Compilation

Prof. Youness Tabii

Plan

- Introduction et rappel
- Analyseur lexical
- Analyseur syntaxique
- Analyseur sémantique
- Représentation intermédiaire
- Génération de code
- Optimisation

Introduction

- Les micro-processeurs deviennent indispensables et sont embarqués dans tous les appareils que nous utilisons dans la vie quotidienne
  - Transport
    - Véhicules, Systèmes de navigation par satellites (GPS), Avions, …
  - Télécom
    - Téléphones portables, Smart Phones, …
  - Électroménager
    - Machine à laver, Micro-ondes, Lave vaisselles, …
  - Loisir
    - e-book, PDA, Jeux vidéo, Récepteurs, Télévision, TNT, Home Cinéma…
  - Espace
    - Satellites, Navettes, Robots d'exploration, …
- Pour fonctionner, ces micro-processeurs nécessitent des programmes spécifiques à leur architecture matérielle

Introduction

- Programmation en langages de bas niveau (proches de la machine)
  - **très difficile** (complexité)
    - Courbe d'apprentissage très lente
    - Débuggage fastidieux
  - **très coûteuse en temps** (perte de temps)
    - Tâches récurrentes
    - Tâches automatisables
  - **très coûteuse en ressource humaine** (budget énorme)
    - Tâches manuelles
    - Maintenance
  - **très ingrate** (artisanat)
    - Centrée sur les détails techniques et non sur les modèles conceptuels
  - **n'est pas à 100% bug-free** (fiabilité)
    - Le programmeur humain n'est pas parfait

Introduction

- Besoin continu de
  - Langages de haut niveau
    - avec des structures de données, de contrôles et des modèles conceptuels proches de l'humain
  - Logiciels de **génération** des **langages bas niveau** (propres aux microprocesseurs) à partir de **langages haut niveau** et vice versa
    - **Compilateurs**
    - **Interpréteurs**
    - **Pré-processeurs**
    - **Dé-compilateur**
    - **etc.**

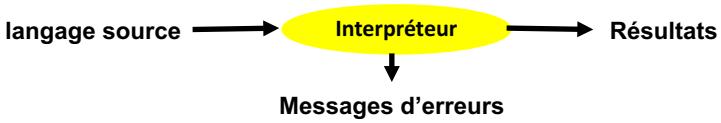
Compilateur - Définition

- Un compilateur est un logiciel (une *fonction*) qui prend en entrée un programme **P1** dans un langage source L1 et produit en sortie programme équivalent **P2** dans un langage L2

- Exemple de compilateurs :
  - C pour Motorola, Ada pour Intel, C++ pour Sun, doc vers pdf, ppt vers Postscript, pptx vers ppt, Latex vers Postscript, etc.

Interpréteur - Définition

- Un interpréteur est logiciel qui prend en entrée un programme **P1** dans un langage source L1 et produit en sortie les **Résultats** de l'exécution de ce programme



- Exemples d'interpréteurs :
  - Batch DOS, Shell Unix, Prolog, PL/SQL, Lisp/Scheme, Basic, Calculatrice programmable, etc.

Dé-compilateur

- Dé-Compilateur est un compilateur dans le sens inverse d'un compilateur (depuis le langage bas niveau, vers un langage haut niveau)
- Applications :
  - Récupération de vieux logiciels
    - Portage de programmes dans de nouveaux langages
    - Recompilation de programmes vers de nouvelles architectures
  - Autres
    - Compréhension du code d'un algorithme
    - Compréhension des clefs d'un algorithme de sécurité

Questions

- Quels sont les constituants d'un langage naturel ?
- Si l'on veut programmer un traducteur de l'Anglais vers le Français que proposeriez-vous comme algorithme ?

Quelques éléments de réponses

- Quelques constituants d'un langage naturel ?
  - Vocabulaire (lexique), Syntaxe (grammaire), Sémantique (sens selon le contexte)
- Un macro-algorithme pour traduire de l'Anglais vers le Français que proposeriez-vous comme ?
  1. Analyser les mots selon le dictionnaire Anglais
  2. Analyser la forme des phrases selon la grammaire de l'Anglais
  3. Analyser le sens des phrases selon le contexte des mots dans la phrase anglaise
  4. Traduire le sens des phrases dans la grammaire et le vocabulaire du Français

Étapes et Architecture d'un compilateur

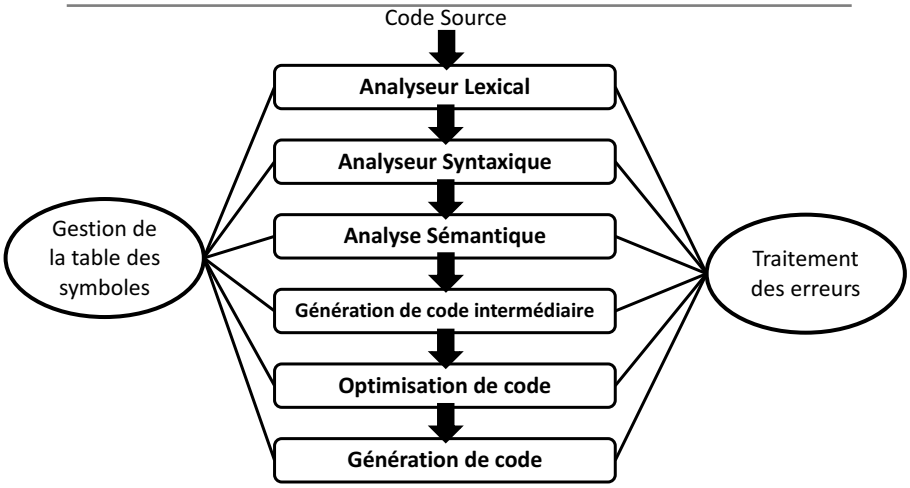
1. Compréhension des entrées (**Analyse**)
   - Analyse des éléments lexicaux (lexèmes : mots) du programme : **analyseur lexical**
   - Analyse de la forme (structure) des instructions (phrases) du programme : **analyseur syntaxique**
   - Analyse du sens (cohérence) des instructions du programme : **analyseur sémantique**
2. Préparation de la génération (**Synthèse**)
   - Génération d'un code intermédiaire (nécessitant des passes d'optimisation et de transcription en code machine) : **générateur de pseudo-code**
   - Optimisation du code intermédiaire : **optimisateur de code**
3. Génération finale de la sortie (**Synthèse suite**)
   - Génération du code cible à partir du code intermédiaire : **générateur de code**

Étapes et Architecture d'un compilateur

✓ Autres Fonctionnalités

- gestion les erreurs et guide le programmeur pour corriger son programme : **gestionnaire d'erreurs**
- gestion du dictionnaire des données du compilateur (symboles réservés, noms des variables, constantes) : **gestionnaire de la table des symboles**

Architecture d'un compilateur

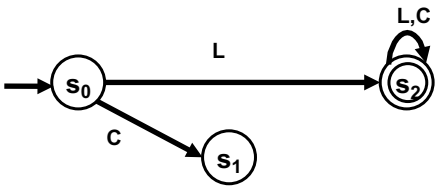


Rappel

Comment Transcrire une expression régulière en langage C ?

Expression : **L(L+C)*** Avec **L=[a-zA-Z]** et **C=[0-9]**

Rappel
ER → DFA minimal → Programme en C



Etat mort/Trash

Rappel
ER → DFA minimal → Programme en C

```
typedef  enum {S0, S1, S2} State;

int main(int argc,char *argv[]){
   // 1- traitement intial
   State state = S0;

   printf("S0\n");

   automate(state);

   return 0;
}
```

```
void automate(State current_state){
    State state = current_state;
    char c = getchar();
    if (c != '$') {
    // 2- traitement récursif
    switch (state){
            case S0 :
                if (((c >= 'a') && (c <='z')) || ((c >= 'A') && (c <='Z'))){
                    state = S1; printf("S0 --> S1\n");
                }else {state = S2; printf("S0 --> S2\n");}
                break;
            case S1 :
                if (((c >= 'a') && (c <='z')) || ((c >= 'A') && (c <='Z')) || ((c
     >= '0') &&  (c <= '9')))
                 state = S1; printf("S1 --> S1\n");
                break;
            case S2 :
                state = S2;  printf("S2 --> S2\n");
                }
    automate(state);
    }else{
   // 3- traitement final
   if (state == S1) printf("mot accepté par l'automate des identificateurs\n");
   else printf("mot refusé par l'automate des identificateurs\n");}}
```


Rappel
ER → DFA minimal → Programme en C

./idAutomate

state0 **a123456**$

state0 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

mot accepté par l'automate des identificateurs

./idAutomate

state0 **12345**$

state0 --> state2

state2 --> state2

state2 --> state2

state2 --> state2

state2 --> state2

mot refusé par l'automate des identificateurs

Analyse Lexical

Définitions

- Le lexique d'un langage de programmation est son vocabulaire.
- Un lexème (token) est une collection de symboles élémentaires (un mot) ayant un sens pour le langage.
- Plusieurs lexèmes peuvent appartenir à une même classe. Une telle classe s'appelle une unité lexicale.
- L'ensemble des lexèmes d'une unité lexicale est décrit par une règle appelée modèle associé à l'unité lexicale.

Analyseur Lexical

- Le module qui effectue l'analyse lexicale s'appel un analyseur lexical (lexer ou scanner).
- Un analyseur lexical prend en entrée une séquence de caractères individuels et les regroupe en lexèmes.
- Autres tâches d'un analyseur lexical
  - Ignorer tous les éléments n'ayant pas un sens pour le code machine (Les espaces, les tabulations, les retours à la ligne, et les commentaires).

Définition

- **Unités lexicales (Tokens)**:
  - Symboles : identificateurs, chaînes, constantes numériques
  - Mots clefs : while, if, case, …
  - Opérateurs : <=, =, ==, +, -, …

Analyseur Lexical
Exemple

Input : a = b+c*2;

Ouput :

IDENTIFICATEUR
OPPAFFECT
IDENTIFICATEUR
PLUS
IDENTIFICATEUR
OPPMULT
CONSTANTE
PTVIRG

Tokens	Unité Lexical
a	Identificateur
=	Opérateur Affectation
b	Identificateur
+	Opérateur Plus
c	Identificateur
*	Opérateur Multiplication
2	Constante

Analyseur Lexical

✓ Les meilleurs modèles qui existent pour identifier les types lexicaux de tokens sont les expressions régulières

- **Alphabétique : (‘a’ | ‘b’ | … | ‘z’ | ‘A’ | ‘B’ | … | ‘Z’)**
- **Numérique : (0 | …| 9)**
- **Opérateurs : (+ | - | / | * | = | <= | >= | < | >)**
- **Naturel : Numérique+**
- **Entier : (+ | - | ε) Naturel**
- **Identificateur : (Alphabétique (Alphabétique | Numérique)*)**
- **ChaîneAlphaNumérique : " (Alphabétique | Numérique)+ "**

Analyseur Lexical

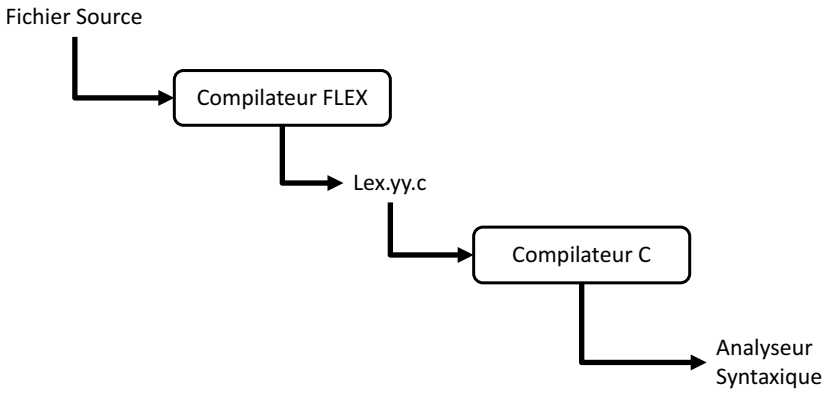
Il existe des algorithmes et outils permettant de générer le code implantant l'automate d'états finis (scanner) correspondant à une expression régulière (Lex)

Générateur de scanner - FLEX

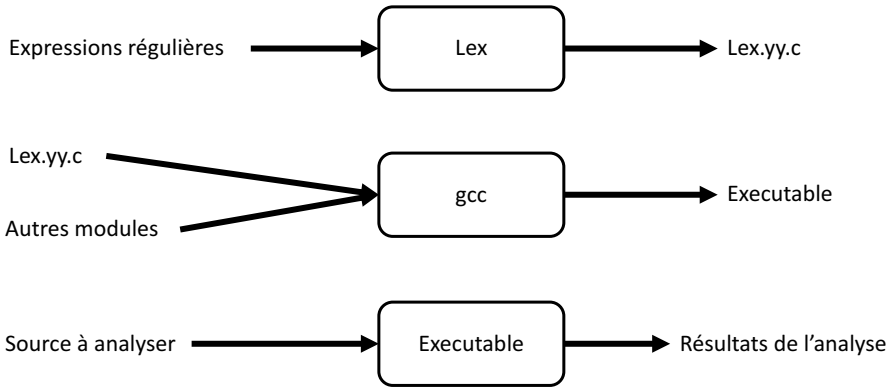
FLEX (FAST LEX)

- Flex est un outil de génération automatique d'analyseurs lexicaux.
- Un fichier Flex contient la description d'un analyseur lexical à générer.
  - Cette description est donnée sous la forme d'expressions régulières étendues et du code écrit en langage C (ou C + +).
- Flex génère comme résultat un fichier contenant le code C du future analyseur lexical (nommé lex.yy.c).
- La compilation de ce fichier par un compilateur C, génère finalement le code exécutable de l'analyseur lexical en question.

FLEX (FAST LEX)



FLEX
Etapes



FLEX

- Lorsque l'exécutable est mis en œuvre, il analyse le fichier source pour chercher les occurrences d'expressions régulières.
- Lorsqu'une régulière est trouvée, il exécute le code C correspondant.

Structure d'un programme FLEX

```
%{
/* Déclarations */
%}
/* Définitions */
%%
/* Règles */
%%
/* Code utilisateur */
```

- Déclaration/Code utilisateur: du C tout à fait classique
- Définitions : Expressions rationnelles auxquelles on attribue un nom
- Règles de production : Associations ER → code C à exécuter

Variables FLEX

- Dans les actions, on peut accéder à certaines variables spéciales :
  - **yylex()** : est la fonction principale du programme LEX.
  - **yyleng** : contient la taille du *token* reconnu ;
  - **yytext** : est une variable de type char* qui pointe vers la chaîne de caractères reconnue par l'expression régulière.
  - **yylval** : qui permet de passer des valeurs entières à YACC.
  - Il existe aussi une action spéciale : **ECHO** qui équivaut à **printf("%s",yytext).**
  - **yyin** : entrée du scanner
  - **yyout** : sortie du scanner

Expression régulière FLEX

Symbole	Signification
x	Le caractere 'x'
.	N'importe quel caractere sauf \n
[xyz]	Soit x, soit y, soit z
[^bz]	Tous les caracteres, SAUF b et z
[a-z]	N'importe quel caractere entre a et z
[^a-z]	Tous les caracteres, SAUF ceux compris entre a et z
R*	Zero R ou plus, ou R est n'importe quelle expression reguliere
R+	Un R ou plus
R?	Zero ou un R (c'est-a-dire un R optionnel)
R{2,5}	Entre deux et cinq R
R{2,}	Deux R ou plus
R{2}	Exactement deux R
"[xyz\"foo"	La chaine '[xyz"foo'
{NOTION}	L'expansion de la notion NOTION definie plus haut
\X	Si X est un 'a', 'b', 'f', 'n', 'r', 't', ou 'v', represente l'interpretation ANSI-C de \X.
\0	Caractere ASCII 0
\123	Caractere ASCII dont le numero est 123 EN OCTAL
\x2A	Caractere ASCII en hexadecimal
RS	R suivi de S
R|S	R ou S
R/S	R, seulement s'il est suivi par S
^R	R, mais seulement en debut de ligne
R$	R, mais seulement en fin de ligne
<<EOF>>	Fin de fichier

FLEX
Commandes

- Les étapes à suivre pour obtenir un analyseur avec Flex :

1. Ecrire votre analyseur dans un fichier portant une extension **".l"** ou **".lex"**. Par exemple, "**prog.lex**".
2. Compiler votre analyseur par la commende flex : **flex prog.lex**
3. Compiler le fichier par la commande gcc le fichier lex.yy.c produit par l'étape précédente :
   gcc lex.yy.c –lfl –o prog
4. Lancer l'analyseur en utilisant le nom de celui-ci : **prog**

Exemple

```
/*reconnaître les chiffres, lettres, réels et identificateurs*/
%{
/* Déclaration VIDE */
%}
%option noyywrap

/* Définition */
chiffre [0-9]
lettre  [a-zA-Z]
entier {chiffre}+
reel {chiffre}+(\.{chiffre}+(e(\+|\-)?{chiffre}+)?)?
ident {lettre}({lettre}|{chiffre})*

%%
/* Régles exrp → action*/
{entier} printf("\n entier %s \n ", yytext);
{reel}   printf("\n reel %s \n ", yytext);
{ident}  printf("\n identificateur %s \n ", yytext);
%%

int main(void){
        yylex();
        return 0;}
```

12 → entier 12

3.4→ reel 3.4

L → identificateur L

6.5e+4 → reel 6.5e+4

Exercice 1

- Compte le nombre de Voyelles, Consonnes et les caractères de ponctuation d'un texte entré en clavier.

- Etapes :
  - Déclaration des compteurs
  - Définition des expressions régulières (les consonnes, voyelles et ponctuation)
  - L'action à exécuté pour chaque expression régulière
  - La fonction main (pour affichage des résultat)

Solution

```
%{
int nbConsonnes,nbVoyelles,nbPonctuation;
%}

%option noyywrap
consonne [b-df-hj-np-xz]
ponctuation    [,;:?!\.]

%%

[aeiouy] nbVoyelles++;
{consonne} nbConsonnes++;
{ponctuation} nbPonctuation++;
.|\n     // ne rien faire

%%

int main(void){
        nbConsonnes=0;
        nbVoyelles=0;
        nbPonctuation=0;
        yylex();
        printf("\n Nb Consonnes : %d,
                    Nb Voyelles : %d,
                    Nb Poncuation : %d \n",
                    nbConsonnes,nbVoyelles,nbPonctuation);
       return 0;
        }
```

$ bonjour,

Nb Consonnes : 4, Nb Voyelles : 3, Nb Ponctuation : 1

Fin Séance