Compilation: Lex & Yacc

Le 17 janvier 2016 , SVN-ID 382

Contents

1 Introduction

Création: année 70

Très Utilisés: 5/10% des logiciels
doxygen 10 lex, les compilateurs & interpéteurs, les WMs, ...

Fonction: générateurs de filtres et compilateurs
analyses lexicales et grammaticales

Avantages:

- garantir des automates d'analyse lexicale et grammaticale sans erreur
  (exemple: traiter les macros YKV(var) dans une chaîne de caractères:
  ".... YKV(var0) ... YKV(var1) ....")
- modification rapide (exemple: ajouter un autre type de macro YKE(var), ajouter une valeur par défaut YKE(var val))

Succès informatique:

2 Analyse lexicale

2.1 Fonction

Fonction générale En fonction d'un ensemble de (motif,traitement) génère un programme qui:

1. lit un flux d'entrée
2. reconnaît les motifs
3. effectue les traitements associés aux motifs

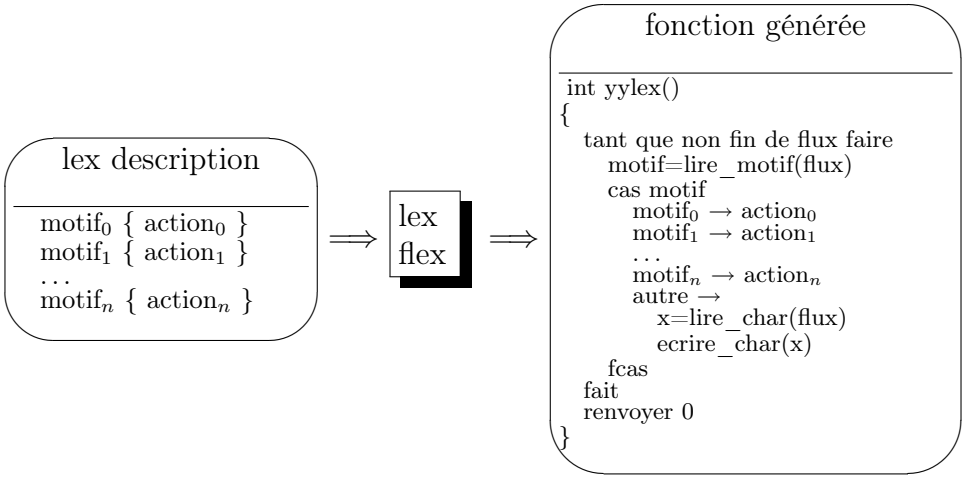
Exemples:

- transformer des fichiers unix en fichiers windows

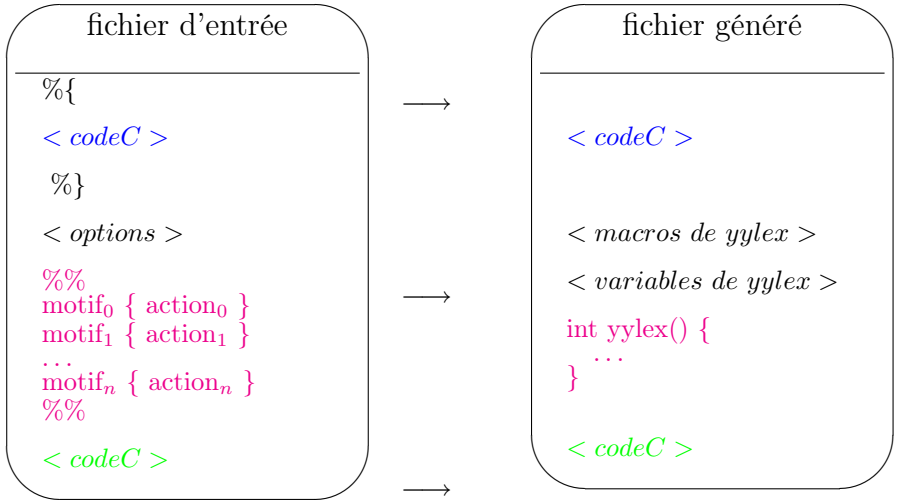
- transformer tous les caractères d'un fichier en majuscule
- correcteur orthographique

Différence avec sed, perl, ... le traitement n'est pas réduit à une substitution, le filtre est plus rapide.

2.2 Principe de lex/flex



2.3 Format de lex/flex



Attention: les commandes %... et les motifs doivent commencer en première colonne.

2.4 Motifs

Expressions régulières (base)

`x` match the character 'x'

`.` any character (byte) except newline

`\X` if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \X. Otherwise, a literal 'X' (used to escape operators such as '*')

`[xyz]` a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'

`[abj-oZ]` a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'

`[↑A-Z]` a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an

uppercase letter.

`[↑A-Z\n]` any character EXCEPT an uppercase letter or a newline

`"[xyz]\"foo"` the literal string: [xyz]"foo

`\0` a NUL character (ASCII code 0)

`\123` the character with octal value 123

`\x2a` the character with hexadecimal value 2a

`«EOF»` an end-of-file

Expressions régulières (composition)

`(r)` match an r; parentheses are used to override precedence

`r*` zero or more r's, where r is any regular expression

`r+` one or more r's

`r?` zero or one r's (that is, "an optional r")

`r{2,5}` anywhere from two to five r's

`r{2,}` two or more r's

`r{4}` exactly 4 r's

`rs` the regular expression r followed by the regular expression s; called "concatenation"

`r|s` either an r or an s

`r/s` an r but only if it is followed by an s.

`↑r` an r, but only at the beginning of a line.

`r$` an r, but only at the end of a line (i.e., just before a newline). Equivalent to "$r/\backslash n$".

Expressions régulières (extension)

{name} the expansion of the "name" definition

<**s**>**r** an r, but only in start condition s

<**s1,s2,s3**>**r** same, but in any of start conditions s1, s2, or s3

<*>**r** an r in any start condition, even an exclusive one.

<**s1,s2**>**«EOF»** an end-of-file when in start condition s1 or s2

`Caractères à "échapper"` / etc

`Exemples`

- un mot
- un mot commençant par une majuscule
- un entier
- une séquence sans blanc ni tabulation.
- un identifiant C
- un commentaire shell

2.5 Actions

- pour les sections "codes C" et les actions tout le langage C ou C++.
- la variable **yytext** (char*) contient le motif reconnu.
- la variable **yyleng** (int) contient le nombre de caractères de yytext.
- action par défaut: écriture du motif
- action ";" permet d'ignorer le motif

Attention: yytext est une zone mémoire fixe et donc réécrasée dans chaque action.

2.6 Exemple

Source

```
%option noyywrap
%%
[A-Z] { printf("%c",*yytext-'A'+'a'); }
```

Compilation & exécution

```
shell> flex fich.lex
shell> cc lex.yy.c -lfl
shell> ./a.out
abcd<return>
abcd
AbCd<return>
abcd
<CTL-D>
shell>
```

Remarques

- Par défaut le flux d'entrée est le "standard input"
- Par défaut le flux de sortie est le "standard output"
- Par défaut les motifs non reconnus sont écrits sur le flux de sortie.
- libfl.a contient le main

2.7 Quelques extensions

2.7.1 Commentaires et variables

Les lignes commençant par un blanc ou une tabulation sont copiées telles quelles dans le fichier généré.

```
 %option noyywrap

 %%
    int cnt=0;
 bon  cnt++;
    /* ceci est un commentaire */
 (.|\n) ;
 «EOF» { printf("%d\n",cnt); return 0; }
```

2.7.2 Définition de macro de motifs

```
ENTIER (\+|\-)?[0-9]+
VAR (v|V)ENTIER

%option noyywrap

%%
{ENTIER} ...
{VAR} ...
.|\n ;
```

2.7.3 Changement du flux d'entrée

Par défaut lex/flex lit sur yyin et écrit sur yyout. Ce sont des FILE* et ils sont initialisés à stdin et stdout.

```
 %option noyywrap

 %%
    int cnt=0;
 bon  cnt++;
    /* ceci est un commentaire */
 (.|\n) ;
 «EOF» { printf("%d\n",cnt); return 0; }
 %%

 int main(int argc,char**argv)
 {
    if (argc!=3) { ... ; return 1; }
    if ((yyin=fopen(argv[1],"r"))==0) { ... ; return 1; }
    if ((yyout=fopen(argv[2],"w"))==0) { ... ; return 1; }
    yylex();
    return 0;
 }
```

lex/flex utilise la macro "`YY_INPUT(buf,result,max_size)`" pour lire le flux d'entrée. Il suffit de la redéfinir pour lire où on veut.

```
 %{
 #define YY_INPUT(buf,res,nb) \
    if (inputbuffer && *inputbuffer) {\
       *buf=*inputbuffer++; res=1;\
    } else res=0;

 static char* inputbuffer;
 %}
```

```
%option noyywrap
%%
...
%%

int main(int argc,char**argv)
{
  if (argc!=2) { ... ; return 1; }
  inputbuffer = argv[1];
  yylex();
  return 0;
}
```

2.7.4 Plusieurs lex

Avec l'option -P de flex

```
bash> flex -Pmonyy fich.l
bash>
```

Le fichier générés s'appelle "lex.monyy.c", la fonction générée s'appelle "monyylex()", les variables des actions "monyytext" et "monyyleng".
Dans fich.l on peut toujours utiliser "yylex()", "yytext", et "yyleng".

Avec les "start-conditions"

```
%option noyywrap

%x code

%%
<INITIAL>BCODE { BEGIN code; printf("%s",yytext); }
<code>ECODE { BEGIN INITIAL; printf("%s",yytext); }
<code>[A-Z] { printf("%c",*yytext-'A'+'a'); }
```

2.8 Exercices

Exercice 1

Écrire un programme qui lit le fichier standard d'entrée et écrit sur le fichier standard de sortie et qui implante les filtres suivants:

filtre a) Il passe en majuscule la première lettre qui suit un '.', un '!' ou un '?' et séparée de ce caractère par au moins un espace. Un espace est une suite d'au moins 1 caractère parmi les caractères blanc, tabulation et fin de ligne.

filtre b) Il exécute le filtre a) si et seulement si le caractère de ponctuation n'est pas précédé du caractère '\'. Dans ce cas le '\' n'est pas imprimé.

filtre c) Il insère un blanc derrière les caractères de ponctuation '.', '!' et '?' suivis d'une lettre. De plus si la lettre est une minuscule, il la passe en majuscule.

filtre d) Il n'effectue pas le filtre c) si le le caractère de ponctuation est précédé de '\'. Dans ce cas le '\' n'est pas imprimé.

Exercice 2

Écrire un programme qui lit un fichier passé en argument et qui écrit "oui" sur le fichier standard de sortie si il contient 4 lignes consécutives contenant le mot "prof" suivi du mot "eleve". Autrement, il écrit "non". On écrira ce programme de deux manières différentes:

(a) en utilisant des variables locales

(b) en utilisant les contextes (start condition)

(c) des expressions regulières

(d) une expression regulière

exemple:
```
aaa prof aaa eleve aaa\n                      ⟶ oui
aaa\n aaa prof aaa\n\n\n eleve aaa\n aaa\n     ⟶ oui
aaa\n aaa eleve aaa\n\n\n prof aaa\n aaa\n     ⟶ non
aaa\n aaa prof aaa\n\n\n aaa\n eleve aaa\n     ⟶ non
aaa eleve aaa prof aaa\n                      ⟶ non
```

Exercice 3

Écrire un programme qui lit un fichier passé en argument et qui écrit "oui" sur le fichier standard de sortie si il contient 4 lignes consécutives contenant les mots "prof" et "eleve". Autrement, il écrit "non".

exemples:
```
aaa prof aaa eleve aaa\n                      ⟶ oui
aaa\n aaa prof aaa\n\n\n eleve aaa\n aaa\n     ⟶ oui
aaa\n aaa eleve aaa\n\n\n prof aaa\n aaa\n     ⟶ oui
aaa\n aaa prof aaa\n\n\naaa\n eleve aaa\n      ⟶ non
aaa eleve aaa prof aaa\n                      ⟶ oui
```

Exercice 4

Écrire un programme qui réécrit son argument après l'avoir passé dans le filtre suivant:

- Le caractère '%' est doublé si il n'est pas suivi du caractère '%'.
- Les séquences "\\" sont remplacées par le caractère '\'.
- Les séquences "\b", "\n", \f" sont remplacées par les caractères BEL (0x7), LF (0xa) et FF (0xc).

exemples:
```
aa%bb%%cc%%%dd ⟶ 'a' 'a' '%' '%' 'b' 'b' '%' '%' '%'
                  'c' 'c' '%' '%' '%' '%' 'd' 'd'
\a\b\c\nd\f    ⟶ '\' 'a' 0x07 '\' 'c' 0x0a 'd' 0x0c
\a\b\c\\nd\f   ⟶ '\' 'a' 0x07 '\' 'c' '\' 'n' 'd' 0x0c
```

Exercice 5

Ajouter à l'exercice précédent le remplacement des séquences "$\backslash nnn$", "$\backslash nn$" et "$\backslash n$" (où n est un chiffre octal) par le caractère dont les codes octals sont respectivement "nnn", "nn" et "n".

exemples:
```
aa\000bb\70cc\5dd ⟶ 'a' 'a' 0x00 'b' 'b' '8' 'c' 'c' 0x05 'd' 'd'
aa\0010bb\\70cc   ⟶ 'a' 'a' 0x01 '0' 'b' 'b' '\' '7' '0' 'c' 'c'
aa\48bb\049cc     ⟶ 'a' 'a' 0x04 '8' 'b' 'b' 0x04 '9' 'c' 'c'
```

3 Analyse grammaticale

3.1 Fonction

Fonction générale

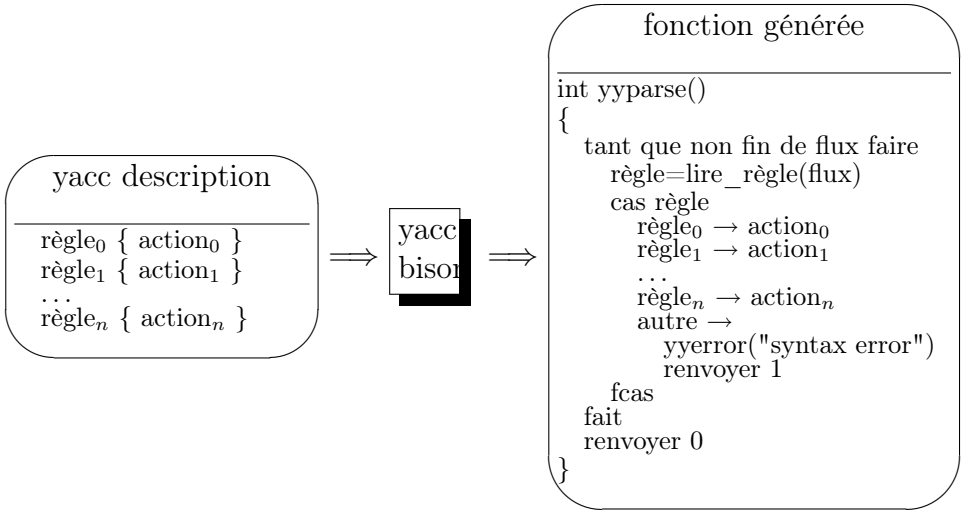
En fonction d'un ensemble de (règle de BNF,traitement) génère un programme qui:

1. lit un flux d'entrée
2. si une règle de BNF est reconnue, effectue le traitement associé
3. si aucune règle n'est reconnue, génère un message d'erreur

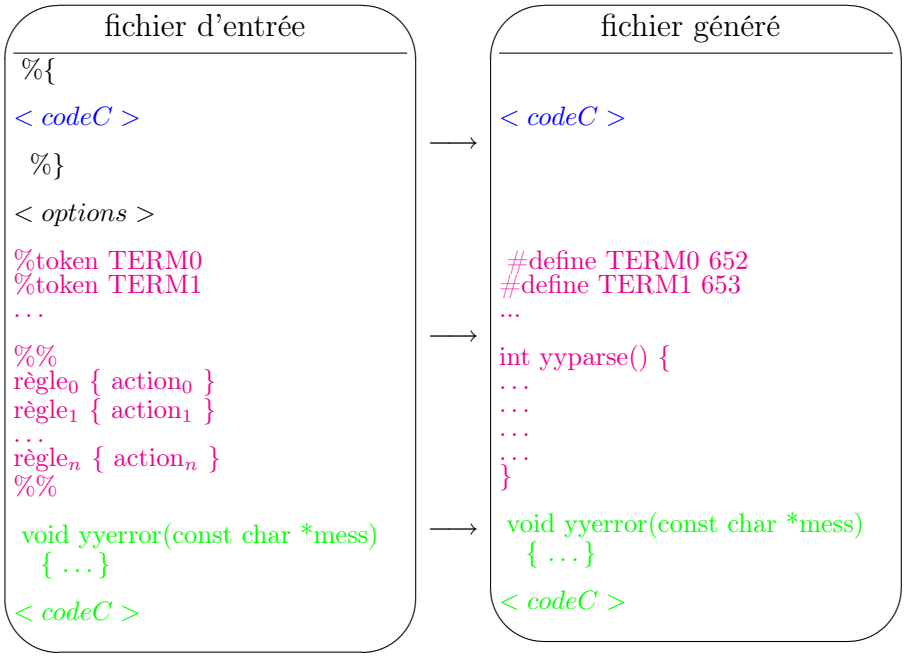
Exemples:

- Un fichier de configuration:
  config := config variable |
  variable := IDENT EQUAL STRING
- Un compilateur C

3.2 Principe de yacc/bison



3.3 Format de yacc/bison



3.4 Syntaxe

3.4.1 Syntaxe des règles

Format général

```
< nom regle >
   : < nom_regle_ou_terminal > ...
   | < nom_regle_ou_terminal > ...
   ...
   | < nom_regle_ou_terminal > ...
;
```

Exemple

```
 /* définition des terminaux */
%token ENTIER

/* règle BNF principale */
%start debut

%%
debut: liste_d_entiers ;
liste_entiers
   : liste_entiers ENTIER
   | ENTIER
;
%%
```

3.4.2 Syntaxe des actions

Format général

r : rt_1 rt_2 ... rt_n { < *codeC* > } ...

- tout le langage C ou C++.
- Les éléments d'une règle ont une valeur associée, celle ci peut être référencée par l'opérateur $.
  $i la valeur associée à rt_i (lecture)
  $$ la valeur associée à r (écriture)
- le rôle de l'action est en général de calculer $$ en fonction des $i.
- Les valeurs associées sont par défaut des "int".
- L'action par défaut est: $$=$1
- L'action { } permet de désactiver l'action par défaut.

Exemple

```
/* définition des terminaux */
%token ENTIER

/* règle BNF principale */
%start debut

%%
debut: liste_entiers          { printf("total=%d\n",$1); } ;

liste_entiers
   : liste_entiers ENTIER     { $$=$1+$2; }
   | ENTIER                   { $$=$1; }
;
%%
```

3.5 Lecture du flux

flux $(t_0, v_0)(t_1, v_1)...(t_n, v_n)$

acquisition Chaque fois que Yacc/Bison a besoin d'un (t_i, v_i) il appelle la fonction `"int yylex()"`.

- elle renvoie l'identifiant du terminal t_i
- elle met dans la variable globale `yylval` la valeur associée v_i.

Le type par défault de `yylval` est un entier.

Exemple en utilisant lex

```
%%
   /* un entier */
[0-9+] { yylval=atoi(yytext); return ENTIER; }
   /* saute les blancs, les tabs, les LFs */
[\t\n] ;
   /* renvoie un terminal inconnu -> yyerror() */
. { return TK_ERROR; }
```

Remarque Il n'y a aucune obligation d'utiliser lex/flex, `"int yylex()"` peut être écrite à la main.

3.6 Extension

3.6.1 Terminal implicite

Dans une règle: un caractère entre quote est considéré comme un terminal. Le code du caractère est alors le code du terminal.

Yacc

```
%token ENTIER
...
%%
somme
   : somme '+' ENTIER
   |
;
...
```

Lex

```
%%
\+ { return *yytext; } /* ou return '+'; */
...
```

3.6.2 Changer le type des valeurs associées et de yylval

1. Définition du type dans le préambule

```
%union {
   double    reel;
   char*     str;
   int       entier;
}
```

2. Associer un champ à chaque terminal et règle utilisés dans les actions

```
%token<entier>    NB0
%token<reel>      NB1
%type<reel>       somme
%type<str>        chaine
```

3. Utiliser l'opérateur $ dans les actions, yacc/bison ajoutera automatiquement le nom du champ.

```
%%
somme
   : somme '+' NB0  { $$=$1+$3; }
   | somme '+' NB1  { $$=$1+$3; }
   | chaine         { $$=atoi($1); }
;
...
```

4. Utiliser explicitement le champ dans le lex/flex.

```
%%
[0-9]+          { yylval.entier=atoi(yytext); return NB0; }
[0-9]+\.[0-9]*  { yylval.reel   =atof(yytext); return NB1; }
...
```

Attention: yyparse manipule des piles de yylval $\Longrightarrow$ évitez:

```
%union {
   double reel_set[1000];
   ...
}
```

3.6.3 Mixer lex/yacc

par inclusion

exemple.y	exemple.l	compilation
%{ ... %} ... %% ... %% #include "lex.yy.c" int yyerror(char*s)	%{ ... %} ... %% ... %% ...	bash> ls exemple.l exemple.y bash> flex exemple.l bash> ls exemple.l exemple.y lex.yy.c bash> bison exemple.y bash> ls exemple.l exemple.tab.c exemple.y lex.yy.c bash> cc exemple.tab.c -lfl bash>

par compilation séparée

exemple.y	exemple.l	compilation

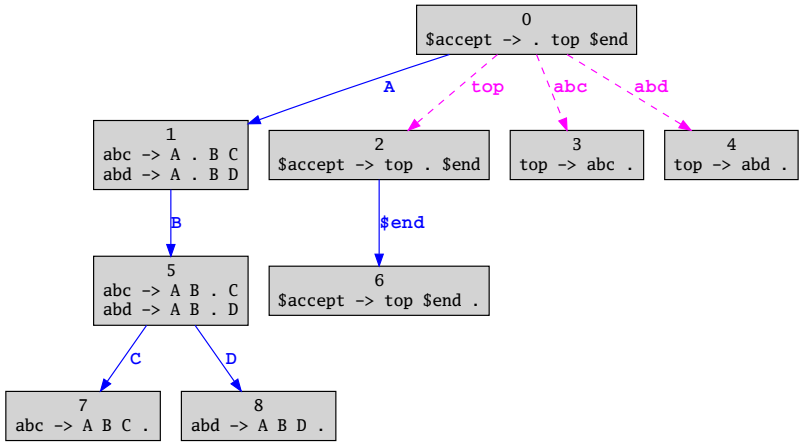
%{
...
%}
...
%%
...
%%
...
	%{
	...
	#include "example.tab.h"
	...
	%}
	...
	%%
	...
	%%
	...
	int yyerror(char*s)
	  ...
	...
		bash> ls
		exemple.l exemple.y
		bash> flex exemple.l
		bash> ls
		exemple.l exemple.y lex.yy.c
		bash> bison -d exemple.y
		bash> ls
		exemple.l exemple.tab.c
		exemple.tab.h exemple.y
		lex.yy.c
		bash> cc exemple.tab.c \
		  lex.yy.c -lfl
		bash>

3.7 Automate shift/reduce

3.7.1 Exemple

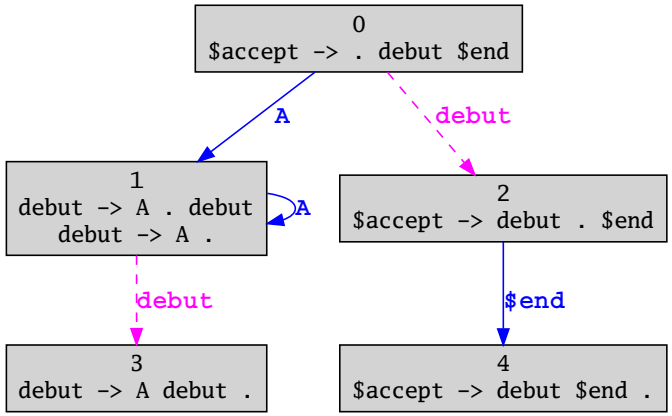
```
debut: abc | abd ;
abc  : A B C ;
abd  : A B D ;
```

```
state 5
    3 abc: A . B . C
    4 abd: A . B . D

    C  shift, go to state 7
    D  shift, go to state 8

state 8
    3 abc: A . B . C

    $default, reduce rule 3
```



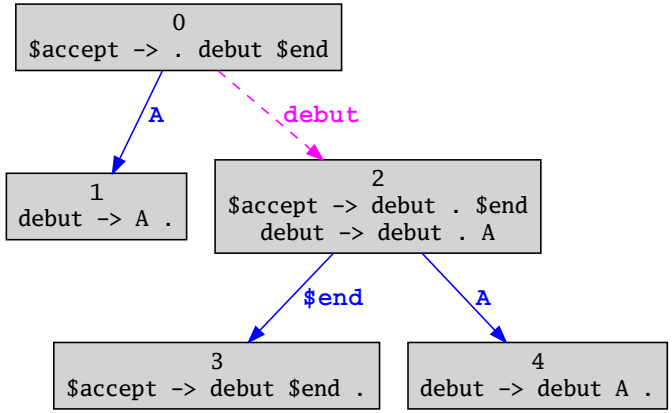
3.7.2 Attention à la pile

```
debut
   : A debut
   | A
;
```




```
state 1

   1 debut: A . debut

   2 debut: A .

   A  shift, go to state 1

   $default  reduce rule 2
```

```
debut
   : debut A
   | A
;
```

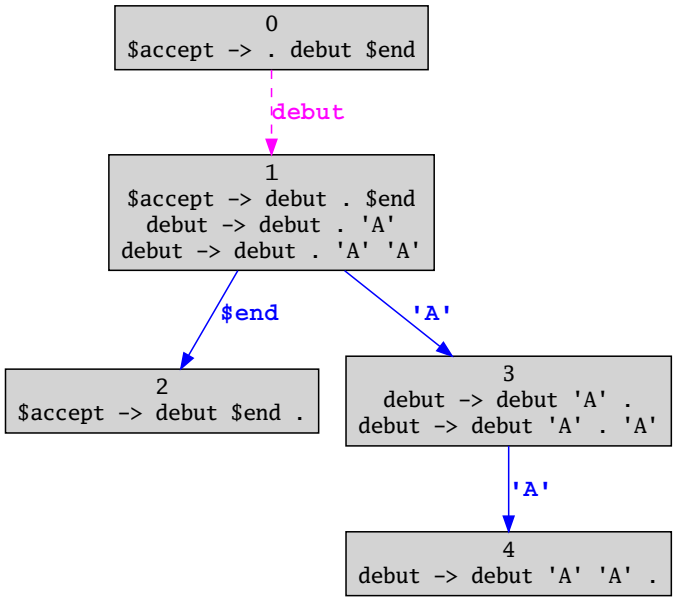



```
state 2

   0  $accept: debut . $end

   2  debut : debut . A

   $end shift, go to state 3

   A    shift, go to state 4
```

3.7.3 Conflits

3.7.3.1 shift/reduce

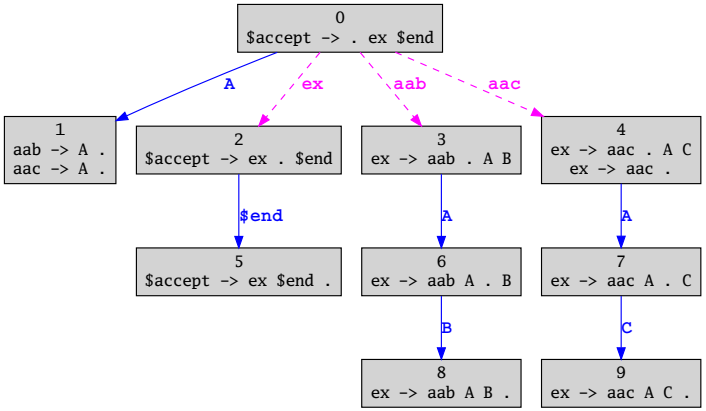
```
debut
   : debut 'A'
   | debut 'A' 'A'
   |
;
```




```
state 3

   1 debut: debut 'A' .

   2 debut: debut 'A' . 'A'


   'A'  shift, go to state 4


   'A'          [reduce rule 1]

   $default  reduce rule 1
```

3.7.3.2 reduce/reduce

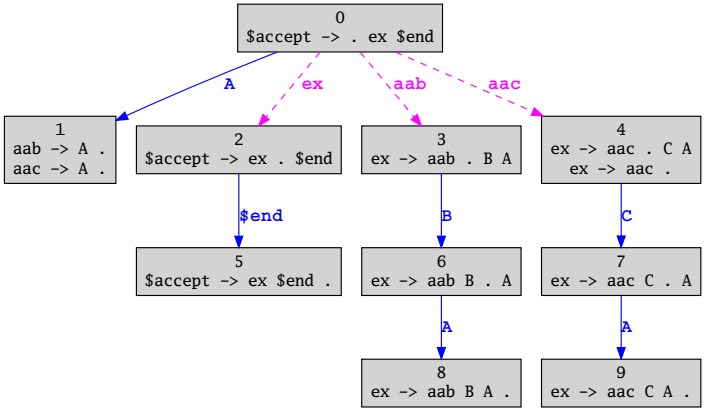
```
top
   : aab A B
   | aac A C
;
aab : A ;
aac : A ;
```




```
state 1

   4 aab: A .
   5 aac: A .

   $end      reduce rule 5
   A         reduce rule 4
   A         [reduce rule 5]
   $default  reduce rule 4
```

3.7.3.3 exemple sans conflit

```
top
   : aab B A
   | aac C A
;
aab : A ;
aac : A ;
```




```
state 1

   4 aab: A .
   5 aac: A .

   B         reduce rule 4
   $default  reduce rule 5
```

3.8 Exemple/Exercice

Ennoncé

Ecrire un programme qui évalue l'expression booléenne passée dans son premier argument.
L'expression est complètement parenthésée, les terminaux sont:

VRAI FAUX	constantes
& | +	opérateurs binaires: et, ou, ou exclusif
↑	opérateurs unaire: non
( )	parenthèses ouvrante et fermante

Quelques exemples:

VRAI
↑FAUX
((VRAI +FAUX) & ↑(FAUX|VRAI))

Analyse lexicale

```
%{ /* Fichier: expr.l */
#define YY_INPUT(buf,res,nb) \
    if (stream && *stream) {\
            *buf=*stream++; res=1;\
    } else res=0;
static char* stream;
%}

%option noyywrap
...

%%
...

%%

int yyerror(char* m) {
  fprintf(stderr,"%s (near %s)\n", m,yytext);
```

```
  exit(1);
}
```

Analyse grammaticale

```
%{
/* Fichier: expr.y */
#include <stdio.h>
static int result;
%}

%token TK_ERROR
%token ...
...

%%
...


%%

#include "lex.yy.c"

int main(int nb, char** arg) {
  int ret;
  if (nb!=2) {
    fprintf(stderr,"usage: %s expr\n",arg[0]);
    return 1;
  }
  stream=argv[1];
  if ( (ret=yyparse)!=0 )
    printf("%s\n",result?"VRAI":"FAUX");
  return ret;
}
```

Compilation & exécution

```
bash> ls
expr.l expr.y
bash> flex expr.l
bash> bison expr.y
bash> ls
expr.l  expr.tab.c  expr.y  lex.yy.c
bash> cc expr.tab.c
bash> ./a.out "VRAI + FAUX"
    syntax error (near +)
bash> ./a.out "(VRAI + FAUX)"
    VRAI
bash>
```

3.9 Exercices

Exercice 1

Écrire un compilateur qui lit sur le fichier standard d'entrée des listes d'entiers et qui écrit sur le fichier standard de sortie la somme de chaque liste. On écrira un compilateur pour chacune des définitions de liste d'entiers données ci-dessous:

1. Une liste est une suite de nombres séparés par un espace, la fin de liste étant donnée par le caractère ';'. Un espace est une suite d'au moins 1 caractère parmi les caractères blanc, tabulation et fin de ligne.

2. Une liste est une suite de nombres séparés par un espace sur la même ligne. Un espace est une suite d'au moins 1 caractère parmi les caractères blanc et tabulation.

3. Une liste est une suite de nombres séparés par un espace, la fin de liste étant donnée par les caractères ';' ou fin de ligne. Un espace

est une suite d'au moins 1 caractère parmi les caractères blanc et tabulation.

Exercice 2

Écrire un compilateur qui lit sur le fichier standard d'entrée des listes d'entiers et qui écrit sur le fichier standard de sortie la moyenne de chaque liste.
Une liste est une suite de nombres séparés par un espace, la fin de liste étant donnée par le caractère ';'. Un espace est une suite d'au moins 1 caractère parmi les caractères blanc, tabulation et fin de ligne.

Exercice 3

Écrire un compilateur qui prend en entrée 1 liste d'entiers et qui écrit sur le fichier standard de sortie la somme des entiers. On écrira les compilateurs qui lisent sur les flux suivants:

1. La liste est lue sur le fichier standard d'entrée.
2. La liste est lue dans un fichier passé en argument.
3. la liste est le premier argument.

Exercice 4

Écrivez l'analyseur grammatical qui reconnaît un arbre binaire d'entiers.

Exercice 5

Écrivez l'analyseur grammatical qui reconnaît un arbre n-aire d'entiers.

Exercice 6

Écrivez l'analyseur grammatical qui reconnaît un graphe orienté valué.

Exercice 7

Écrivez un compilateur qui lit le fichier d'entrée standard et indique si il correspond au format "xml" défini ci-dessous.

- le fichier ne contient que les balises ouvrantes: $< l0 >$, $< l1 >$ et $< l2 >$.
- le fichier ne contient que les balises fermantes: $< /l0 >$, $< /l1 >$ et $< /l2 >$.
- les balises peuvent être séparées par des espaces. Un espace est une suite d'au moins 1 caractère parmi les caractères blanc, tabulation et fin de ligne.
- toute balise ouvrante doit être fermée.
- entre une balise ouvrante et sa fermante, il ne doit pas y avoir de balises ouvrantes non fermées ni de balises fermantes non ouvertes.

Si l'entrée est correcte, le compilateur n'écrit rien, sinon il écrit le numéro de ligne ou se trouve le problème et la balise fautive.

Exercice 8

Même exercice que le précédent (3.7) mais les balises ne sont pas prédéfinies. Une balise est une lettre suivi de zéro ou plusieurs caractères alphanumériques.

Exercice 9

Même exercice que le précédent (3.7) mais les balises $l0$ ne doivent imbriquer que des balises $l1$ qui ne doivent imbriquer que des balises $l2$. Ces dernières ne pouvant pas contenir de balises.

Exercice 10

Sur la grammaire ci-contre, yacc indique un conflit shift/reduce. Indiquez pour les entrées ci-dessous les interprétations possibles et celles que yacc choisit.

```
"A"
"A A"
```

```
%token A
%%
prog: x | y | z;
x: A  ;
y: A A ;
z: x A ;
```

Exercice 11

Sur la grammaire ci-contre, yacc indique un conflit shift/reduce.

1. Indiquez pour chacune des entrées suivantes les interprétations possibles et celle que yacc choisit:
   `"IF I ELSE IF I"`,
   `"IF IF I ELSE I"`,
   `"IF IF I ELSE IF IF I ELSE I"`.

2. Proposez une syntaxe permettant de lever ce conflit.

```
%token IF I ELSE

%%
i : I
  | IF i
  | IF i ELSE i
;
```

Exercice 12

Dans les grammaires suivantes, les terminaux sont A, B, C, X et Y. Complètez le tableau en indiquant pour chaque entrée toutes les réductions possibles. Indiquez si la grammaire est conflictuelle et dans l'affirmative décrivez le conflit.

grammaire	entrée

`prog: xx | yy ;`
`xx: x X ;`
`yy: y Y ;`
`x: A | B ;`
`y: A | C ;`
	AX → x:A ; xx:x.X ; prog:xx
	AY →
	BX →
	CY →

`prog: xx | yy ;`
`xx: x X ;`
`yy: y X ;`
`x: A | B ;`
`y: A | C ;`
	AX →
	→
	BX →
	CX → y:C ; yy:y.X ; prog:yy

`prog: xx | yy ;`
`xx: x X X;`
`yy: y X Y;`
`x: A | B ;`
`y: A | C ;`
	AXX →
	→
	BXX → x:B ; ; xx:x.X.X ; prog:xx
	CXX → y:C ; ; yy:y.X.X ; prog:yy
	AXY →
	→
	BXY → x:B ; ; xx:x.X.Y ; prog:xx
	CXY → y:C ; ; yy:y.X.Y ; prog:yy

4 Travaux pratiques

4.1 Analyse lexicale

Dans les exercices qui suivent on appelle "chaîne de caractères" toute suite de caractères comprise entre double quotes et ne contenant ni LF (retour à la ligne) ni double quotes. Les doubles quotes servent de délimiteurs et ne font pas partie de la chaîne.

Exercice 1

Les premiers «Télétypes» utilisaient deux caractères pour débuter une nouvelle ligne, un pour ramener le chariot à la position d'origine (retour chariot ["Carriage Return"], CR), et un autre pour faire avancer le papier (saut-de-ligne ["Line Feed"], LF). Les différents concepteurs de systèmes d'exploitation ont hérité de cette tradition tout en l'adaptant au passage. Le monde Unix choisit d'utiliser LF (tout seul) comme fin-de-ligne, Apple sur ses vieux MAC choisit d'utiliser CR (tout seul) et Microsoft de conserver la combinaison CR LF. Cela signifie que si vous tentez de passer un fichier d'un système à un autre, vous rencontrerez des problèmes avec les coupures de lignes. Écrivez les programmes suivants qui lisent le fichier standard d'entrée et écrivent sur le fichier standard de sortie.

dosTOunix convertit du format dos au format unix en suprimant les CR excédentaires dans les combinaisons CR/LF

macTOunix convertit du format mac au format unix en remplaçant les CR par des LF

unixTOdos convertit du format unix au format dos en ajoutant des CR devant les LF.

Pour valider vos programmes vous avez dans /pub/ia/ico/dosMacUnix les fichiers data.win, data.unix et data.mac et visualiser les entrées et sorties de vos programmes par:

```
$ od -c /pub/ia/lex-yacc/data.win
00000 a b c \r \n a b c \r \n
$ dosTOunix < /pub/ia/lex-yacc/data.win | od -c
00000 a b c \n a b c \n
$
```

Exercice 2

Écrivez un filtre:

a) qui s'arrête sur une chaîne de caractères incomplète

b) qui extrait et imprime les chaînes de caractères (une par ligne)

Exercice 3

Complèter le filtre précédent en faisant précéder les chaînes reconnues par leur numéro de ligne dans le fichier.

Exercice 4

Complèter l'exercice précèdent mais en considérant que le fichier d'entrée contient des commentaires. Un commentaire commence par un dièse (#) et coure jusqu'à la fin de la ligne. Dans les commentaires les chaînes n'ont pas à être considérées. Dans les chaînes les commentaires n'ont pas à être considérés.

Exercice 5

Ecrire un programme qui lit le fichier passé en argument et écrit sur le fichier standard de sortie, le fichier d'entrée moins ces commentaires. Les commentaires sont ceux du C ansi, ils commencent par `"/*"`, se terminent par `"*/"` et peuvent s'étendre sur plusieurs lignes.
On le testera en particulier sur les entrées suivantes:

- `a/*b*/c/*b*/a`
- `a/*b/*c*/a`

Exercice 6

Réaliser le filtre défini dans l'exercice 4 à la page 6. On pourra le tester entre autre avec les commandes:

```
./a.out '\a\b\c\nd\f' | od -x1
./a.out '\a\b\c\nd\f' | od -t x1
./a.out '\a\b\c\\nd\f' | od -t x1
```

Exercice 7

Réaliser le filtre défini dans l'exercice 5 à la page 6. On pourra le tester entre autre avec les commandes:

```
./a.out 'aa\000bb\70cc\5dd' | od -t x1
./a.out 'aa\0010bb\\70cc' | od -t x1
./a.out 'aa\48bb\049cc' | od -t x1
```

4.2 Analyse grammaticale (1)

L'analyseur syntaxique "/pub/ia/ico/cal.y" ci-dessous sert de base à cette série d'exercices.

```
// Fichier cal.y
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token NUM
%%
expr
  : NUM
  | expr '+' expr { $$=$1+$3;
     printf("%d = %d + %d\n",
             $$,  $1,  $2); }
  | expr '*' expr { $$=$1*$3;
     printf("%d = %d * %d\n",
             $$,  $1,  $2); }
;
%%
#include "yy.lex.c"
int yyerror(const char*mess)
{
  fprintf(stderr,
    "FATAL:%s (near %s)\n",
    mess,yytext);
  exit(1);
}
int main()
{
  yyparse(); return 0;
}
```

Exercice 1

Indiquez ce que fait l'analyseur syntaxique cal.y. Écrivez ce fichier cal.y et le fichier cal.l l'analyseur lexical lui correspondant.

Exercice 2

Générer le compilateur, vous pouvez tapez la commande:

```
$ bison cal.y -g -v && flex cal.l && gcc cal.tab.c
```

L'option -v indique à bison de générer un dump de l'automate dans le fichier "cal.output". L'option -g indique à bison de générer un schéma de l'automate dans le fichier "cal.dot". La commande

```
$ dot -Tpdf cal.dot > cal.pdf
```
permet d'obtenir un PDF de ce schéma.

1. Expliquez les conflits "shift" "reduce".
2. Essayez l'entrée "1+2*3+4" et d'autres entrées.
   a) Comment sont placées les parenthèses?
   b) Quand a lieu le premier "reduce" ?
3. Ajoutez dans le fichier "cal.y" après la ligne "`%token NUM`", la ligne "`%right '+' '*'`".
   Cette primitive indique que les opérateurs sont associatifs droits ($a+b+c = a+\{b+c\}$). Régénérez le compilateur et refaites quelques expérimentations.
   a) Pourquoi les conflits ont disparus ?
   b) Quand a lieu le premier "reduce" ?
4. Changer dans le fichier "cal.y" le right par left. Cette primitive indique que les opérateurs sont associatifs gauche ($a+b+c = \{a+b\}+c$). Régénérez le compilateur et refaites quelques expérimentations.
   a) Pourquoi les conflits ont disparus ?
   b) Quand a lieu le premier "reduce" ?
   c) Soit un opérateur binaire associatif. Dans une grammaire faut il mieux le déclarer associatif gauche ou associatif droit ?
5. Changer dans le fichier "cal.y" la ligne "`%left '+' '*'`" par "`%left '+'`" et "`%left '*'` sur 2 lignes.

Exercice 3

Modifiez légèrement le cal.y et le cal.l de l'exercice précédent pour supporter les expressions parenthésées comme: "1+2*3+4", "(1+2)*(3+4)", "((1+(3))*(3+(4)))".

Exercice 4

En repartant du cal.y initial, générez un compilateur d'expressions non parenthésées sans conflit et sans utiliser les primitives "`%right` et "`%left`".
On introduira la règle "produit"

Exercice 5

Modifiez le cal.y de l'exercice précédent pour supporter les expressions parenthésées comme: "1+2*3+4", "(1+2)*(3+4)", "((1+(3))*(3+(4)))".

4.3 Analyse grammaticale (2)

L'objectif de ce TP est de implanter un petit langage graphique produisant des dessins au format PostScript. `X` et `Y` étant des entiers, les commandes de base sont:

- `Aller X Y` : qui positionne le curseur au point (X, Y).
- `Tracer X Y` : qui trace une ligne du curseur au point (X, Y), et positionne le curseur au point (X, Y).
- Quitter : qui termine le programme.
- Un commentaire commence par un dièse (#) et coure jusqu'à la fin de la ligne.

Un script est une suite de commandes terminée par `Quitter`. La fin d'une commande est indiquée par un ';' ou un saut de ligne. Une ligne vide ou un ';' tout seul sont ignorés. Pour traduire un script en PostScript vous devez :

- Afficher l'entête suivante:
  ```
  /Times-Roman 48 selectfont
  0 0 moveto
  ```
- traduire les commandes Aller et Tracer par les instructions PostScript
  `"X Y moveto"` pour se positionner en `X,Y`,
  `"X Y lineto"` pour tracer de la position courante à `X,Y`.
- afficher le pied de page suivant :
  ```
  stroke
  showpage
  ```

Exercice 1

Écrivez un analyseur syntaxique qui vérifie qu'un script est correct. L'analyseur syntaxique lit le script sur le flux standard d'entrée, ses fonctions sont:

- En cas d'erreur, avant de s'arrêter, il écrit un message d'erreur sur le flux standard d'erreur.
- En cas de succès, il écrit "correct" sur le flux standard d'erreur.

Exercice 2

Complétez l'analyseur pour qu'il génère le fichier PS sur le flux standard de sortie.

Exercice 3

Modifiez l'analyseur pour que les commandes "A", "a" et "aller" soient des synonymes de "Aller", pour que les commandes "T", "t" et "tracer" soient des synonymes de "Tracer", pour que la fin de fichier ainsi que les commandes "Q", "q" et "quitter" soient des synonymes de "Quitter",

Exercice 4

Ajoutez la commande

```
Ecrire "un texte"
```

qui affiche `"un texte"` à la position courante. La commande PS pour écrire ce texte est:

```
(le texte) show
```

Exercice 5

Modifiez la commande `Ecrire` pour qu'elle accepte plusieurs chaînes de caractères et qu'elle les écrive les unes sous les autres. Par exemple la commande

```
Ecrire "aaa" "bbbb" "ccccc"
```

écrit à la position courante

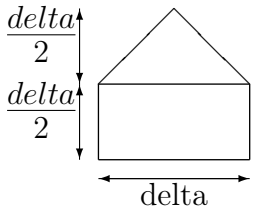
```
aaa
bbbb
ccccc
```

Exercice 6

Ajoutez la commande

```
Maison delta
```

qui affiche le dessin si contre. Le point du bas à gauche est la position courante. La position courante n'étant pas modifiée.



Exercice 7

Complétez la commande Maison avec un paramètre entier n optionnel

```
Maison delta [n]
```

qui affiche n maisons de taille delta cote à cote.
Si n est omis une seule maison est dessinée.
Le point du bas à gauche est la position courante. La position courante n'étant pas modifiée.

Exercice 8

Complétez la commande Maison avec des options optionnelles

```
Maison delta [n] [opt,] [opt,] [opt]
```

qui affiche n maisons de taille delta cote à cote.
Les options sont:

S n les maisons sont espacées de n (défaut 0).

E n les maisons ont n étages (défaut 0).

F n les maisons ont n fenêtres par étage (defaut 0).

C les maisons ont 1 cheminée.

P les maisons ont 1 porte.

D les fenêtres des maisons sont à deux ventaux.

L'ordre des options est indifférents. Si une option est présente plusieurs fois c'est la dernière valeur qui est utilisée.

Exercice 9

Modifiez la commande Maison pour qu'une option ne puisse être présente qu'une seule fois.

Exercice 10

Nous allons maintenant implanter un langage de type "LOGO". Dans un tel langage on dessine en déplaçant un crayon. Le crayon peut être baissé ou lever. Quand il est baissé les déplacements tracent un trait. A tout nomment on maintient une direction pour le crayon. Cette direction peut être modifier. Les commande de base sont :

Init place le crayon au point courant, la direction vers le haut, et en position levée.

Init X Y place le crayon au point (X,Y), direction vers le haut, et en position levée.

Baisser baisse le crayon.

Lever lève le crayon.

Avancer d avance de d dans la direction courante.

Gauche a tourne à gauche de a degrés

Droite a tourne à droite de a degrés

Pour maintenir l'état du crayon on pourra utiliser le type et la variable globale suivant :

```
typedef struct _etat{
   int x;
   int y;
   double angle ;
   int leve; /* 0 si en train d'ecrire */
} etat;
```

```
etat crayon={0,0,0,0};
```

Exercice 11

Ajoutez la gestion des variables pour les valeurs numériques. L'identifiant d'une variable est composé de 'V' suivi de 2 chiffres décimaux (example: V00, V01, ..., V99). Une valeur est assignée à une variable par la commande `set`:

```
Set V10 670
```

Dans le langage, toute valeur numérique peut être remplacée par une variable comme dans les examples suivants:

```
Trace V30 670
Init 670 V43
Aller V00 V24
```

A l'initialisation les variables valent zéro.

Exercice 12

Ajoutez la possibilité de faire de l'arithmétique sur toutes les valeurs numériques. On se limitera à l'addition, la soustraction et à des expressions complètement parenthésées. Par exemple:

```
Trace V30+10-V12 (1-(-12+670)
Init V43-(V11-1) (0)
Aller V00 V24
```

5 Projet

5.1 Description de la machine PICO

La machine PICO exécute des programmes en assembleur PICODE. PICO a un espace mémoire sur 15 bits (32 ko). Elle possède 16 registres R_i ($i \in [0:15]$) de 32 bits. La mémoire est gérée en petit indien (octet de poids faible dans les adresses faibles). Les entiers sont codés en complément à deux.

bits du 1^{er} octet					mode	information
7	6	5	4	3-0		
0	0	0	d	0	immédiat	la donnée est sur 1, 2 ou 4 octets suivant le paramètre SZ de l'instruction.
0	0	1	d	0	direct mémoire	Les 2 octets suivants donnent l'adresse.
0	1	0	d	0	indirect mémoire	Les 2 octets suivants donnent l'adresse.
1	0	0	d	#	direct registre	# est le numéro du registre, pas d'octet supplémentaire.
1	0	1	d	#	indirect registre	# est le numéro du registre, pas d'octet supplémentaire.
1	1	0	d	#	indirect registre post-incrémenté	# est le numéro du registre, pas d'octet supplémentaire.
1	1	1	d	#	indirect registre pré-incrémenté	# est le numéro du registre, pas d'octet supplémentaire.

d=0: ce n'est pas la dernière "effective addresse" de l'instruction.
d=1: c'est la dernière "effective addresse" de l'instruction.

Table 1: Types et codages des "Effective Adresses" de l'assembleur PICODE.

fonction	OP	SZ	N	D	S_0	S_1	S_i	S_{N-1}	N2
$D = \sum_0^{N-1} S_i$	1	M	M	M	M	M	*	*	
$D = S_0 - \sum_1^{N-1} S_i$	2	M	M	M	M	M	*	*	
$D = \prod_0^{N-1} S_i$	3	M	M	M	M	M	*	*	
$D = S_0$	4	M	M	M	M				
branchement	5	0	M						
branch. à N2 si $D = S_0$	6	M	M	M	M				M
branch. à N2 si $D < S_0$	7	M	M	M	M				M
branch. à N2 si $D \leq S_0$	8	M	M	M	M				M
branch. à N2 & empile N	9	0	M						M
dépile x & branch. à x	10	0							
input into D	11	M	M	M					
output from D	12	M	M	M					
stop	13	0							

M: champ obligatoire; *: champ facultatif; ni M ni *: champ interdit

Table 2: Instructions et codages des instruction de l'assembleur PICODE.

Une instruction est composée d'une suite d'octets dont seul le premier est obligatoire.

OPSZ Sur un octet.

OP Le code opération sur 6 bits (bits 7 à 2).

SZ La taille sur la quelle porte l'opération sur 2 bits (bits 1 à 0) avec 00:non utilisée, 01:1 octet, 10:2 octets, 11:4 octets.

N L'adresse de l'instruction suivante sur 2 octets.

D L'opérande destination sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

S_0 La première opérande source sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

S_i La $i^{ième}$ opérande source sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

S_{N-1} La dernière opérande source sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

N2 Une deuxième adresse de branchement sur 2 octets.

Le format et les instructions supportées sont donnés sur la table 2.

Une **Effective Adresse** ommence par un octet. Suivant la valeur de cet octet, il y a 0, 1, 2 ou 4 octets supplémentaires. La table 1 spécifie leurs différents types ainsi que leurs codages.

5.2 Description de la VM PICO

L'exécutable vmpico est un simulateur de la machine pico. Il a un argument optionnel (défaut picode) qui est un fichier binaire d'au maximum 2^{15} octets de PICODE.
Il lit ce fichier et lance l'exécution du PICODE en commençant à l'adresse 0. Si le PICODE fait des lectures, elles sont faites sur le le flux standard d'entrée, les écritures sont faites sur le le flux standard de sortie. D'autres options sont disponibles et peuvent être affichées avec l'option -h.

5.3 Sujet

Réalisez l'assembleur aspico de PICODE pour la machine PICO. aspico lit un fichier assembleur passé en argument et génére un fichier PICODE exécutable par la vmpico. Ses options sont:

-t affiche sur le flux standard de sortie la table des symboles puis se termine.

-d affiche sur le flux standard de sortie un dump humainement lisible du PICODE généré.

-o file le fichier de PICODE généré. Si cette option est absente le fichier est par défaut picode.

Ses caractéristiques générales sont: 1) il doit tourner sur toute machine Unix après régénération; 2) le programme vérifie que les instructions sont valides; 3) le programme vérifie qu'il n'y a pas de conflits.

5.4 Syntaxe de l'assembleur

La syntaxe de l'assembleur est composée de constantes (nommée <cst> par la suite), de définitions de labels, de primitives et d'instructions.

5.4.1 Commentaire

Le caractère '#' marque le début d'un commentaire, il va jusqu'au bout de la ligne.

5.4.2 Constante

Constantes numériques terminales Ce sont des caractères, des nombres décimals et hexadécimals et elles sont définies comme en C. Voici des exemples: 7, 0xF4, 'A', -12, +122, -'c', +0xa7.

Constantes label Il sont définis comme un identifiant suivi d'un ':' (sans espace). L'identifiant (sans le ':') devient une constante. Voici des exemples de définitions
de labels correctes: gnu:, bee:, g1n1n1:
de labels incorrectes: gnu :, 3bee:, g$:

Constantes (<cst> dans la suite) Ce sont des expressions arithmétiques éventuellement parenthésées formées avec les opérateurs +, -, *, << et des constantes numériques et/ou des labels.
Voici des exemples de constantes: bee+12, (2*(gnu+-1)*3), a-+40, a - +40

5.4.3 Primitive

Les primitives sont des mots clé commençant par le caractère '.'.

.org <cst> Indique que l'instruction ou la primitive suivante se trouvent à l'adresse <cst>.

.long <cst> Réserve 4 octets et y met la valeur <cst>.

.word <cst> Réserve 2 octets et y met la valeur <cst>.

.byte <cst> Réserve 1 octets et y met la valeur <cst>.

.string "str" Réserve autant d'octets plus 1 que la chaîne de caractères et y met la chaîne suivie d'un ∅.

Il y a des restrictions pour l'utilisation de la primitive ".org <cst>"

- Son <cst> ne doit pas contenir de label.
- Si dans un fichier, on a une primitive ".org <cst1>" et un peu plus loin une primitive ".org <cst2>" alors <cst2> doit être plus grand que <cst1>.
- Le <cst> d'une primitive ".org" ne doit pas positionner sur une partie de la mêmoire déjà initialisée.

5.4.4 Instruction avec opérandes

Le format général d'une instruction avec opérandes est le suivant.

OPSZ une instruction commence par un mnémotechnique.

DEST le mnémotechnique est suivi de l'opérande destination,

, SRC$_i$ puis des opérandes sources (chaque opérande source est précédée d'une virgule),

AIS puis de l'adresse de l'instruction suivante. Celle-ci est facultative. Son absence indique que l'exécution continue en séquence.

Les mnémotechnique des instructions (OPSZ) sont avec sz indiquant la taille (b pour 1, w pour 2, ou l pour 4): add$_{sz}$, sub$_{sz}$, mul$_{sz}$, mov$_{sz}$, if$_{sz}$, in$_{sz}$, out$_{sz}$.

Le format des opérandes sources et destinations (DEST, SRC$_i$) est: $<cst> pour de l'immédiat, <cst> pour du direct memoire, (<cst>) pour de l'indirect mémoire, %i pour du direct registre sur R$_i$, (%i) pour de l'indirect registre sur R$_i$, ++(%i) pour de l'indirect registre pré-incrémenté sur R$_i$, (%i)++ pour de l'indirect registre post-incrémenté sur R$_i$.

Le format de l'adresse de l'instruction suivante (AIS) est: @<cst>.

Les formats de l'instruction "if$_{sz}$" sont
  if$_{sz}$ OP1 <oc> OP2 <cst-true>
ou
  if$_{sz}$ OP1 <oc> OP2 <cst-true> else <cst-false>
OP1 et OP2 suivent le format standard des opérandes, <oc> est un des opérateurs de comparaison suivants: '==', '!=', '<', '>', '<=', '>=', '=<', '=>'. <cst-true> est l'adresse de branchement si la condition est vrai. <cst-false> est l'adresse de branchement si la condition est fausse. L'instruction "if$_{sz}$" ne peut pas avoir d'AIS.

5.4.5 Instruction sans opérandes

jmp <cst-bra> Branche à l'adresse <cst-bra>.

call <cst-bra> Branche à l'adresse <cst-bra> et empile l'adresse de l'instruction suivante.

call <cst-bra> AIS Branche à l'adresse <cst-bra> et empile l'adresse AIS.

ret Dépile une adresse et s'y branche.

stop Arrête l'exécution.

5.4.6 Exemples

Voici 2 exemples de PGCD:

```
1  ### PGCD 1
2  .org 0
3  jmp main
4  a: .long 0
5  b: .long 0
6  main:
7    inl a
8    inl a+4
9  loop:
10   ifl a==b end
11   ifl a<b  AinfB
12        else BinfA
13 BinfA:
14   subl a,a,b @loop
15 AinfB:
16   subl b,b,a @loop
17 end:
18   outl a
19   stop
```

```
1  ### PGCD 2
2  .org 0
3  jmp 1024
4  pgcd: # a/b/ret=%1/(%2)/%0
5    ifl %1=(%2) pgcd_end
6    ifl %1>(%2) AsupB
7          else BsupA
8  BsupA:
9    subl (%2),(%2),%1 @pgcd
10 AsupB:
11   subl %1,%1,(%2) @pgcd
12 pgcd_end:
13   movl %0,%1
14   ret
15
16 .org 1024
17   inl %1 # a
18   inl 1024-4 # b
19   movw %2,$1024-4
20   call pgcd
21   outl %0
22   stop
```

5.5 Éléments à rendre

Les livrables seront faits à partir d'un dépôt SVN.

Livrable 1: analyseur syntaxique de l'assembleur

Le dépot SVN contient au moins les fichiers asm.y, asm.l, main.c, et makefile.

Le asm.y n'inclue pas le asm.l (compilation séparée). Le makefile est fait manuellement, il génère l'exécutable aspico.

Ce dernier 1) analyse la ligne de commande, 2) indique si la description assembleur est syntaxiquement correcte. Dans le cas où la description est syntaxiquement incorrecte, il indique l'erreur et où elle s'est produite (nom du fichier, numéro de ligne, et éventuellement le token).

La syntaxe proposée est conflictuelle, il faudra la corrigée pour la rendre dérivable (2 très légères modifications).

Livrable 2: analyseur sémantique de l'assembleur

Les bugs éventuels du premier livrable sont corrigés. L'exécutable aspico vérifie la sémantique et l'option -t est opérationnelle.

label adresse
label adresse
...

Si la sémantique est incorrecte, il indique en clair le problème et si c'est possible sa localisation.

Livrable 3: compilateur complet de l'assembleur

De plus on ajoutera à aspico, par ordre de priorité, les fonctionnalités de macro-assembleur:

1. une macro pour faire des alternatives (si COND alors BLOC [sinon BLOC] fsi) de façon légère comme dans un langage de haut niveau.
2. une macro pour faire des boucles "tant que" (tq COND faire BLOC fait) de façon légère comme dans un langage de haut niveau.
3. dans les macros précédentes, les conditions peuvent être des expressions booléennes "(COND1 ou non(COND2 et COND3)) et COND4" avec le ET prioritaire sur le OU.
4. une macro pour faire des boucles "pour" (pour i de d à f pas de p faire BLOC fait) de façon légère comme dans un langage de haut niveau.