et qui calcule sa valeur. Télécharger `src0_calculette.tgz`, et décompacter avec `tar xvfz src0_calculette.tgz`.

```
Fichiers: lexer.l  = analyseur lexical    (à compiler avec *flex*),
          parser.y = analyseur syntaxique (à compiler avec *bison*)
          main.c   = fichier pilote
```

**Etat des lieux:** Regarder le contenu des fichiers. Ouvrir le makefile. A quoi sert l'option -d de bison ? (man) Donc: pourquoi bison doit être exécuté avant flex ?

**Dans lexer.l:**

1. A quoi correspondent les variables TK_* ? (regarder les %tokens dans `parser.y`...)
2. A quoi sert la ligne avec break; (L14) ?
3. Dans la partie à completer (L25-) ajouter l'expression régulière qui reconnaît un entier (positif).
4. Selon vous, a quoi sert la ligne avec yylval et yytext ? (regarder dans parser.y
5. Quel est le sens du . (L33) ?
6. Ouvrir main.c: a quoi correspondent yyin et yyparse() ?

**Dans parser.y:**

1. A quoi servent yylex() et yytext ?
2. A quoi sert le %start (L52) ?
3. On considère l'expression $1 + 2 * 3$.
   - Donner son arbre de dérivation.
   - Dans quel ordre les dérivations sont elles réduites ?
   - Ajouter les actions sémantiques pour afficher l'ordre des derivations, et vérifier.
4. Ajouter les attributs et les actions sémantiques pour calculer la valeur de l'expression.
5. Modifier les actions sémantiques pour afficher l'expression sous forme **préfixée**.
   Exemple: $+(1, *(2, 3))$ est la forme préfixée de $1 + 2 * 3$.
6. Modifier les actions sémantiques pour afficher l'expression sous forme **postfixée**.
   Exemple: $(1, (2, 3)*)+$ est la forme **postfixée** de $1 + 2 * 3$.

## Exercice 2. *Et maintenant... le compilateur!*

Maintenant que vous vous êtes bien échauffés, on va pouvoir passer aux choses sérieuses...
Télécharger `src1_lexyacc.tgz` et décompacter avec `tar xvfz src1_lexyacc.tgz`.

On souhaite compiler un langage impératif simple dont voici un exemple de programme:

```
function fact(int n)
int result
begin
  n = 10
  result = 1
  while n != 1 do
    result = result * n
    n = n - 1
  done
  return result
end

function main()
int fact5
begin
  fact5 = fact(5)
end
```

Ce langage ne manipule que des entiers. Un programme est une suite de fonctions, la derniere étant la fonction principale. On souhaite générer du code pour la machine x86 décrite en cours avec des instructions supplémentaires qui seront précisées au fûr et à mesure du TP.

**Etat des lieux:** Inspecter les fichiers lexer.l et parser.y. Répérer la hiérarchie des non-terminaux dans `parser.y`: expr (expressions) $\longrightarrow$ test (conditions) $\longrightarrow$ stmt (contrôle) $\longrightarrow$ function (fonctions) $\longrightarrow$ prog (programme).

**Dans lexer.l:**

1. Modifier le fichier pour compter les lignes (compter le nombre de retour-chariot)
2. Ajouter les entiers négatifs et les réels.
3. Ajouter l'expression régulière des identificateurs (L52-)
4. Bonus: ajouter les commentaires //

**Dans parser.y:**

1. Ajouter les !, && et || dans test (L117).
2. Ajouter le while (L141)
3. Ajouter les fonctions (L195). On utilisera le non-terminal decl_args pour la declaration des arguments, et le non-terminal decl_local_var pour la déclaration des variables locales.

Conclusion: Vous êtes maintenant fin prêts à générer le code! To be continued...