Université de Strasbourg

Master Informatique

Compilation — Travaux Pratiques n° 2
Analyse syntaxique ascendante

I. Premier exemple

Considérez la grammaire

$$\begin{array}{lcl} S & \rightarrow & CC \\ C & \rightarrow & cC \mid d \end{array}$$

Écrivez le source Yacc correspondant (sans actions) et compiler avec l'option -v. Vérifier que le fichier y.output contient bien la table d'analyse LALR.

II. Expressions arithmétiques

Utilisez la grammaire naïve des expressions arithmétiques :

$$\begin{array}{lcl} E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & (\ E\ ) \\ E & \rightarrow & \texttt{entier} \end{array}$$

pour implanter la calculatrice de base présentée en introduction. Vérifier que l'analyseur construit engendre de nombreux conflits. Lever les conflits en précisant les règles de priorité et d'associativité des opérateurs. Ajouter le moins unaire et vérifier que les priorités sont correctement gérées.

III. Interpréteur d'expressions ensemblistes

Vous devez réaliser, à l'aide des outils Lex et Yacc, un interpréteur d'expressions ensemblistes répondant aux spécifications suivantes (on considérera uniquement des ensembles d'entiers compris entre 1 et 32). Le vocabulaire est :

`a..z A..Z`	identificateur mono-lettre (ident) non "case-sensitive"
`{ }`	délimiteurs classiques des ensembles
`:=`	affectation
`,`	séparateur d'éléments
`1..32`	seuls éléments possibles
`\n`	fin de ligne
`union UNION`	union ensembliste
`inter INTER`	intersection ensembliste
`comp COMP`	complémentaire dans 1,...,32
`diff DIFF`	différence ensembliste

La grammaire est :

```
liste          ::= <empty>                  // sortie de l'interpréteur
                 | liste instruction '\n'   // interprétation de l'instruction
```

```
instruction    ::= ident ':=' expression   // calcul et affectation
                 | ident                   // affichage

expression     ::= operande                        // renvoie l'opérande
                 | operande operateur2 operande  // renvoie le résultat de
                 | operateur1 operande           //   l'opération ensembliste

operateur2     ::= 'UNION' | 'INTER' | 'DIFF' | 'union' | 'inter' | 'diff'
operateur1     ::= 'COMP' | 'comp'

operande       ::= ident
                 | ensemble

ensemble       ::= { }                     // ensemble vide
                 | { liste-elements }      // ensemble non vide

liste-elements ::= élément
                 | élément , liste-elements
```

Voici un exemple d'exécution :

```
A := { }             met l'ensemble vide dans A
a := { 1, 22 }       met dans A l'ensemble { 1, 22 }
b := a UNION { 3 }   met dans B l'union de A et du singleton { 3 }
B                    affiche l'ensemble B :
  { 1, 3, 22 }
C := { 1, 3, 5 }     met dans C l'ensemble { 1, 3, 5 }
A := B inter C       met dans A l'intersection de B et de C
a                    affiche l'ensemble A :
  { 1, 3 }
```

Indication : grâce à un codage des ensembles sur des entiers longs (32 bits), on pourra réaliser simplement les opérations ensemblistes avec les opérateurs "bitwise" du C/C++ (`&` `|` `~`)