Compilation FLEX/BISON
Cahier d'apprentissage

Licence informatique 3ème année

Université de La Rochelle

 [1]

2025-2026_1

[1] https://creativecommons.org/licenses/by-nc-nd/4.0/deed.fr

Table des matières

Liste des exercices

Préambule

Un passage identifié par l'icone suivante représente une information à retenir:

À retenir

Un passage identifié par l'icone suivante représente un fichier source:

Fichier source

Un passage identifié par l'icone suivante représente une session avec un terminal:

```
$ console
```

Un passage identifié par l'icone suivante représente un exercice à réaliser:

Exercice

1 Introduction

En informatique, le terme compilateur[2] est utilisé pour traduire un langage de haut niveau vers un langage de bas niveau. Par exemple :

- un compilateur du langage C traduit du code écrit en C vers une représentation interne directement exécutable par le système d'exploitation hôte
- un compilateur du langage `java` traduit du code écrit en `java` vers une représentation directement exécutable par une machine virtuelle `java`.

Lorsque l'outil considéré traduit un langage vers un langage du même niveau, le bon terme à employer est un transpilateur ou transpiler[3].

Enfin, un interpréteur[4] est un outil capable d'exécuter un programme écrit dans un langage de haut niveau. L'opération d'interprétation peut être réalisée à chaque exécution du programme ou mise en cache afin d'accélérer l'exécution.

- le langage PHP est un langage dit *interprété*.
- le langage `python` est un langage dit *interprété*. Il mémorise l'interprétation de chaque fichier lorsqu'ils sont modifiés.

[2] https://fr.wikipedia.org/wiki/Compilateur
[3] https://fr.wikipedia.org/wiki/Compilateur_source_à_source
[4] https://fr.wikipedia.org/wiki/Interprète_(informatique)

Dans ce cahier, nous allons explorer pas à pas la création d'un transpilateur permettant de traduire un programme écrit dans le langage *facile* (que nous décrirons) vers l'assembleur[5] du langage c#[6]. Par abus de langage, nous utiliserons le mot compilateur dans le reste du texte.

Il y a généralement plusieurs phases pour aboutir à la création d'un compilateur :

- l'analyseur lexical est responsable de la découpe du texte en éléments atomiques (lexicaux). L'utilisation d'expressions régulières[7] permet de simplifier l'analyse des éléments complexes par leur traduction en automate à états finis[8]. L'outil historique s'appelle `lex`[9] mais nous utiliserons une version plus rapide `flex`[10]. Ces logiciels créent un automate à états finis écrit en langage C à partir d'un fichier source écrit dans leur propre langage. Ils sont considérés comme des compilateurs.
- l'analyseur syntaxique va vérifier que l'agencement des éléments lexicaux obéit à une grammaire non contextuelle[11] à l'aide d'un automate à pile[12]. L'outil historique s'appelle `yacc`[13] (*Yet Another Compiler of Compilers*). Pour la petite histoire, le nom de l'animal *yack* a donné le logo du projet GNU[14] (l'animal *gnou*). Dans ce document, nous utiliserons l'outil `bison`[15] qui est une version libre et plus rapide que `yacc`. Ces logiciels créent un automate à pile écrit en langage C à partir d'un fichier source écrit dans leur propre langage (qui incorpore également des instructions en C). Ils sont également considérés comme des compilateurs.
- l'analyseur sémantique permet de vérifier certaines choses qui ne peuvent être réalisées par une grammaire non contextuelle[16]. Nous utiliserons le langage C incorporé dans les sources de `flex` et `bison` directement. Au final, nous avons en sortie un arbre d'analyse représentant le source de notre langage.
- l'optimiseur tente de réduire l'arbre d'analyse en recherchant des optimisations dans le code (comme le font les options `-Ox` du compilateur `gcc`[17].
- le générateur de code traduit l'arbre traduit l'arbre d'analyse dans le langage cible du compilateur. Dans notre cas, il s'agira de l'assembleur C#[18].
- enfin, il peut y avoir une dernière phase d'optimisation liée au langage cible du compilateur. Nous n'aurons pas cette phase.

- `flex` et `bison` sont des outils pour créer des compilateurs ;
- un *transpileur* convertir des fichiers sources entre langages de même niveau ;
- un *compilateur* convertit des fichiers sources entre un langage de haut niveau vers un langage de bas niveau ;

[5] https://fr.wikipedia.org/wiki/Common_Intermediate_Language
[6] https://fr.wikipedia.org/wiki/C_sharp
[7] https://fr.wikipedia.org/wiki/Expression_régulière
[8] https://fr.wikipedia.org/wiki/Automate_fini
[9] https://fr.wikipedia.org/wiki/Lex_(logiciel)
[10] https://fr.wikipedia.org/wiki/Flex_(logiciel)
[11] https://fr.wikipedia.org/wiki/Grammaire_non_contextuelle
[12] https://fr.wikipedia.org/wiki/Automate_à_pile
[13] https://fr.wikipedia.org/wiki/Yacc_(logiciel)
[14] https://fr.wikipedia.org/wiki/GNU
[15] https://fr.wikipedia.org/wiki/GNU_Bison
[16] https://fr.wikipedia.org/wiki/Grammaire_non_contextuelle
[17] https://fr.wikipedia.org/wiki/GNU_Compiler_Collection
[18] https://fr.wikipedia.org/wiki/C_sharp

- une *grammaire* d'un langage informatique peut être exprimée par la norme de Backus-Naur ou par des diagrammes de Conway.

2 Le langage `facile`

Pour décrire un langage informatique, on utilise ce que l'on appelle une grammaire. Cette grammaire peut être décrite par différents outils textuels ou graphiques. La description textuelle utilise généralement la forme de Backus-Naur[19] ou sa version étendue[20] normalisée dans la norme ISO. La description graphique utilise quant à elle généralement les diagrammes de Conway[21] ou diagrammes syntaxiques. C'est par cette représentation que sera décrit le langage `facile`. Le langage `facile` est un langage permettant de manipuler les nombres entiers, les booléens, leurs opérations habituelles et les structures algorithmiques classiques que sont les tests et les répétitions.

Dans cette représentation, les formes graphiques suivantes seront utilisées (voir Railroad Diagram Generator[22]) :

pour représenter des éléments terminaux de la grammaire.

pour représenter des éléments terminaux de la grammaire (avec leur expression régulière).

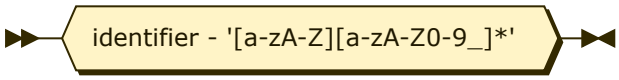
pour représenter des éléments non-terminaux.

La grammaire est volontairement ambiguë, il faudra certainement la retravailler afin de lever certaines ambiguïtés.

2.1 Identificateurs

Un identificateur est un élément terminal décrit par une expression régulière :

identifier :

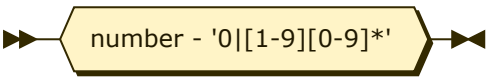


2.2 Nombres

Un nombre est soit le chiffre 0 tout seul soit un chiffre différent de 0 suivi d'autres chiffres.

[19] https://fr.wikipedia.org/wiki/Forme_de_Backus-Naur
[20] https://fr.wikipedia.org/wiki/Extended_Backus-Naur_Form
[21] https://fr.wikipedia.org/wiki/Diagramme_syntaxique
[22] https://bottlecaps.de/rr/ui

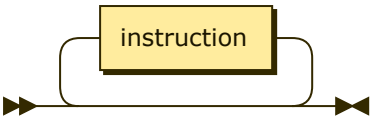
number :



2.3 Programme

Un programme est une suite d'instructions.

code :

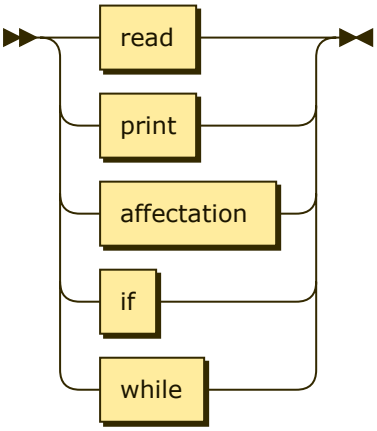


2.4 Instructions

Une instruction peut être soit :

- une lecture au clavier
- une écriture à l'écran
- une affectation
- un test
- une boucle

instruction :



2.5 Instruction de lecture

Une instruction de lecture commence par le mot-clé `read` suivi du nom de la variable à lire suivi par un point-virgule.

read :

2.6 Instruction d'écriture

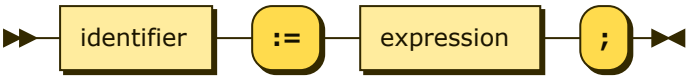
Une instruction d'écriture commence par le mot-clé `print` suivi d'une expression numérique suivi par un point-virgule.

print :

2.7 Instruction d'affectation

Une instruction d'affectation commence par un identificateur suivi de l'opérateur d'affectation `:=` suivi d'une expression et suivi par un point-virgule.

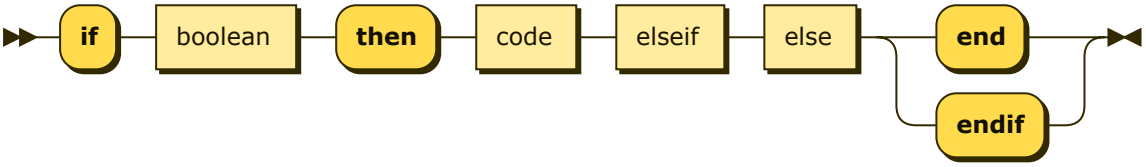
affectation :



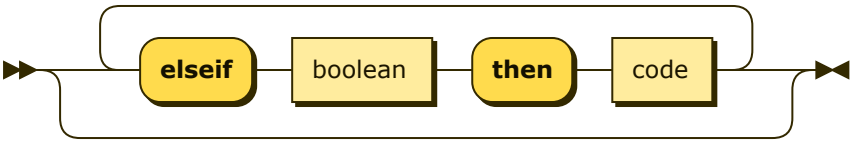
2.8 Instruction de test

L'instruction de test contient les mots-clés `if` et `then` et se termine par le mot-clé `end` ou `endif`. Elle peut contenir une répétition optionnelle de `elseif` et un `else` final optionnel.

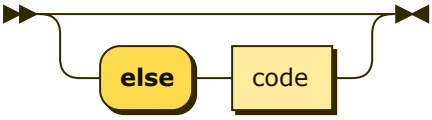
if :



elseif :



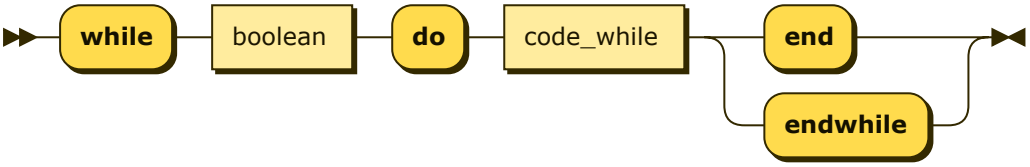
else :



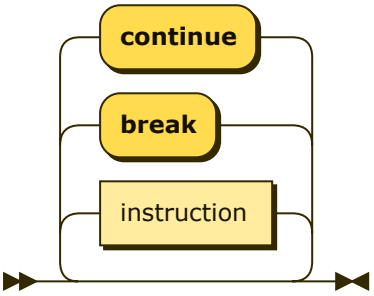
2.9 Instruction de boucle

Une instruction de boucle commence par le mot-clé `while` suivi par une expression booléenne, suivi par le mot-clé do suivi par du code et se terminant par le mot-clé end ou `endwhile`.

while :



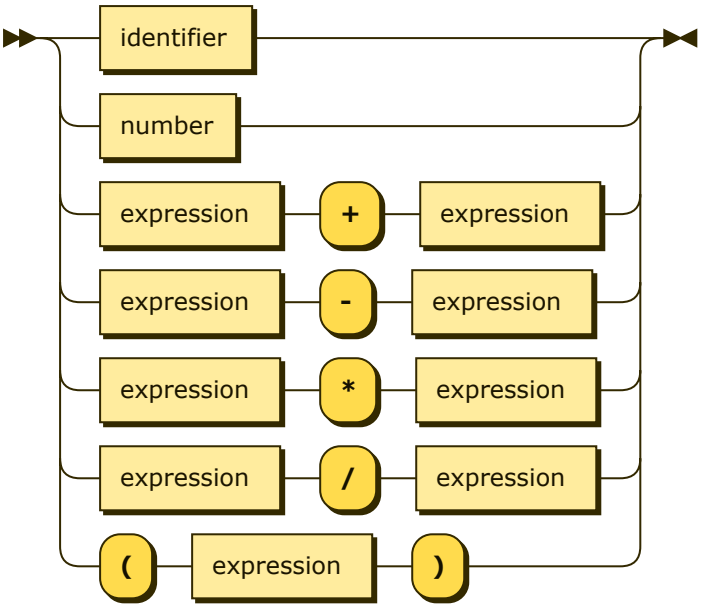
code_while :



2.10 Expression

Une expression permet de calculer une valeur numérique entière

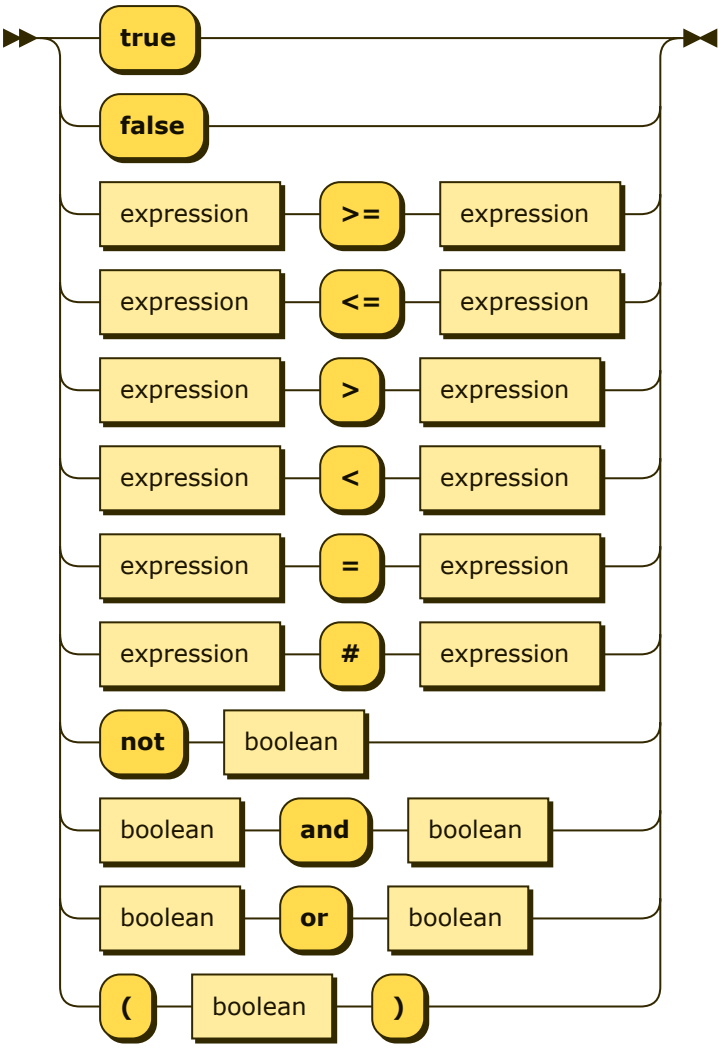
expression :



2.11 Booléens

Un booléen permet de calculer une valeur booléenne

boolean :



L'opérateur # représente la non-égalité.

- le langage `facile` est un langage simple ne manipulant que des entiers et ne comportant pas de fonctions. Il incorpore cependant les tests et les boucles ;
- le langage `facile` vise à être traduit en instructions de l'assembleur *Common Intermediate Language* de C#.

3 Analyseur lexical

L'analyseur lexical est responsable de la découpe du texte créé par le programmeur en unités atomiques. L'outil que nous utilisons est le programme `flex`.

`flex` permet de créer du code écrit en langage C à partir d'un fichier de description des éléments terminaux. Le fichier décrivant les éléments terminaux porte généralement l'extension `.lex` ou `.flex`.

Il comporte 3 parties séparées par des lignes contenant les caractères %% (voir la documentation de `flex`):

- la première partie contient des définitions ;
- la seconde partie contient des règles d'analyse lexicale ;
- la dernière partie contient exclusivement du code écrit en langage C.

3.1 Analyseur basique

```
%%
%%

/*
 * file: facile.lex
 * version: 0.1.0
 */
```

Pour créer notre premier analyseur, nous allons utiliser le système de compilation fourni avec `cmake`. Nous devons tout d'abord écrire un fichier `CMakeLists.txt` permettant d'exprimer ce que nous voulons créer.

```
# file: CMakeLists.txt
# version: 0.1.0

cmake_minimum_required(VERSION 3.0)

project(facile VERSION 0.1.0 LANGUAGES C)

# Search for the flex cmake package
find_package(FLEX)

# Definition of a scanner
flex_target(
  FACILE_SCANNER
  facile.lex
  "${CMAKE_CURRENT_BINARY_DIR}/facile.lex.c"
)

# Creation of the "facile" executable
add_executable(facile ${FLEX_FACILE_SCANNER_OUTPUTS})

# Add flex library to the "facile" executable
target_link_libraries(facile fl)

# Add zip generator
set(CPACK_SOURCE_GENERATOR "ZIP")
set(CPACK_SOURCE_IGNORE_FILES "build;~$;${CPACK_SOURCE_IGNORE_FILES}")
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
include(CPack)
```


Pour créer le premier analyseur:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Pour exécuter l'analyseur:

```
$ ./facile
```

Il lit les caractères depuis le clavier et ne fait que les recopier à l'écran. Pour sortir de l'analyseur, il faut utiliser le caractère de fin de fichier CTRL−D.

3.2 Analyseur des mots-clés

Reconnaître les mots-clés d'un langage est l'étape la plus facile. Il suffit d'indiquer dans la partie des règles quel est le mot-clé et de lui associer un bloc ou instruction écrits en langage C. Il faut que le code associé retourne un entier représentant la catégorie de ce que l'on a reconnu. Pour différencier les caractères seuls des expressions complexes, `flex` considère que les nombres accessibles commencent à la valeur 258.

```
%{
#include <assert.h>

#define TOK_IF    258
#define TOK_THEN  259
%}

%%

if {
  assert(printf("'if' found"));
  return TOK_IF;
}

then {
  assert(printf("'then' found"));
  return TOK_THEN;
}

[ab]*a[ab]*b[ab]*b[ab]*a assert(printf("'abba' found")); return yytext[0];

%%

/*
 * file: facile.lex
```

```
 * version: 0.2.0
 */
```

L'analyseur reconnaît maintenant les mots-clés du langage que nous avons défini (et imprime les messages lorsque l'on n'est pas en mode *Release*).

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make
$ ./facile
hello
hello
if
'if' found
then
'then' found
world
world
```

Exercice 1 *(Ajout des mots-clés)*
Ajoutez dans le fichier `facile.lex` les autres mots clés du langage. Nous verrons par la suite comment gérer les chaines contenant des caractères spéciaux, les identificateurs et les nombres.

3.3 Analyseur des ponctuations

L'analyse des symboles de ponctuation tels que : `:=`, `;` ... n'est pas plus difficile que les mots-clés, mis à part qu'ils font partie des caractères spéciaux de l'outil `flex`. Le plus simple est de les entourer de double-quote `"`.

```
%{
#include <assert.h>

#define TOK_IF          258
#define TOK_THEN        259
#define TOK_SEMI_COLON  260
#define TOK_AFFECTATION 261
#define TOK_ADD         262
#define TOK_SUB         263
#define TOK_MUL         264
#define TOK_DIV         265
%}

%%

if {
  assert(printf("'if' found"));
  return TOK_IF;
}

then {
```

```
  assert(printf("'then' found"));
  return TOK_THEN;
}

";" {
  assert(printf("';' found"));
  return TOK_SEMI_COLON;
}

":=" {
  assert(printf("':=' found"));
  return TOK_AFFECTATION;
}

"+" {
  assert(printf("'+' found"));
  return TOK_ADD;
}

"-" {
  assert(printf("'-' found"));
  return TOK_SUB;
}

"*" {
  assert(printf("'*' found"));
  return TOK_MUL;
}

"/" {
  assert(printf("'/' found"));
  return TOK_DIV;
}

%%

/*
 * file: facile.lex
 * version: 0.3.0
 */
```

Exercice 2 *(Ajout des ponctuations)*
Ajoutez dans le fichier `facile.lex` les autres ponctuations du langage.

3.4 Analyseur des identificateurs et des nombres

À ce stade, notre analyseur analyse presque tous les éléments du langage, il ne lui reste plus qu'à prendre en compte les identificateurs et les nombres. Nous avons besoin d'utiliser les expressions

régulières rendues possibles par l'outil `flex`. Nous en profitons pour presenter deux variables globales disponibles dans tout le programme :

- `yytext` contient un pointeur vers le dernier élément lexical analysé ;
- `yyleng` contient le nombre de caractères du dernier élément lexical analysé.

```
%{
#include <assert.h>

#define TOK_IF          258
#define TOK_THEN        259
#define TOK_SEMI_COLON  260
#define TOK_AFFECTATION 261
#define TOK_ADD         262
#define TOK_SUB         263
#define TOK_MUL         264
#define TOK_DIV         265
#define TOK_IDENTIFIER  266
%}

%%

if {
  assert(printf("'if' found"));
  return TOK_IF;
}

then {
  assert(printf("'then' found"));
  return TOK_THEN;
}

";" {
  assert(printf("';' found"));
  return TOK_SEMI_COLON;
}

":=" {
  assert(printf("':=' found"));
  return TOK_AFFECTATION;
}

"+" {
  assert(printf("'+' found"));
  return TOK_ADD;
}

"-" {
  assert(printf("'-' found"));
```

```
  return TOK_SUB;
}

"*" {
  assert(printf("'*' found"));
  return TOK_MUL;
}

"/" {
  assert(printf("'/' found"));
  return TOK_DIV;
}

[a-zA-Z][a-zA-Z0-9_]* {
  assert(printf("identifier '%s(%d)' found", yytext, yyleng));
  return TOK_IDENTIFIER;
}

%%

/*
 * file: facile.lex
 * version: 0.4.0
 */
```

```
$ make
$ ./facile
hello
identifier 'hello(5)' found
```

Exercice 3 *(Ajout des nombres)*
Ajoutez dans le fichier `facile.lex` l'analyse des nombres tels qu'ils sont définis dans la norme du langage `facile`.

3.5 Analyseur des autres caractères

Lorsque l'on écrit un analyseur lexical, il faut prendre en compte tous les caractères. Si l'on ne le fait pas, `flex` considère que l'action associée aux caractères non définis est leur affichage à l'écran. De plus, dans la plupart des langages informatiques, les *séparateurs blancs* (espace, tabulation, saut de ligne) sont généralement ignorés (instructions du langage C limitées à une instruction vide `;`).

```
%{
#include <assert.h>

#define TOK_IF          258
#define TOK_THEN        259
#define TOK_SEMI_COLON  260
#define TOK_AFFECTATION 261
```

```
#define TOK_ADD         262
#define TOK_SUB         263
#define TOK_MUL         264
#define TOK_DIV         265
#define TOK_IDENTIFIER  266
%}

%%

if {
  assert(printf("'if' found"));
  return TOK_IF;
}

then {
  assert(printf("'then' found"));
  return TOK_THEN;
}

";" {
  assert(printf("';' found"));
  return TOK_SEMI_COLON;
}

":=" {
  assert(printf("':=' found"));
  return TOK_AFFECTATION;
}

"+" {
  assert(printf("'+' found"));
  return TOK_ADD;
}

"-" {
  assert(printf("'-' found"));
  return TOK_SUB;
}

"*" {
  assert(printf("'*' found"));
  return TOK_MUL;
}

"/" {
  assert(printf("'/' found"));
  return TOK_DIV;
}
```



```
[a-zA-Z][a-zA-Z0-9_]* {
  assert(printf("identifier '%s(%d)' found", yytext, yyleng));
  return TOK_IDENTIFIER;
}

[ \t\n] ;

. {
  return yytext[0];
}

%%

/*
 * file: facile.lex
 * version: 0.5.0
 */
```

- À ce stade, l'analyseur lexical est complet. Il est capable d'analyser tous les éléments du langage, d'ignorer les espaces et de retourner la catégorie des éléments lexicaux en utilisant un entier. Nous reviendrons sur son contenu dans le chapitre suivant lorsque nous écrirons les premières règles syntaxiques de notre grammaire.
- L'outil `flex` utilise des expressions régulières pour analyser les éléments lexicaux.

4 Analyseur syntaxique

L'outil que nous allons utiliser pour écrire l'analyseur syntaxique s'appelle `bison`. C'est l'héritier d'un outil anciennement nommé `yacc`. `bison` fonctionne comme l'analyseur lexical `flex`. Il produit, à partir d'un fichier décrivant la grammaire à analyser, un fichier écrit en langage C permettant de vérifier qu'un code source correspond à la grammaire décrite.

4.1 Analyseur basique

Nous allons tout d'abord créer un analyseur basique permettant d'analyse un fichier vide. La grammaire d'un fichier que `bison` est capable d'analyser s'écrit dans un fichier texte ayant comme extension `.y` (en référence à `yacc`).

Il comporte 3 parties séparées par des lignes contenant les caractères %% (voir la documentation de `bison`), comme son homologue de `flex` :

- la première partie contient des définitions ;
- la seconde partie contient des règles d'analyse syntaxique ;
- la dernière partie contient exclusivement du code écrit en langage C.

Nous devons tout d'abord modifier le fichier `CMakeLists.txt` permettant d'utiliser `bison`.


```
# file: CMakeLists.txt
# version: 0.6.0

cmake_minimum_required(VERSION 3.0)

project(facile VERSION 0.6.0 LANGUAGES C)

# Search for the flex cmake package
find_package(FLEX)

# Definition of a scanner
flex_target(
  FACILE_SCANNER
  facile.lex
  "${CMAKE_CURRENT_BINARY_DIR}/facile.lex.c"
)

# Definition of a parser
find_package(BISON)
bison_target(
  FACILE_PARSER
  facile.y
  "${CMAKE_CURRENT_BINARY_DIR}/facile.y.c"
)

# Creation of the "facile" executable
add_executable(
  facile
  ${FLEX_FACILE_SCANNER_OUTPUTS}
  ${BISON_FACILE_PARSER_OUTPUTS}
)

# Add flex library to the "facile" executable
target_link_libraries(facile fl)

# Add zip generator
set(CPACK_SOURCE_GENERATOR "ZIP")
set(CPACK_SOURCE_IGNORE_FILES "build;~$;${CPACK_SOURCE_IGNORE_FILES}")
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
include(CPack)
```

L'analyseur basique permettant d'analyser un fichier vide doit contenir néanmoins une règle d'analyse.

```
%{
#include <stdlib.h>
```

```
#include <stdio.h>

extern int yylex(void);
extern int yyerror(const char *msg);
%}

%%

program: ;

%%

/*
 * file: facile.y
 * version: 0.6.0
 */

int yyerror(const char *msg) {
  fprintf(stderr, "%s\n", msg);
}

int main(int argc, char *argv[]) {
  yyparse();
  return EXIT_SUCCESS;
}
```

La première partie contient des inclusions de C standard ainsi que la déclaration de deux fonctions essentielles pour le compilateur :

- `yylex` qui est responsable de l'analyse lexicale des éléments du langage. Vous noterez ici le nom de cette fonction qui est exactement le même que celle produite par le logiciel `flex`. `bison` et `flex` sont conçus pour fonctionner ensemble ;
- `yyerror` qui est responsable du traitement des erreurs. Nous l'avons défini dans la troisème partie comme affichant le message d'erreur sur le fichier d'erreurs.

La seconde partie contient les règles de la grammaire. Dans cet exemple, il n'y a qu'une règle indiquant qu'un programme est défini par rien.

La troisième partie contient du code C. Nous avons défini la fonction `yyerror` et la fonction `main` qui se contente d'appeler la fonction `yyparse` et de retourner le code de succès en sortie. La fonction `yyparse` va être créee par `bison`. Elle correspond aux règles décrites dans la grammaire et utilise la fonction `yylex` comme analyseur lexical.

4.2 Collaboration entre `flex` et `bison`

Nous avons déjà vu que `bison` collabore avec `flex` au travers de la fonction `yylex`. La collaboration va bien plus loin puisqu'il est possible qu'ils utilisent facilement la même façon de coder les catégories

lexicales au moyen d'un fichier que bison peut produire.

En indiquant maintenant à bison une grammaire permettant de gérer des affectations d'entiers à des identificateurs :

```
%{
#include <stdlib.h>
#include <stdio.h>

extern int yylex(void);
extern int yyerror(const char *msg);
%}


%token TOK_NUMBER
%token TOK_IDENTIFIER
%token TOK_AFFECTATION
%token TOK_SEMI_COLON
%token TOK_IF
%token TOK_THEN
%token TOK_ADD
%token TOK_SUB
%token TOK_MUL
%token TOK_DIV

%%

program: code;

code: code instruction | ;

instruction: affectation ;

affectation:
  identifier TOK_AFFECTATION expression TOK_SEMI_COLON;

expression:
  identifier |
  number ;

identifier:
  TOK_IDENTIFIER ;

number:
  TOK_NUMBER ;

%%

/*
```

```
 * file: facile.y
 * version: 0.7.0
 */

int yyerror(const char *msg) {
  fprintf(stderr, "%s\n", msg);
}

int main(int argc, char *argv[]) {
  yyparse();
  return EXIT_SUCCESS;
}
```

Les catégories des éléments terminaux ont été décrites au moyen du mot-clé **%token** de bison et la grammaire contient maintenant 7 règles permettant de contenir une série d'affectation. Si l'on effectue la compilation du projet :

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
-- The C compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found FLEX: /usr/bin/flex (found version "2.6.4")
-- Found BISON: /usr/bin/bison (found version "3.0.4")
-- Configuring done
-- Generating done
-- Build files have been written to: .../build
$ make
[ 20%] [BISON][FACILE_PARSER] Building parser with bison 3.0.4
[ 40%] [FLEX][FACILE_SCANNER] Building scanner with flex 2.6.4
Scanning dependencies of target facile
[ 60%] Building C object CMakeFiles/facile.dir/facile.lex.c.o
[ 80%] Building C object CMakeFiles/facile.dir/facile.y.c.o
[100%] Linking C executable facile
[100%] Built target facile
$ ls -1
CMakeCache.txt
CMakeFiles
cmake_install.cmake
facile
facile.lex.c
facile.y.c
facile.y.h
```


`Makefile`

Vous noterez que `bison` a produit deux fichiers :

- `facile.y.c` qui contient le code de l'analyseur syntaxique ;
- `facile.y.h` qui contient des définitions et notamment la description des éléments terminaux définis par la grammaire au moyen d'entiers commençant au numéro 258.

Nous pouvons maintenant remplacer les instructions de définition de constantes que nous avions écrites dans le fichier `facile.lex` au moyen du préprocesseur du langage C et de la construction `#define` par l'inclusion du fichier `facile.y.h` :

```
%{
#include <assert.h>

#include "facile.y.h"
%}

%%

if {
  assert(printf("'if' found"));
  return TOK_IF;
}

then {
  assert(printf("'then' found"));
  return TOK_THEN;
}

";" {
  assert(printf("';' found"));
  return TOK_SEMI_COLON;
}

":=" {
  assert(printf("':=' found"));
  return TOK_AFFECTATION;
}

"+" {
  assert(printf("'+' found"));
  return TOK_ADD;
}

"-" {
  assert(printf("'-' found"));
  return TOK_SUB;
}
```

```
"*" {
  assert(printf("'*' found"));
  return TOK_MUL;
}

"/" {
  assert(printf("'/' found"));
  return TOK_DIV;
}

[a-zA-Z][a-zA-Z0-9_]* {
  assert(printf("identifier '%s(%d)' found", yytext, yyleng));
  return TOK_IDENTIFIER;
}

"0" {
  assert(printf("number '%s(%d)' found", yytext, yyleng));
  return TOK_NUMBER;
}

[ \t\n] ;

. {
  return yytext[0];
}

%%

/*
 * file: facile.lex
 * version: 0.7.0
 */
```

Nous ponvons maintenant tester notre compilateur en écrivant les lignes de codes directement au clavier :

```
$ ./facile
a:=0;
b:=a;
c:=1;
syntax error
```

Dès que bison détecte une erreur, il affiche le message d'erreur détectée.

4.3 Gestion des erreurs

Dans la section précédente, nous avons vu que l'affichage des erreurs était quelque peu succint. bison propose un mécanisme afin d'afficher une erreur plus compréhensible par le programmeur :

Tout d'abord, nous allons activer la verbosité des erreurs en ajoutant la ligne

```
%define parse.error verbose
```

dans le fichier `bison`.

Ensuite, nous allons associer à chaque élément terminal du langage une expression textuelle au travers de la construction **`%token`**.

Enfin, nous allons activer l'option **`%option yylineno`** dans le fichier `flex` afin qu'il maintienne un numero de ligne courant (accessible par la variable globale `yylineno`.

```
%{
#include <assert.h>

#include "facile.y.h"
%}

%option yylineno

%%

if {
  assert(printf("'if' found"));
  return TOK_IF;
}

then {
  assert(printf("'then' found"));
  return TOK_THEN;
}

";" {
  assert(printf("';' found"));
  return TOK_SEMI_COLON;
}

":=" {
  assert(printf("':=' found"));
  return TOK_AFFECTATION;
}

"+" {
  assert(printf("'+' found"));
  return TOK_ADD;
}

"-" {
  assert(printf("'-' found"));
  return TOK_SUB;
```

```
}

"*" {
  assert(printf("'*' found"));
  return TOK_MUL;
}

"/" {
  assert(printf("'/' found"));
  return TOK_DIV;
}

[a-zA-Z][a-zA-Z0-9_]* {
  assert(printf("identifier '%s(%d)' found", yytext, yyleng));
  return TOK_IDENTIFIER;
}

"0" {
  assert(printf("number '%s(%d)' found", yytext, yyleng));
  return TOK_NUMBER;
}

[ \t\n] ;

. {
  return yytext[0];
}

%%

/*
 * file: facile.lex
 * version: 0.8.0
 */
```

```
%{
#include <stdlib.h>
#include <stdio.h>

extern int yylex(void);
extern int yyerror(const char *msg);
extern int yylineno;
%}

%define parse.error verbose

%token TOK_NUMBER      "number"
%token TOK_IDENTIFIER  "identifier"
%token TOK_AFFECTATION ":="
```

```
%token TOK_SEMI_COLON  ";"
%token TOK_IF          "if"
%token TOK_THEN        "then"
%token TOK_ADD         "+"
%token TOK_SUB         "-"
%token TOK_MUL         "*"
%token TOK_DIV         "/"

%%

program: code;

code: code instruction | ;

instruction: affectation ;

affectation:
  identifier TOK_AFFECTATION expression TOK_SEMI_COLON;

expression:
  identifier |
  number ;

identifier:
  TOK_IDENTIFIER ;

number:
  TOK_NUMBER ;

%%

/*
 * file: facile.y
 * version: 0.8.0
 */

int yyerror(const char *msg) {
  fprintf(stderr, "Line %d: %s\n", yylineno, msg);
}

int main(int argc, char *argv[]) {
  yyparse();
  return EXIT_SUCCESS;
}
```

- `bison` permet de créer l'analyseur syntaxique par l'intermédiaire d'un fichier de description de la grammaire ;

- `bison` et `flex` sont faits pour fonctionner ensemble : `bison` utilise la fonction `yylex` créée par `flex` ;
- `flex` est capable de maintenir le numéro de ligne courant et bison peut afficher des messages d'erreurs compréhensibles.

5 Génération de code

5.1 Fichier minimal `cil`

Comme nous l'avions déjà écrit dans les chapitres précédents, nous allons utiliser l'assembleur[23] du *Common Intermediate Language* pour réaliser notre compilateur. Il est possible de trouver sur la version anglaise de *wikipedia* la liste des instructions supportées[24].

Vous pourrez également utiliser le compilateur `c#` (`mcs` sous linux) et le désassembleur (`monodis` sous linux) pour observer les structures de code utilisées par le le compilateur `c#`.

Tout d'abord, nous devons connaître le nombre minimum d'instructions en langage `cil` pour pouvoir être compilé avec l'assembleur `ilasm`. Un rapide coup d'œil à la version anglaise de *wikipedia*[25] concernant le langage `cil` permet d'en avoir une idée. La structure du code que nous allons générer reprendra le même schéma.

Soit le fichier `ReadWrite.cs` suivant :

```
/*
 * file: ReadWrite.cs
 * version: 0.9.0
 */

using System;

public class ReadWrite {
  static public void Main () {
    int a;
    a = int.Parse(Console.ReadLine());
    Console.WriteLine(a);
  }
}
```

```
$ mcs ReadWrite.cs
$ ls -1
build
CMakeLists.txt
facile.lex
facile.y
```

[23] https://fr.wikipedia.org/wiki/Common_Intermediate_Language
[24] https://en.wikipedia.org/wiki/List_of_CIL_instructions
[25] https://en.wikipedia.org/wiki/Common_Intermediate_Language#Example

```
ReadWrite.cs
ReadWrite.exe
$ monodis ReadWrite.exe > ReadWrite.il
$ tail -20 ReadWrite.il
    // method line 2
    .method public static hidebysig
           default void Main ()  cil managed
    {
        // Method begins at RVA 0x2058
    .entrypoint
    // Code size 18 (0x12)
    .maxstack 1
    .locals init (
        int32   V_0)
    IL_0000:  call string class [mscorlib]System.Console::ReadLine()
    IL_0005:  call int32 int32::Parse(string)
    IL_000a:  stloc.0
    IL_000b:  ldloc.0
    IL_000c:  call void class [mscorlib]System.Console::WriteLine(int32)
    IL_0011:  ret
    } // end of method ReadWrite::Main

  } // end of class ReadWrite
```

Le fichier produit utilise les instructions de l'assembleur de la machine virtuelle gérée par le langage `cil` et des appels à la librairie standard `mscorlib`.

5.2 Compilateur minimal

Maintenant que nous avons un analyseur syntaxique fonctionnel (quoique basique), nous allons pouvoir générer du code.

Nous allons utiliser les possibilités offertes par la `glib`[26], notamment la structure d'arbres n-aires et les tables de hachage. Le fichier `CMakeLists.txt` doit donc être modifié pour la prendre en compte:

```
# Add glib
find_package(PkgConfig REQUIRED)
pkg_check_modules(GLIB2 REQUIRED glib-2.0)
include_directories(${GLIB2_INCLUDE_DIRS})
link_directories(${GLIB2_LIBRARY_DIRS})
add_definitions(${GLIB2_CFLAGS_OTHER})
target_link_libraries(facile ${GLIB2_LIBRARIES})
```

Tout d'abord, nous allons permettre à notre compilateur d'analyser un fichier en entrée afin d'éviter à avoir à écrire à chaque fois nos exemples en modifiant la fonction `main`. Vous aurez à ajouter quelques déclarations et certaines inclusions dans la première partie du fichier `facile.y`.

[26] https://fr.wikipedia.org/wiki/GLib

```
int main(int argc, char *argv[]) {
  if (argc == 2) {
    char *file_name_input = argv[1];
    char *extension;
    char *directory_delimiter;
    char *basename;
    extension = rindex(file_name_input, '.');
    if (!extension || strcmp(extension, ".facile") != 0) {
      fprintf(stderr, "Input filename extension must be '.facile'\n");
      return EXIT_FAILURE;
    }
    directory_delimiter = rindex(file_name_input, '/');
    if (!directory_delimiter) {
      directory_delimiter = rindex(file_name_input, '\\');
    }
    if (directory_delimiter) {
      basename = strdup(directory_delimiter + 1);
    } else {
      basename = strdup(file_name_input);
    }
    module_name = strdup(basename);
    *rindex(module_name, '.') = '\0';
    strcpy(rindex(basename, '.'), ".il");
    char *onechar = module_name;
    if (!isalpha(*onechar) && *onechar != '_') {
      free(basename);
      fprintf(stderr, "Base input filename must start with a letter or an
↪   underscore\n");
      return EXIT_FAILURE;
    }
    onechar++;
    while (*onechar) {
      if (!isalnum(*onechar) && *onechar != '_') {
        free(basename);
        fprintf(stderr, "Base input filename cannot contains special
↪   characters\n");
        return EXIT_FAILURE;
      }
      onechar++;
    }
    if (stdin = fopen(file_name_input, "r")) {
      if (stream = fopen(basename, "w")) {
        table = g_hash_table_new_full(g_str_hash, g_str_equal, free,
↪   NULL);
        yyparse();
        g_hash_table_destroy(table);
        fclose(stream);
        fclose(stdin);
```

```
      } else {
        free(basename);
        fclose(stdin);
        fprintf(stderr, "Output filename cannot be opened\n");
        return EXIT_FAILURE;
      }
    } else {
      free(basename);
      fprintf(stderr, "Input filename cannot be opened\n");
      return EXIT_FAILURE;
    }
    free(basename);
  }

  else {
    fprintf(stderr, "No input filename given\n");
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

Ensuite, nous allons utiliser la construction `bison` **`%union`** pour permettre de construire l'arbre d'analyse. Cette construction permet de déclarer quelle valeur peut prendre chaque élément de l'arbre.

Dans notre compilateur, nous n'avons besoin que de 3 valeurs possibles :

- `number` que nous allons déclarer comme étant un `gulong` ;
- `string` que nous allons déclarer comme étant un `gchar *` ;
- `node` que nous allons déclarer comme étant un `GNode *` (pointeur vers un arbre n-aire).

```
%union {
  gulong number;
  gchar *string;
  GNode * node;
}
```

Ces valeurs possibles vont devoir être utilisées dans le fichier `facile.lex` et `facile.y` afin de donner une valeur sémantique à tous les symboles de la grammaire (terminaux et non-terminaux).

```
%token<number> TOK_NUMBER          "number"
%token<string> TOK_IDENTIFIER      "identifier"
%token         TOK_AFFECTATION     ":="
%token         TOK_SEMI_COLON      ";"
%token         TOK_IF              "if"
%token         TOK_THEN            "then"
%left          TOK_ADD             "+"
%left          TOK_SUB             "-"
```

```
%left           TOK_MUL                 "*"
%left           TOK_DIV                 "/"
%token          TOK_OPEN_PARENTHESIS    "("
%token          TOK_CLOSE_PARENTHESIS   ")"
%token          TOK_PRINT               "print"
%token          TOK_READ                "read"

%type<node>     code
%type<node>     expression
%type<node>     instruction
%type<node>     identifier
%type<node>     print
%type<node>     read
%type<node>     affectation
%type<node>     number
```

Vous noterez l'utilisation du mot-clé **`%left`** permettant de préciser que l'opérateur est prioritaire à gauche. Cela permet de simplifier grandement l'écriture de la grammaire.

Pour les éléments terminaux décrits dans le fichier `facile.lex`, nous allons utiliser la variable globale `yylval` définie par `bison` dans le fichier `facile.y.h` pour la renseigner.

```
[a-zA-Z][a-zA-Z0-9_]* {
  assert(printf("identifier '%s(%d)' found", yytext, yyleng));
  yylval.string = yytext;
  return TOK_IDENTIFIER;
}

"0" {
  assert(printf("number '%s(%d)' found", yytext, yyleng));
  sscanf(yytext, "%lu", &yylval.number);
  return TOK_NUMBER;
}
```

Puis, dans le fichier `facile.y`, nous allons produire l'arbre syntaxique en ajoutant des instructions de construction dans chaque règle. `$i` représente la valeur du $i^{ème}$ élément de la règle et `$$` représente la valeur du symbole non-terminal défini par la règle.

```
program: code {
  begin_code();
  produce_code($1);
  end_code();
  g_node_destroy($1);
}
;

code:
  code instruction
  {
    $$ = g_node_new("code");
```

```
    g_node_append($$, $1);
    g_node_append($$, $2);
  }
|
  {
    $$ = g_node_new("");
  }
;

instruction:
  affectation |
  print |
  read
;

affectation:
  identifier TOK_AFFECTATION expression TOK_SEMI_COLON
  {
    $$ = g_node_new("affectation");
    g_node_append($$, $1);
    g_node_append($$, $3);
  }
;

print:
  TOK_PRINT expression TOK_SEMI_COLON
  {
    $$ = g_node_new("print");
    g_node_append($$, $2);
  }
;

read:
  TOK_READ identifier TOK_SEMI_COLON
  {
    $$ = g_node_new("read");
    g_node_append($$, $2);
  }
;


expression:
  identifier
|
  number
|
  expression TOK_ADD expression
  {
```

```
    $$ = g_node_new("add");
    g_node_append($$, $1);
    g_node_append($$, $3);
  }
|
  expression TOK_SUB expression
  {
    $$ = g_node_new("sub");
    g_node_append($$, $1);
    g_node_append($$, $3);
  }
|
  expression TOK_MUL expression
  {
    $$ = g_node_new("mul");
    g_node_append($$, $1);
    g_node_append($$, $3);
  }
|
  expression TOK_DIV expression
  {
    $$ = g_node_new("div");
    g_node_append($$, $1);
    g_node_append($$, $3);
  }
|
  TOK_OPEN_PARENTHESIS expression TOK_CLOSE_PARENTHESIS
  {
    $$ = $2;
  }
;

identifier:
  TOK_IDENTIFIER
  {
    $$ = g_node_new("identifier");
    gulong value = (gulong) g_hash_table_lookup(table, $1);
    if (!value) {
      value = g_hash_table_size(table) + 1;
      g_hash_table_insert(table, strdup($1), (gpointer) value);
    }
    g_node_append_data($$, (gpointer)value);
  }
;

number:
  TOK_NUMBER
  {
```

```
    $$ = g_node_new("number");
    g_node_append_data($$, (gpointer)$1);
  }
;
```

Enfin, nous allons écrire une fonction `produce_code` permettant de produire du code `cil` pour chaque élément de l'arbre syntaxique généré.

```
void produce_code(GNode* node) {
  if (node->data == "code") {
    produce_code(g_node_nth_child(node, 0));
    produce_code(g_node_nth_child(node, 1));
  } else if (node->data == "affectation") {
    produce_code(g_node_nth_child(node, 1));
    fprintf(stream, "        stloc\t%ld\n",
↪   (long)g_node_nth_child(g_node_nth_child(node, 0), 0)->data - 1);
  } else if (node->data == "add") {
    produce_code(g_node_nth_child(node, 0));
    produce_code(g_node_nth_child(node, 1));
    fprintf(stream, "        add\n");
  } else if (node->data == "sub") {
    produce_code(g_node_nth_child(node, 0));
    produce_code(g_node_nth_child(node, 1));
    fprintf(stream, "        sub\n");
  } else if (node->data == "mul") {
    produce_code(g_node_nth_child(node, 0));
    produce_code(g_node_nth_child(node, 1));
    fprintf(stream, "        mul\n");
  } else if (node->data == "div") {
    produce_code(g_node_nth_child(node, 0));
    produce_code(g_node_nth_child(node, 1));
    fprintf(stream, "        div\n");
  } else if (node->data == "number") {
    fprintf(stream, "        ldc.i4\t%ld\n", (long)g_node_nth_child(node,
↪   0)->data);
  } else if (node->data == "identifier") {
    fprintf(stream, "        ldloc\t%ld\n", (long)g_node_nth_child(node,
↪   0)->data - 1);
  } else if (node->data == "print") {
    produce_code(g_node_nth_child(node, 0));
    fprintf(stream, "        call void class
↪   [mscorlib]System.Console::WriteLine(int32)\n");
  } else if (node->data == "read") {
    fprintf(stream, "        call string class
↪   [mscorlib]System.Console::ReadLine()\n");
    fprintf(stream, "        call int32 int32::Parse(string)\n");
    fprintf(stream, "        stloc\t%ld\n",
↪   (long)g_node_nth_child(g_node_nth_child(node, 0), 0)->data - 1);
  }
```

```
}
```

Notre compilateur est prêt à l'emploi :

```
$ make
[ 20%] [BISON][FACILE_PARSER] Building parser with bison 3.0.4
[ 40%] [FLEX][FACILE_SCANNER] Building scanner with flex 2.6.4
Scanning dependencies of target facile
[ 60%] Building C object CMakeFiles/facile.dir/facile.lex.c.o
[ 80%] Building C object CMakeFiles/facile.dir/facile.y.c.o
[100%] Linking C executable facile
[100%] Built target facile
$ cat test.facile
read a;
read b;
c := a+b;
print c;
$ ./facile test.facile
$ ls -1
CMakeCache.txt
CMakeFiles
cmake_install.cmake
facile
facile.lex.c
facile.y.c
facile.y.h
Makefile
test.facile
test.il
$ cat test.il
.assembly test {}
.assembly extern mscorlib {}
.method static void Main()
{
    .entrypoint
    .maxstack 10
    .locals init (int32, int32, int32)
    call string class [mscorlib]System.Console::ReadLine()
    call int32 int32::Parse(string)
    stloc   0
    call string class [mscorlib]System.Console::ReadLine()
    call int32 int32::Parse(string)
    stloc   1
    ldloc   0
    ldloc   1
    add
    stloc   2
    ldloc   2
    call void class [mscorlib]System.Console::WriteLine(int32)
```

```
    ret
}
$ ilasm test.il
Assembling 'test.il' , no listing file, to exe --> 'test.exe'

Operation completed successfully
$ chmod 755 test.exe
$ ./test.exe
36
64
100
```

- `bison` permet d'associer des valeurs aux éléments terminaux et non-terminaux par l'intermédiaire de l'option **`%union`**. Au final, une variable globale `yylval` est accessible dans l'analyseur lexical pour faire remonter des informations ;
- `bison` peut associer des valeurs aux éléments non-terminaux en utilisant les constructions `$i` et `$$` ;
- le compilateur de `c#` sous linux s'appelle `mcs` ;
- le désassembleur de `c#` sous linux s'appelle `monodis` ;
- l'assembleur de `c#` sous linux s'appelle `ilasm` ;

Exercice 4 *(Gestion des tests simples)*
Ajoutez des règles au langage `facile` pour qu'il puisse gérer les instructions `if` sans l'utilisation des mots-clés `else` et `elseif` et sans tests imbriqués.

Exercice 5 *(Gestion des tests complexes)*
Ajoutez des règles au langage `facile` pour qu'il puisse gérer les instructions `if` avec l'utilisation des mots-clés `else` et `elseif`.

Exercice 6 *(Gestion des tests imbriqués)*
Ajoutez des règles au langage `facile` pour qu'il puisse gérer les instructions `if` avec des tests imbriqués.

Exercice 7 *(Gestion des boucles simples)*
Ajoutez des règles au langage `facile` pour qu'il puisse gérer les instructions `while` sans l'utilisation des mot-clés `break` et `continue` et sans boucles imbriquées.

Exercice 8 *(Gestion des boucles complexes)*
Ajoutez des règles au langage `facile` pour qu'il puisse gérer les instructions `while` avec l'utilisation des mot-clés `break` et `continue`.

Exercice 9 *(Gestion des boucles imbriquées)*
Ajoutez des règles au langage `facile` pour qu'il puisse gérer les instructions `while` avec des boucles imbriquées.

Exercice 10 *(Le* pgcd*)*
Écrivez un programme dans le langage `facile` permettant de calculer le plus grand commun diviseur de deux nombres saisis au clavier.

Historique des modifications

2019-2020_1 *Lundi 16 mars 2020*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Création de la première version.

2019-2020_2 *Vendredi 27 mars 2020*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Utilisation de la macro `assert` pour l'impression des messages ;
- Ajout de l'analyse des ponctuations ;
- Ajout de l'analyse des identificateurs ;
- Ignorance des caractères blancs.

2019-2020_3 *Mercredi 1er avril 2020*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Analyse des affectations.

2019-2020_4 *Dimanche 5 avril 2020*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Génération de code basique.

2020-2021_1 *Vendredi 25 février 2021*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Changement d'année ;
- Corrections de coquilles.

2020-2021_2 *Lundi 8 mars 2021*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Corrections de coquilles;
- Ajout des modifications du fichier `CMakeLists.txt` pour prendre en compte la `glib`. Merci à Jérémy Richard pour cette remarque.

2020-2021_3 *Lundi 15 mars 2021*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Ajout de méta-données.

2021-2022_1 *Lundi 4 mars 2022*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Changement d'année.

2023-2024_1 *Lundi 11 mars 2024*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Changement d'année.

2024-2025_1 *Mercredi 04 décembre 2024*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Changement d'année.

2024-2025_2 *Mercredi 12 mars 2025*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Fix listings.

2025-2026_1 *Mardi 3 févier 2026*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Changement d'année.

