Lex & Yacc (Flex & Bison)

– ING1/APP ING1 –

Jonathan Fabrizio
LRDE-EPITA
http://lrde.epita.fr/~jonathan/

EPITA
ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

Lex & Yacc

- Lex :
  - Génère un analyseur lexical
  - Traite les langages de type 3 (réguliers)

  - Flex : Fast lexical analyzer generator

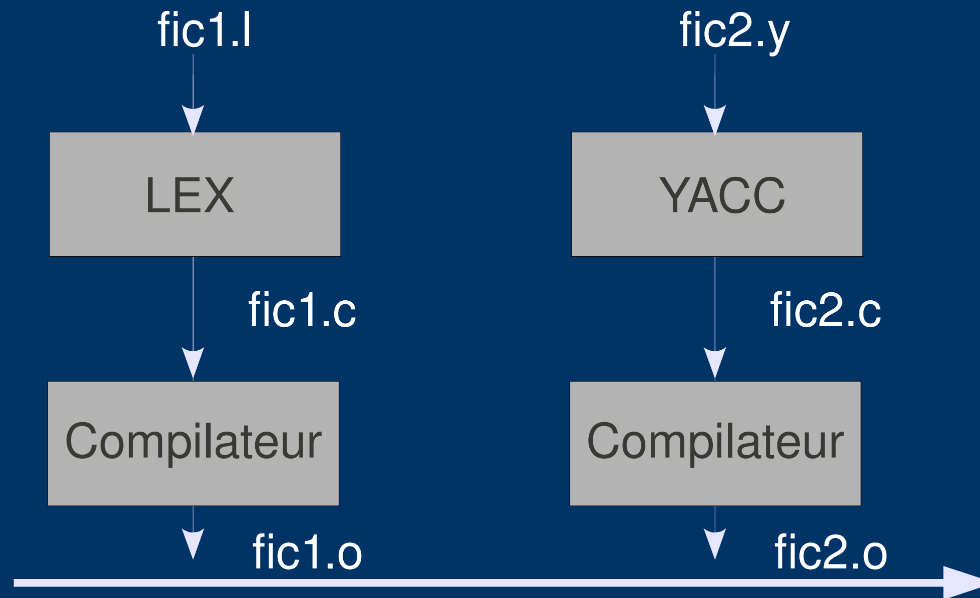
- Yacc
  - Yet another compiler compiler
  - Génère des parsers LALR

  - Bison

Lex&Yacc

- Lex et Yacc sont souvent utilisés ensemble toutefois ils sont indépendants :
  - Lex crée l'analyseur lexical et cet analyseur peut être utilisé sans Yacc (par exemple, si on écrit à la main un analyseur syntaxique LL(1))
  - Yacc crée l'analyseur syntaxique et cet analyseur peut être utilisé sans lex

Lex&Yacc



Bison

- Format du fichier :

prologue : définitions et options
%%
règles
%%
épilogue : code utilisateur

Bison

- <u>Appel :</u>

bison [options] fic.y -o fic.c

retour : le code du compilateur en c prêt à être compilé...

Quelques options possibles :

--xml : sortie en xml

--report=all : génère un rapport complet sur le parser

--graph : sauvegarde du parser sous forme de graph

Note : certaines options peuvent être mises indifféremment dans le prologue ou passées en ligne de commande

Bison

- Sortie :

Bison génère un programme c du parser, il peut être appelé via la fonction :

```
int yyparse();
```

Pour fonctionner correctement, cette fonction a besoin d'une fonction `int yylex();` ainsi que

```
void yyerror(const char *s);
```

Bison

- Gestion d'erreurs :
  - En cas d'erreur la fonction int yyerreur(const char *s) est appelée avec un message d'erreur (en général *syntaxe error*) – c'est à l'utilisateur de définir cette fonction
  - Il est possible d'avoir des messages plus explicites avec l'option %define parse.error verbose dans le prologue
  - Il existe des possibilités pour faire de la reprise sur erreur...
  - yyparse renvoie 0 si le parsing est complet 1 sinon

Bison

- Conflits shift-reduce :
  - Il est possible de spécifier combien de conflits shift/reduce on prévoit pour la grammaire donnée
    option %expect n dans le prologue
    (en cas de conflit, yacc choisi le shift)
  - Dans la pratique :
    %expect 0

Bison

- La Grammaire
  syntaxe :

```
exp
    :exp "+" exp
    | exp "*" exp
    | exp "-" exp
    | exp "/" exp
    ;
```

Bison

- La Grammaire

  syntaxe :

  On insère des actions à accomplir pendant le processus

```
exp
    :exp "+" exp   {printf("Branche +\n");}
    | exp "*" exp {printf("Branche *\n");}
    | exp "-" exp   {printf("Branche -\n");}
    | exp "/" exp {printf("Branche /\n");}
    ;
```

Bison

- La Grammaire

  syntaxe :

```
exp
      :exp "+" exp   {$$ =  $1 + $3;}
      | exp "*" exp {$$ = $1 * $3;}
      | exp "-" exp  {$$ = $1 - $3;}
      | exp "/" exp {$$ = $1 / $3;}
      ;
```

On peut faire apparaître les valeurs

Bison

- La Grammaire
  - Axiome :
    première règle ou %start

Bison

- La Grammaire
  - L'analyseur généré fait appel à yylex() pour avoir le prochain token à analyser. Deux informations doivent être transmises : le token et, éventuellement, la valeur associée. Il faut donc 1/ définir les types possibles et 2/les tokens avec éventuellement leur type :

```
%union
{
    int ival;
    float fval;
}
```

```
%token TOK_PLUS "+"
        TOK_MOINS "-'
%token<ival> TOK_NUMBER "number"
%type<ival> exp
```

Bison

- La Grammaire
  - L'analyseur généré fait appel à yylex() pour avoir le prochain token à analyser. Deux informations doivent être transmises : le token et, éventuellement, la valeur associée. Il faut donc 1/ définir les types possibles et 2/les tokens avec éventuellement leur type :

```
%union
{
    int ival;
    float fval;
}
```

```
%define api.tokens.prefix "TOK_"
%token PLUS "+"
       MOINS "-'
%token<ival> NUMBER "number"
%type<ival> exp
```

Bison

- La Grammaire
  - Définir, dans le prologue, les priorités et l'associativité :

    %left
    %right
    %nonassoc
    %precedence

Bison

- Un exemple (prologue) :

```
%expect 0
%debug
%defines

%code provides {
	void yyerror(const char *msg);
	...
}

%left ...

%define api.tokens.prefix ...
%token ...
%type ...
```

Bison

- Un exemple (règles) :

```
%%
nt1:
      nt2 "t3"  {printf("%d\n", $1);}
    | nt3 "t4"  {printf("%d\n", $1);}
    ;
%%
```

Bison

- Un exemple (épilogue) :

```
void yyerror(const char *msg)
{
    ...
}

int main(int argc, char *argv[])
{
    ...
}
```

Bison

- Faire générer l'analyseur lexical par flex.

Flex

- Structure proche des fichiers :

prologue
%%
règles
%%
epilogue

Flex

- Dans le prologue :
  - Insertion de code %{ %}
  - Ajout d'options par %option

Flex

- Règles

```
%%
"abc"    {action;}
[0-9]+   {action;}
%%
```

Flex

- Epilogue

Ajout direct de lignes de code...

Bison et Flex

- Variables importantes
  - yylval
  - yytext
  - yyleng