Langages, Compilation, Automates. Partie 4: Compilateurs, Analyse lexicale et présentation du projet

Florian Bridoux

Polytech Nice Sophia

2022-2023

Table des matières

Table des matières

Compilateur

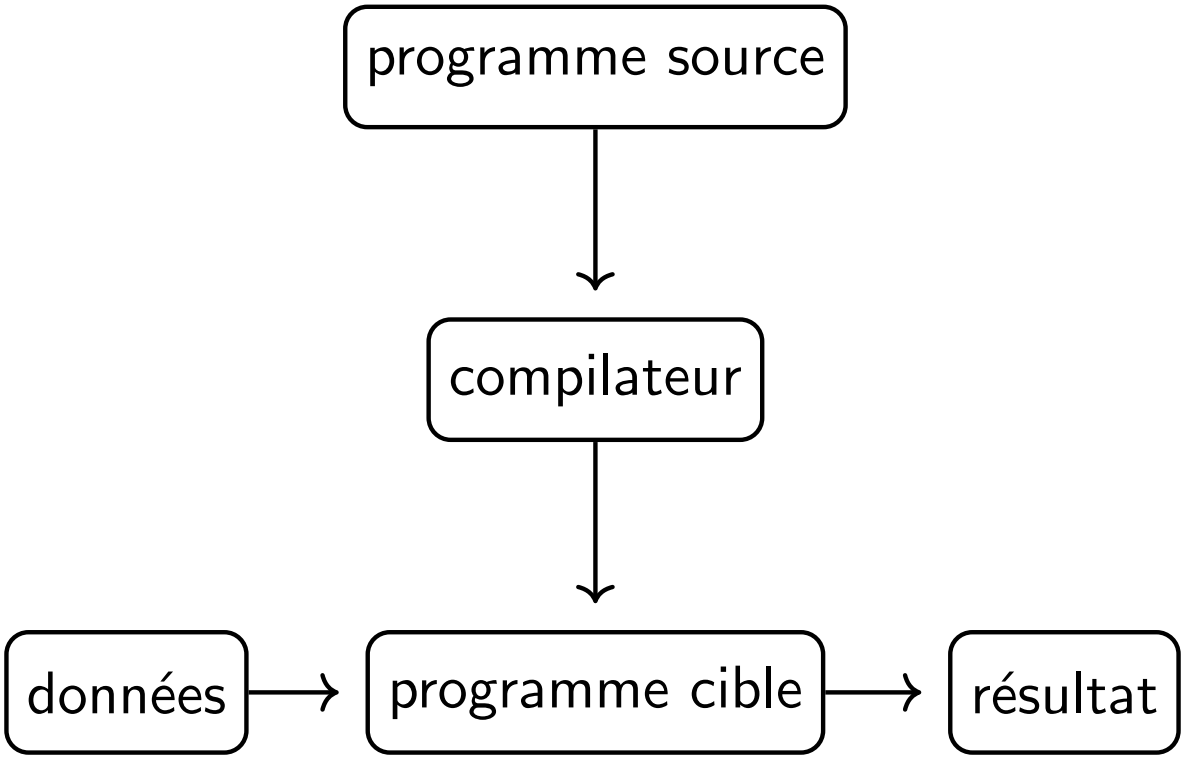
Définition (Compilateur)

Un compilateur est un programme

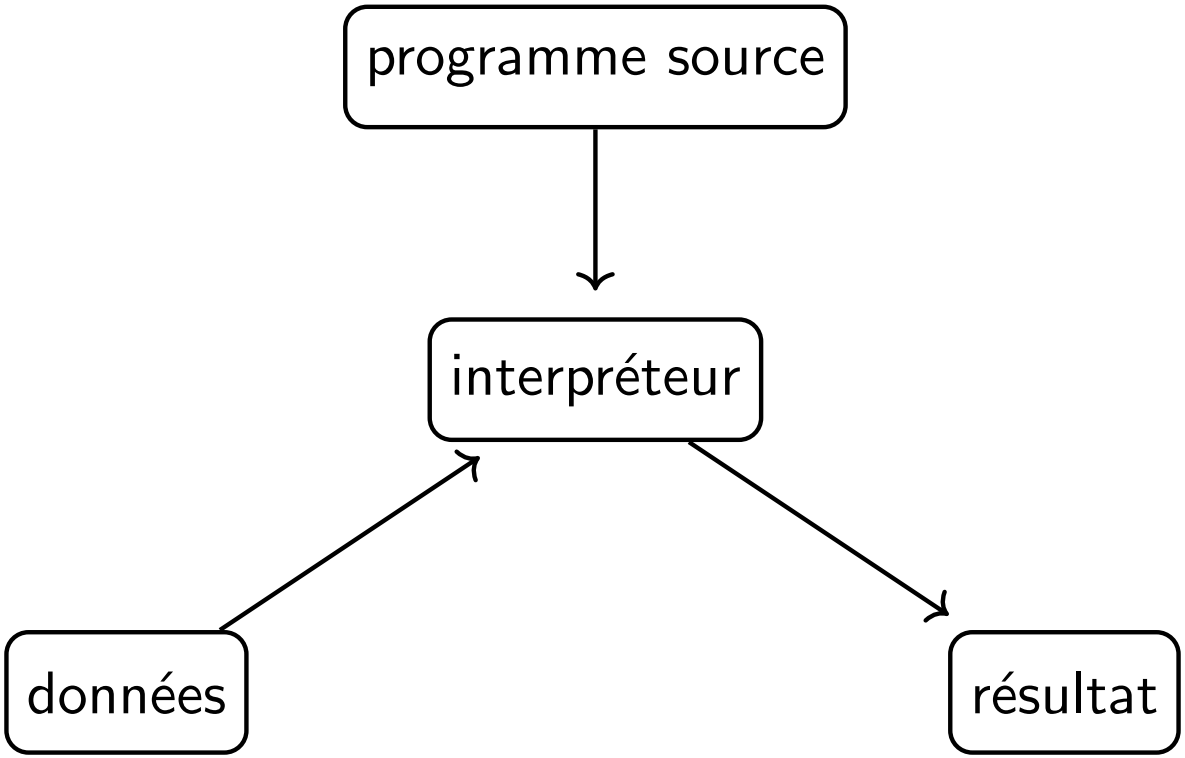
1. qui lit un autre programme rédigé dans un langage de programmation, appelé **langage source**
2. et qui le traduit dans un autre langage, le **langage cible**.

- Le compilateur signale de plus toute erreur contenue dans le programme source et dans la mesure du possible *optimise* le code obtenu.
- Lorsque le programme cible est un programme exécutable, en langage machine, l'utilisateur peut ensuite le faire exécuter afin de traiter des données et de produire des résultats.

Compilateur / interpréteur



Compilateur / interpréteur



Un interpréteur est un programme qui effectue lui-même les opérations spécifiées par le programme source directement sur les données fournies par l'utilisateur.

Structure du compilateur

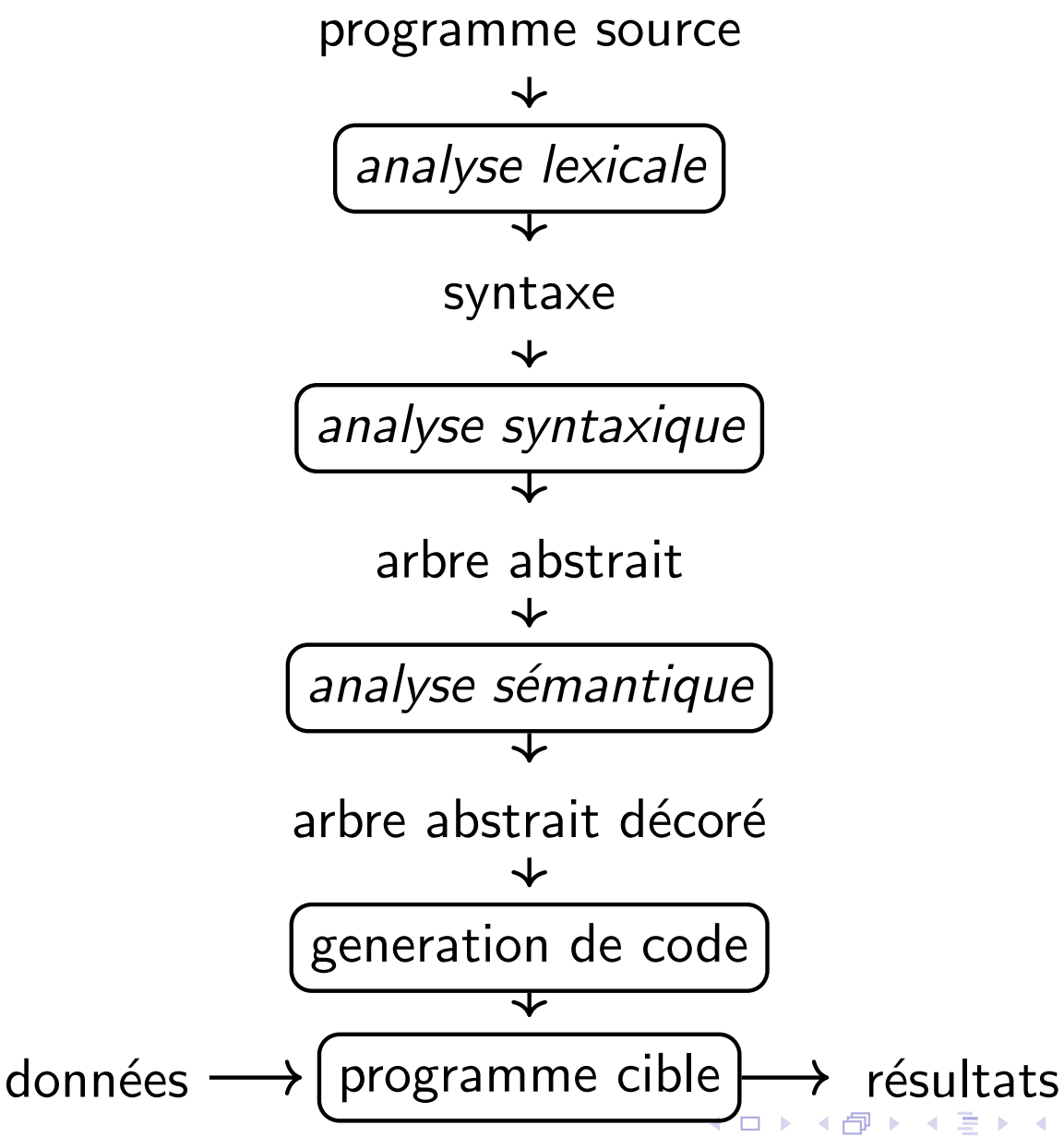


Table des matières

Outil pour le projet: `lex` et `yacc`

`lex` : générateur d'analyseur lexical.

- Prend en entrée la définition des unités lexicales sous la forme de Regex.
- Produit un automate fini déterministe minimal permettant de reconnaître les unités lexicales. que l'on peut utiliser par la suite.
- Il existe plusieurs versions de `lex`. Nous utiliserons `flex` en `C` et la classe `Lexer` de `sly` en `Python`.

`yacc` : générateur d'analyseur syntaxique.

- Prend en entrée la définition d'un schéma de traduction (grammaire + actions sémantiques).
- Produit un analyseur syntaxique pour le schéma de traduction.
- Il existe plusieurs versions de `yacc`, nous utiliserons `bison` en `c` et la classe `Parser` de `sly` en `Python`.

Exemple de grammaire avec `bison` Expressions arithmétiques :

```
/* fichier calc.y */
%token NOMBRE /* liste des terminaux */
%%
expression: expression '+' terme
          | expression '-' terme
          | terme
          ;
terme: terme '*' facteur
     | terme '/' facteur
     | facteur
     ;
facteur: '(' expression ')'
       | '-' facteur
       | NOMBRE
       ;
%%
int yyerror(void)
{ fprintf(stderr, "erreur de syntaxe\n"); return 1;}
```

Exemple d'analyse lexicale avec flex:

```
/* fichier calc.l */
%{
/* fichier dans lequel est defini
la macro constante NOMBRE */
#include "calc.tab.h"
%}

%%
[0-9]+ {return NOMBRE;}
[ \t] ; /* ignore les blancs et tabulations */
\n      return 0;
.       return yytext[0];
%%
```

Compilation de fichiers flex (.l) et bison (.y)

- `$ bison -d calc.y`
  produit les fichiers :
  - `calc.tab.c` qui contient le code en c de l'analyseur
  - et `calc.tab.h` qui contient la définition des codes des unités lexicales, afin qu'elles puissent être partagées par l'analyseur syntaxique et l'analyseur lexical.
- `$ flex calc.l`
  produit le fichier :
  - `lex.yy.c` qui contient le code en c de l'analyseur lexical.
- `$ gcc -o calc calc.c calc.tab.c lex.yy.c`
  produit l'executable :
  - `calc` qui permet d'analyser des expressions arithmétiques.

Communication

Entre les analyseurs syntaxique et lexical :

- Le fichier `lex.yy.c` définit la fonction `yylex()` qui analyse le contenu du flux `yyin` jusqu'à détecter une unité lexicale. Elle renvoit alors le code de l'unité reconnue.
- L'analyseur syntaxique (`yyparse()`) appelle la fonction `yylex()` pour connaître la prochaine unité lexicale à traiter.
- Le fichier `calc.tab.h` associe un entier à tout symbole terminal.

```
#define NOMBRE 258
```

Communication

L'analyseur lexical associe des valeurs aux unités lexicales via la variable globale yylval.

```
%{
#include "calc.tab.h"
%}

%%
[0-9]+ {yylval=atoi(yytext);return NOMBRE;}
[ \t] ; /* ignore les blancs et tabulations */
\n     return 0;
.      return yytext[0];
%%
```

flex

`flex` est un langage de spécification d'analyseurs lexicaux.

- Un programme en `flex` définit un ensemble de schémas qui sont appliqués à un flux textuel.
- Chaque schéma est représenté sous la forme d'une expression régulière.
- Lorsque l'application d'un schéma réussit, les actions qui lui sont associées sont exécutées.
- Le programme `flex` permet de transformer un programme en .l en un programme en c qui définit la fonction `yylex(void)` qui constitue l'analyseur lexical.

Syntaxe des Regex de `flex`

	Concaténation
	`ab`
`|`	Disjonction
	`a | b`
`*`	Etoile de Kleene
	`(a|b)*`
`{n}`	Répétition n fois
	`(a|b){3}`
`?`	Optionnalité (0 ou 1 fois)
	`a?`
`+`	Une fois ou plus
	`a+`
`()`	Groupement d'expressions régulières
	Modification de la priorité dans une expression
	`a(b|c)`

Syntaxe des Regex de `flex`

`.` Tout caractère excepté le retour chariot `\n`

`[]` Ensemble de caractères.

- `[abc]` définit l'ensemble {`a,b,c`}
- `[a-z]` définit l'ensemble {`a,b,c, ..., z`}
- `[a-zA-Z]` définit l'ensemble {`a,b,c, ...,z,A,B,C,...,Z`}

`[^]` Complémentaire d'un ensemble de caractères.

- `[^abc]` définit le complémentaire de l'ensemble {`a,b,c`}

`\` Caractère d'échappement

Syntaxe des Regex de `flex`

`"..."`	interprète tout ce qui se trouve entre les guillements de manière litérale `"a*$"`
`^`	Début de ligne `^abc`
`$`	Fin de ligne `abc$`
`/`	Reconnaît un motif appartenant à l'expression régulière de gauche s'il est suivi par un motif reconnu par l'expression régulière de droite `0/1` reconnaît un `0` s'il est suivi par un `1`

Structure d'un fichier `flex`

```
%{
  Partie 1 : déclarations pour le compilateur C
%}
  Partie 2 : définitions régulières
%%
  Partie 3 : règles
%%
  Partie 4 : fonctions C supplémentaires
```

Structure d'un fichier `flex`

- **La partie 1** se compose de déclarations qui seront simplement recopiées au début du fichier produit.
  On y trouve souvent une directive `#include` qui produit l'inclusion du fichier d'en tête contenant les définitions des codes des unités lexicales.
  Cette partie et les symboles %{ et %} qui l'encadrent peuvent être omis.
- **La partie 4** se compose de fonctions C qui seront simplement recopiées à la fin du fichier produit.
  Cette partie peut être absente également (les symboles %% qui la séparent de la troisième partie peuvent alors être omis.

Définitions régulières

- Les définitions régulières sont de la forme

  identificateur expressionRégulière

  où *identificateur* est écrit au **début de la ligne** et séparé de *expressionRégulière* par des blancs.

```
lettre [A-Za-z]
chiffre [0-9]
```

- Les identificateurs ainsi définis peuvent être utilisés dans les règles et dans les définitions suivantes; il faut alors les encadrer par des accolades.

```
lettre [A-Za-z]
chiffre [0-9]
alphanum {lettre}|{chiffre}
```

Règles

Les règles sont de la forme

$$expressionRégulière \{ action \}$$

où

- *expressionRégulière* est écrit au début de la ligne
- *action* est un morceau de code C, qui sera recopié tel quel, au bon endroit, dans la fonction `yylex`.

```
if                      {return SI;}
then                    {return ALORS;}
{lettre}{alphanum}*     {return IDENTIF;}
```

A la fin de la reconnaissance d'une unité lexicale, la chaîne reconnue est la valeur de la variable `yytext` de type `char *`.
La variable `yylen` indique la longueur de l'unité lexicale contenue dans `yytext`.

```
(+|-)?[0-9]+ {yylval = atoi(yytext); return NOMBRE;}
```

Ordre d'application des règles

- `flex` essaye les règles dans leur ordre d'apparition dans le fichier.
- La règle reconnaissant la séquence la plus longue est appliquée.

  ```
  a  {printf("1");}
  aa {printf("2");}
  ```

  l'entrée `aaa` provoquera la sortie 21
- Lorsque deux règles reconnaissent la séquence la plus longue, la première est appliquée.

  ```
  aa {printf("1");}
  aa {printf("2");}
  ```

  l'entrée `aa` provoquera la sortie 1
- Les caractères qui ne sont reconnus par aucune règle sont recopiés sur la sortie standard.

  ```
  aa {printf("1");}
  ```

  l'entrée `aaa` provoquera la sortie 1a

Quelques variables importantes

- `char *yytext` chaîne de caractères dans laquelle est stockée la suite de caractères qui correspond à l'expression régulière d'une règle.
- `int yyleng` longueur de la chaîne de caractères stockée dans `yytext`.
- `int yylval` valeur associée à une unité lexicale.
- `FILE *yyin` flux dans lequel `yylex` lit le texte à analyser.

Un exemple

```
%{
#include "syntaxe.tab.h"
%}
%%
si              {return SI;}
alors           {return ALORS;}
sinon           {return SINON;}
\$[A-Za-z]+     {return ID_VAR;}
[0-9]+          {yylval.ival=atoi(yytext);return NOMBRE;}
[A-Za-z]+       {return ID_FCT;}
%%
```

La fonction yywrap(void)

- Lorsque `yylex()` atteint la fin du flux `yyin`, il appelle la fonction `yywrap` qui renvoie `0` ou `1`.
- Si la valeur renvoyée est `1`, le programme s'arrête.
- Si la valeur renvoyée est `0`, l'analyseur lexical suppose que `yywrap` a ouvert un nouveau fichier en lecture et le traitement de `yyin` continue.
- Version minimaliste de `yywrap`

```
int yywrap(void)
{
  return 1;
}
```

Découverte de `sly`

Démonstration sur `analyse_lexicale.py`.

Table des matières

Objectif du projet

- concevoir un compilateur du langage *FLO* vers le langage assembleur nasm (lui-même compilé pour être utilisé comme programme exécutable)
- Langage très fortement suggéré: `python` (avec sly) ou c (avec flex+bison).
- Projet à réaliser en binôme.
- Le langage *FLO* est un langage de programmation minimaliste inventé spécialement pour ce cours.

Le langage *FLO*

Programmation impérative, typage fort comme en *C*.

quelques caractéristiques :

Types : Le langage *FLO* connaît deux types de variables :

- Les entiers
- Les booléens

Fonctions Le langage *FLO* permet de décrire des fonctions à valeurs entière ou booléenne.

Opérateurs Le langage *Flo* connaît les opérateurs suivants :

- arithmétiques : `+`, `-`, `*`, `/` (division entière), `%` (modulo)
- comparaison : $==$, $!=$, $<$, $>$, $<=$, $>=$
- logiques : `et`, `ou`, `non`

Entrée/sortie Le langage *FLO* a deux opérations d'entrée (`lire()` et de sortie (`ecrire(nombre)`).

Assembleur NASM

Assembleur pour les processeurs X86.

- Mémoire séparée en trois parties :
  - l'espace global (variables)
  - la zone de code,
  - la pile.
- En plus, registres utilisés dans la plupart des instructions.
  - registres courants : `eax-edx`,
  - pointeur de sommet de pile : `exp` ...

NASM — exemples

`mov eax [k]`	charge la variable à l'adresse `k` dans `eax`
`mov ebx 1`	charge la valeur `1` dans `ebx`
`add eax, ebx`	addition : `eax = eax + ebx`
`jmp label`	saute à l'adresse label dans le code