Compilation

2025-2026

Par : Morou Ganda Yahaya

Email : y.morouganda.35116@gmail.com

Niveau : Licence 3

Volume horaire : 48H

Références

1. Aho A., Sethi R., Ullman J., "Compilateurs : Principes, techniques et outils", Inter-éditions, 1991 et Dunod, 2000
2. Drias H., "Compilation: Cours et exercices", OPU, 1993
3. Wilhem R., Maurer D., "Les compilateurs: Théorie, construction, génération", Masson, 1994
4. Henri Garreta, "Techniques et outils pour la compilation", Janvier 2001
5. Noureddine Chenfour, "Principes et Techniques de Compilation", Université Sidi Mohamed Ben Abdellah Faculté des Sciences Dhar El-Mehraz Fès Département d'Informatique, 2010.

I. Introduction

1. Un peu d'histoire

Les logiciels des premiers ordinateurs étaient écrits en langage assembleur. La capacité de mémoire très limitée des premiers ordinateurs a posé plusieurs problèmes techniques dans le développement des compilateurs. C'est Vers la fin des années 1950 que des langages de programmation indépendants des machines font pour la première fois leur apparition. Par la suite, plusieurs compilateurs expérimentaux sont développés. Le premier compilateur, A-0 System était écrit par Grace Hopper en 1952 qui créa ensuite le langage COBOL (premier langage à être compilé sur plusieurs architectures). En 1962, le premier compilateur « auto-hébergé » capable de compiler en code objet, son propre code source exprimé en langage de haut niveau est créé, pour le Lisp, par Tim Hart et Mike Levin au Massachusetts Institute of Technology (MIT). A partir des années 1970, il est devenu très courant de développer un compilateur dans le langage qu'il doit compiler, faisant du Pascal et du C des langages de développement très populaires.

2. Structure et principe d'un compilateur

Un compilateur est un programme qui a comme entrée un code source écrit en langage de haut niveau (langage évolué, langage C par exemple) est produit comme sortie un code cible en langage de bas niveau (langage d'assemblage ou langage machine appelé aussi langage de bas niveau). Dans le sens le plus usuel du terme, la compilation est une transformation que l'on fait subir à un programme écrit dans un langage évolué pour le rendre exécutable. En généralisant un peu, on peut dire que compiler c'est lire une suite de caractères obéissant à une certaine syntaxe, en construisant une autre représentation de l'information que ces caractères expriment. De ce point de vue, beaucoup d'opérations apparaissent comme étant de la compilation.

Exemple : la lecture d'un nombre, qu'on obtient en C par une instruction comme « scanf("%f", &x) » est déjà de la compilation, puisqu'il s'agit de lire des caractères constituant l'écriture d'une valeur selon la syntaxe des nombres décimaux et de fabriquer une autre représentation de la même information, à savoir sa valeur numérique.

La traduction ne peut être effectué que si le code source est correct car, s'il y a des erreurs, le rôle du compilateur se limitera à produire en sortie des messages d'erreurs comme illustré sur la figure suivante :

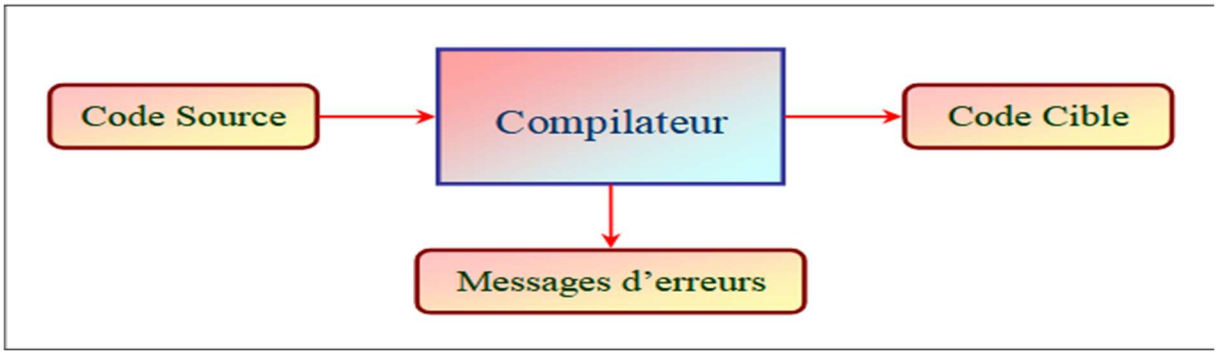


Figure 1.1 : rôle d'un compilateur

Un compilateur est donc un traducteur de langage évolué qu'on ne doit pas confondre avec un interpréteur qui est un autre type de traducteur. En effet, au lieu de produire un programme cible comme dans le cas d'un compilateur, un interpréteur exécute lui-même au fur et à mesure les opérations spécifiées par le programme source. Il analyse une instruction après l'autre puis l'exécute immédiatement. Des exemples de langages interprétés sont : BASIC, scheme, CaML, Tcl, LISP, Perl, Prolog.

La nature de ce qui sort d'un compilateur est très variable. Cela peut être un programme exécutable pour un processeur physique, comme un Pentium III ou un G4, ou un fichier de code pour une machine virtuelle, comme la machine Java, ou un code abstrait, destiné à un outil qui en fera ultérieurement du code exécutable, ou encore le codage d'un arbre représentant la structure logique d'un programme, etc. En entrée d'un compilateur on trouve toujours la même chose : une suite de caractères, appelée le texte source. La structure d'un compilateur est généralement composée de modules correspondant aux phases logiques de l'opération de compilation.

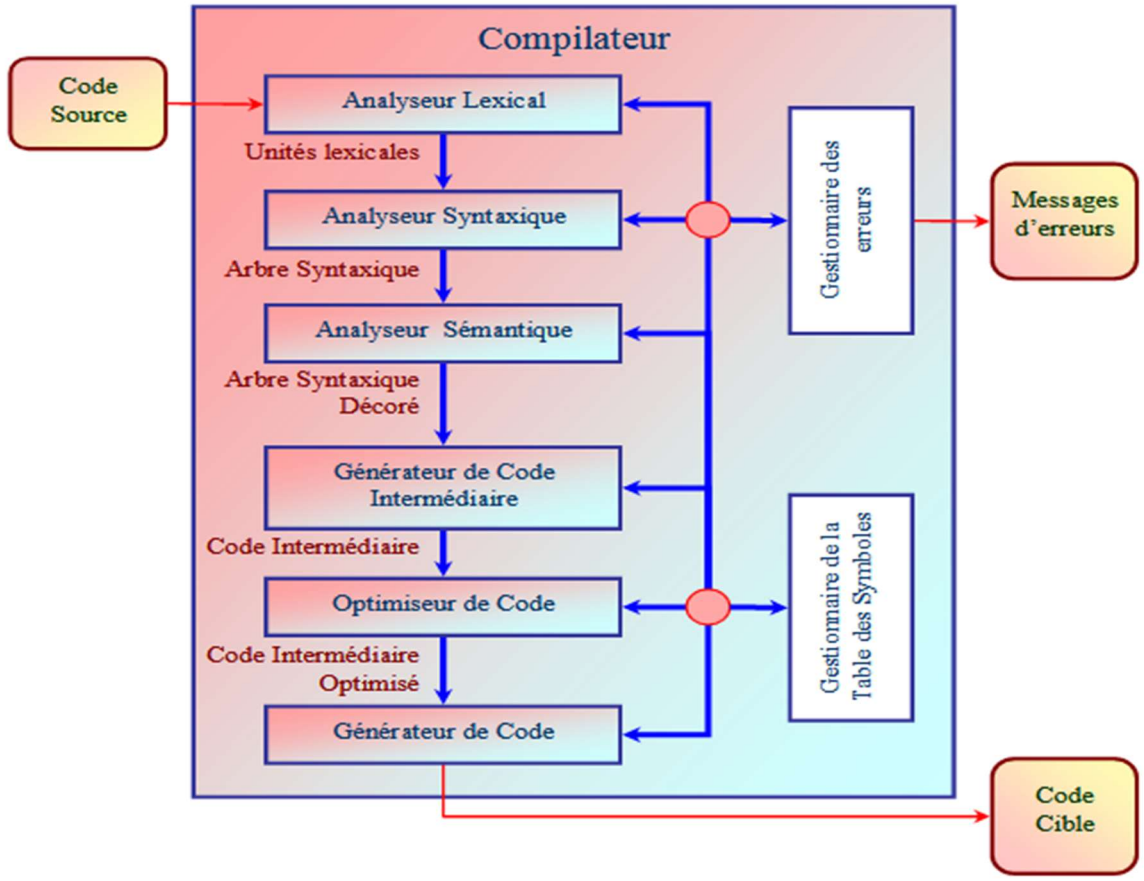


Figure 1.2 : structure d'un compilateur

II. Analyse lexicale

L'analyse lexicale est la première phase de la compilation. Dans le texte source, qui se présente comme un flot de caractères, l'analyse lexicale reconnaît des unités lexicales, qui sont les « mots » avec lesquels les phrases sont formées, et les présente à la phase suivante, l'analyse syntaxique. Il s'agit donc de transformer des suites de caractères du code source en suite de symboles, correspondant aux unités lexicales. Pour cela, il existe deux approches possibles sachant qu'une *passe* est définie comme étant un traitement entier d'une représentation du programme source à une représentation équivalente en mémoire secondaire :

- L'analyseur lexical effectue un traitement de la totalité du code source en une première passe, ce qui lui permet d'en obtenir une représentation équivalente sous forme d'une suite d'unités lexicales, sauvegardée dans un fichier en mémoire secondaire. Ce

fichier sera l'entrée de l'analyseur syntaxique, qui accomplira son travail pendant une deuxième passe. Le schéma de la figure 2.1 illustre cette approche.

- L'analyseur lexical fonctionne comme une procédure sollicitée à chaque fois, par l'analyseur syntaxique, pour lui fournir une unité lexicale. L'analyseur lexical effectue, pour cela, son travail, pas à pas, en synchronisation avec l'avancement de l'analyse syntaxique. On dit que l'analyse lexicale et syntaxique partagent une même passe. Dans ce cas, il n'est plus question de sauvegarder les unités dans un fichier intermédiaire, cependant, les deux analyseurs (lexical et syntaxique) doivent se trouver ensemble en mémoire. Cette approche est la plus utilisée dans la conception des compilateurs, elle est illustrée par la figure 2.2

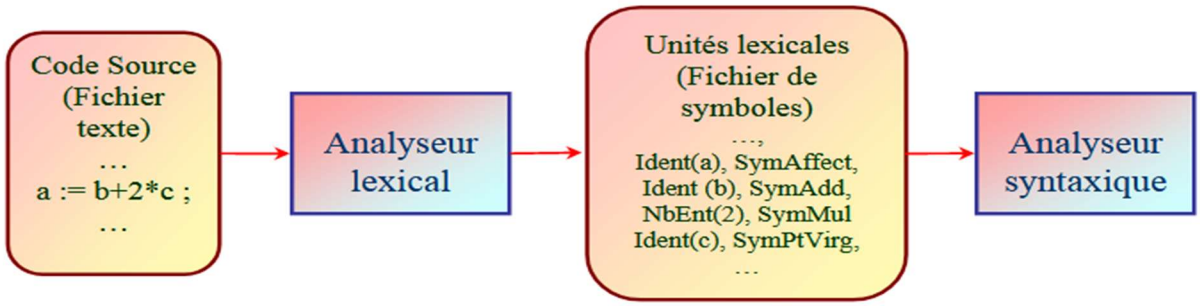


Figure 2.1 : Analyse lexicale et syntaxique en deux passes différentes

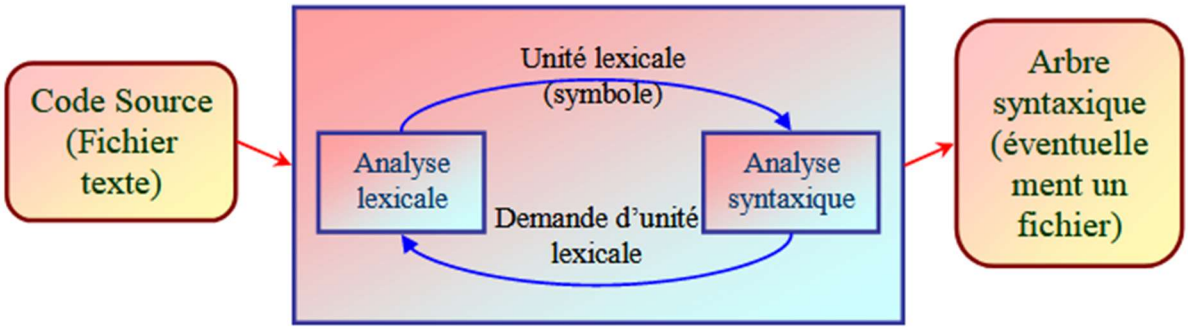


Figure 2.2 : Analyses lexicale et syntaxique en une seule passe

Les principales sortes d'unités lexicales qu'on trouve dans les langages de programmation courants sont :

- Les caractères spéciaux simples : +, =, etc.
- Les caractères spéciaux doubles : <=, ++, etc.

- Les mots-clés : if, while, etc.
- Les constantes littérales : 123, -5, etc.
- Et les identificateurs : i, vitesse_du_vent, etc.

L'ensemble des unités lexicales, qu'un analyseur reconnaît, constitue un langage régulier L qui peut être décrit par des expressions régulières et reconnu par un automate d'états fini.

1. Expressions régulières

Les expressions régulières sont une importante notation pour spécifier formellement des modèles. Pour les définir correctement il nous faut apprendre les définitions de quelques notions :

- **Un alphabet** est un ensemble de symboles. Exemples : {0, 1}, {A, B, C, N}, l'ensemble de toutes les lettres, l'ensemble des chiffres, le code ASCII, etc.
- **Une chaîne** (on dit aussi *mot*) sur un alphabet Σ est une séquence finie de symboles de Σ. Exemples : respectivement relatifs aux alphabets précédents : 00011011, AABCNNBA, Bonjour, 2022. On note ε la chaîne vide, ne comportant aucun caractère.
- **Un langage** sur un alphabet Σ est un ensemble de chaînes construites sur Σ. Exemple : pour Σ = {0, 1}, l'ensemble des nombres en notation binaire est un langage.
- Soit le vocabulaire (alphabet) Σ = {a, b, c}
  1. ε dénote la chaîne vide
  2. L'expression régulière a/b dénote l'ensemble des chaînes {a, b}
  3. L'expression a* dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque de a (pouvant être nul), c à d {ε, a, aa, aaa, ...}
  4. L'expression a+ dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque, non nul, de a, c à d {a, aa, aaa, ...}. Si r est une expression régulière alors r* = r+|ε et r+ = rr*
  5. La notation r? est une abréviation de r|ε et signifie zéro ou une instance de r
  6. La notation [abc], où a, b, c sont des symboles de l'alphabet, dénote l'expression régulière a/b/c. Une classe de caractères [a-z] signifie a/b/c/d/ .../z

Exemple

Pour chacune des expressions régulières r_i suivantes, on souhaite déterminer le langage dénoté par r_i.

$r_1 = (a|b)(a|b)$, $r_2 = (a|b)^*$, $r_3 = (a^*b^*)^*$, $r_4 = a|a^*b$

Correction : Chaque langage dénoté par ri est noté $L(r_i)$

- $L(r_1) = \{aa, ab, ba, bb\}$
- $L(r_2) = \{\varepsilon, a, aa, aaa, ..., b, bb, bbb, ..., ab, aab, ..., abb, abbb, ba, bba, bbaa, aba, ...\}$. C'est l'ensemble de toutes les chaînes composées d'un nombre quelconque de a et d'un nombre quelconque de b.
- $L(r_3) = L(r_2)$
- $L(r_4) = \{a, b, ab, aab, aaab, ..., aaaa...ab\}$ En plus de la chaîne a, cet ensemble contient des chaînes composées d'un nombre quelconque non nul de a suivis par un b.

Remarque

Les expressions régulières sont un outil puissant et pratique pour définir les constituants élémentaires des langages de programmation. Cependant, le pouvoir descriptif des expressions régulières est limité car elles ne peuvent pas être utilisées dans certains cas tels que :

- La définition des constructions équilibrées ou imbriquées. Par exemple, pour les chaînes contenant obligatoirement pour chaque parenthèse ouvrante une parenthèse fermantes, les expressions régulières ne permettent pas d'assurer cela, on dit *qu'elles ne savent pas compter*.
- La définition d'expressions contenant des répétitions de sous chaînes à différente positions (par exemple des chaînes de la forme $\alpha c \alpha d e \alpha$ où α est une combinaison des caractères a et b).
- La définition d'expressions contenant à différentes positions des séquences de caractères ayant la même longueur (par exemple $a_n b_n$ où n désigne le nombre de répétitions du caractère qui le précède).

On peut, cependant, utiliser les expressions régulières dans le cas d'un nombre fixe de répétitions d'une construction donnée.

2. Analyse lexicale

Après la description des unités lexicales, la construction de l'analyseur lexical peut être effectuée selon l'une des manières suivantes :

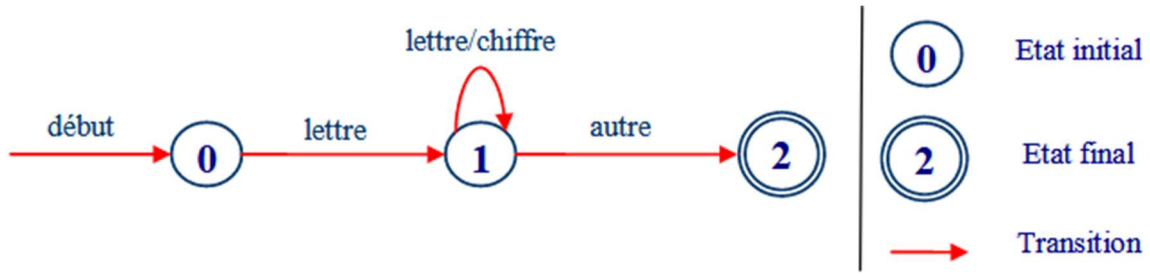
- Une approche simplifiée basée sur les diagrammes de transition.
- Une approche plus rigoureuse basée sur les automates d'états finis.
- Une approche utilisant un outil générateur d'analyseurs lexicaux tel que Lex, Flex…

a. Construction d'analyseur lexical basée sur les diagrammes de transition

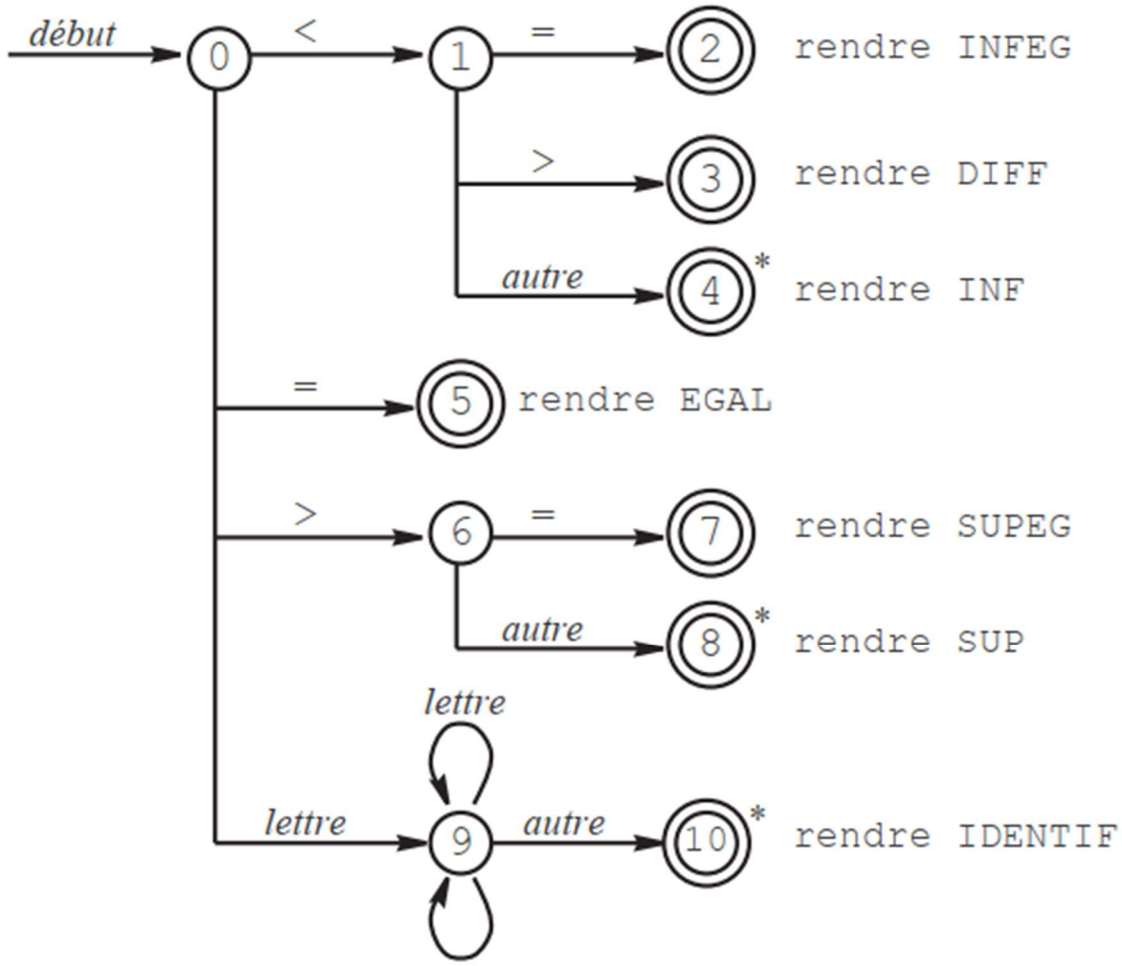
Les diagrammes de transition sont des organigrammes stylisés (respectant une forme stricte) dont la construction représente une étape préparatoire pour la réalisation d'un analyseur lexical. Le diagramme est ensuite converti en un programme qui permet de reconnaître les unités lexicales exprimées par le diagramme. Formellement, un diagramme de transition est un diagramme contenant des cercles numérotés qui représentent des états et des arcs étiquetés matérialisant les transitions entre les états. Les états du diagramme de transition correspondent aux états de l'analyseur lexical et chaque arc étiqueté par un caractère donné correspond à la transition qu'effectuera l'analyseur suite à la rencontre de ce caractère. Ainsi, au fur et à mesure de la reconnaissance d'une unité lexicale, l'analyseur passe d'un état à un autre. Nous avons donc les notations suivantes :

- Un état particulier du diagramme représentera l'état initial où commence la reconnaissance d'une unité lexicale. Il est noté par un état auquel abouti un arc étiqueté par **début**.
- Les états finaux sont identifiés par deux cercles concentriques (cercle doublé) et correspondent, chacun, à la reconnaissance complète d'une unité lexicale.
- Si un ensemble d'arcs sortant d'un état *e1* comprend un arc ayant l'étiquette autre, elle désigne tous les caractères autres que ceux indiqués explicitement sur les autres arcs sortant de *e1*.

Exemple 1 : Le diagramme de transition de la figure suivante, correspond à l'expression régulière lettre(lettre/chiffre)*.



Exemple 2 : Diagramme de transition pour les opérateurs de comparaison et les identificateurs contraire.



D'une manière générale, une suite de diagrammes de transition peut être convertie en un programme qui recherche les unités lexicales selon l'approche suivante :

- Chaque état donne lieu à un segment de programme.
- S'il y a des arcs qui quittent un état, son segment de programme doit inclure la lecture d'un caractère et son test pour sélectionner un arc à suivre.
- S'il y a un arc étiqueté par le caractère lu, ou par une classe de caractères contenant le caractère lu, alors le contrôle est transféré au code correspondant à l'état pointé par cet arc.
- S'il n'existe pas un arc correspondant à la valeur de ch et si l'état actuel n'appartient pas à l'ensemble des états qui indiquent la rencontre d'une unité lexicale (i.e. ce n'est pas un état d'acceptation), alors il y a un échec de l'analyse et la recherche est initialisée à celle de l'unité lexicale spécifiée par le prochain diagramme de transition.
- On utilise généralement une instruction de choix multiple pour trouver l'état de départ du diagramme de transition suivant. On suit les arcs du diagramme de transition en sélectionnant le fragment de code d'un état et on exécute ce fragment pour déterminer le prochain état.

Remarque

On suppose que les diagrammes de transition utilisés sont déterministes. En d'autres termes, aucun caractère ne peut apparaître comme étiquette de deux ou plusieurs arcs quittant un même état.

b. Construction d'analyseur lexical basée sur les automates finis

Les étapes suivantes définies une approche rigoureuse pour la construction d'un analyseur lexical en utilisant les automates d'états finis.

- **Etape 1.** Spécification des unités lexicales : Spécifier chaque type d'unité lexicale à l'aide d'une expression régulière (ER).
- **Etape 2.** Conversion ER en AFN : Convertir chaque expression régulière en un automate d'états fini (non déterministe).
- **Etape 3.** Réunion des AFNs : Construire l'automate Union de tous les automates de l'étape 2 (on peut ajouter un nouvel état initial d'où partent un ensemble d'arcs étiquetés ε).

- **Etape 4.** Déterminisation ou transformation de l'AFN obtenu en AFD : Rendre l'automate de l'étape 3 déterministe.
- **Etape 5.** Minimisation de l'AFD : Minimiser l'automate obtenu à l'étape 4.
- **Etape 6.** Implémentation de l'AFD minimisé : Implémenter l'automate obtenu à l'étape 5 en le simulant à partir de sa table de transition.

On transforme une expression régulière en un reconnaisseur en construisant un diagramme de transition généralisé appelé automate à états finis. Ce dernier peut être déterministe (AFD) ou non déterministe (AFN). La transformation d'un automate à états finis en un analyseur lexical consiste à associer une unité lexicale à chaque état final et à faire en sorte que l'acceptation d'une chaîne produise comme résultat l'unité lexicale associée à l'état final en question. Pour cela il faut implémenter la fonction de transition de l'automate en utilisant une table de transition.

Automate fini non déterministe

Un AFN est défini par :

- Un ensemble d'état E
- Un ensemble de symboles d'entrée ou alphabet Σ
- Un état initial E_0
- Un ensemble F d'états finaux ou d'acceptation
- Une fonction de transition *« Transiter »* qui fait correspondre à chaque couple (état, symbole), un ensemble d'états

L'implémentation la plus simple d'un AFN est une table de transition dans laquelle il y a une ligne pour chaque état et une colonne pour chaque symbole d'entrée (avec la chaîne vide ε si nécessaire). L'entrée pour la ligne i et le symbole a donne l'ensemble des états qui peuvent être atteints à partir de l'état i avec le symbole a. Un AFN accepte une chaîne si et seulement s'il existe au moins un chemin correspondant à cette chaîne entre l'état initial et l'un des états finaux.

Automate fini déterministe

Un AFD est un cas particulier d'AFN dans lequel :

- Aucun état n'a de ε-transition
- Pour chaque état E et chaque symbole d'entrée a il y a au plus un arc étiqueté a qui quitte E

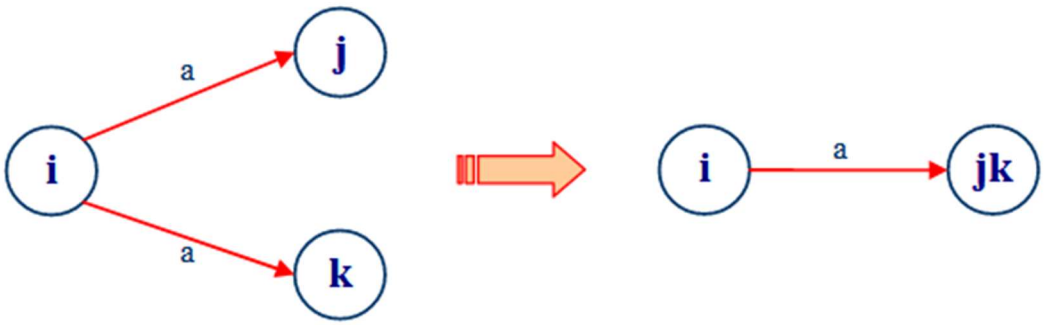
Dans la table de transition d'un AFD, une entrée contient un état unique au maximum (les symboles d'entrée sont les caractères du texte source), il est donc très facile de déterminer si une chaîne est acceptée par l'automate vu qu'il n'existe, au plus, qu'un seul chemin entre l'état initial et un état final étiqueté par la chaîne en question.

Algorithme pour rendre déterministe un AFN :

- Si on peut passer d'un état i à l'état j avec une transition étiquetée ε alors les états i et j appartiennent à la même classe



- Si l'état i mène à l'état j et à l'état k par des transitions identiques (même étiquette), alors les états j et k appartiennent à la même classe



- Une classe d'état est finale dans l'AFD si elle contient au moins un état final de l'AFN.

Pour minimiser un automate, il faut tout d'abord :

- S'assurer qu'il est déterministe.
- Éliminer les états qui ne sont pas atteignables à partir de l'état initial, ainsi que les transitions adjacentes à ces états.
- Éliminer les états à partir desquels on ne peut pas atteindre un état final, ainsi que les transitions adjacentes à ces états

Ensuite l'algorithme et défini comme suit :

- Définir deux classes, C_1 contenant les états finaux et C_2 contenant les états non finaux.
- S'il existe un symbole *a* et deux états e_1 et e_2 d'une même classe tels que Transiter(e_1, *a*) et Transiter(e_2, *a*) n'appartiennent pas à la même classe (d'arrivée), alors créer une nouvelle classe et séparer e_1 et e_2. On laisse dans la même classe tous les états qui donnent un état d'arrivée dans la même classe.
- Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de classes à séparer.
- Chaque classe restante forme un état du nouvel automate

c. Un exemple de générateur d'analyseurs lexicaux : Lex

Les étapes de construction d'un analyseur lexical à partir de la spécification des unités lexicales peuvent être automatisées en utilisant un outil logiciel dit générateur d'analyseur lexical tel que l'outil LEX qui peut être considéré comme le plus ancien. Cette approche de construction des analyseurs lexicaux est considérée comme la plus simple car on se limite à fournir les définitions exactes des unités lexicales du langage considéré et le générateur fait le reste en générant le code source de l'analyseur lexical. Le générateur LEX permet de générer un analyseur lexical en langage C à partir des spécifications des unités lexicales exprimées en langage LEX. Le noyau de LEX permet de produire l'automate d'état fini correspondant à la reconnaissance de l'ensemble des unités lexicales spécifiées. Ainsi, LEX construit une table de transition pour un automate à états finis à partir des modèles d'expressions régulières de la spécification en langage LEX.

Le fichier de règles pour LEX contient quatre parties successives.

3.1. Déclarations (partie facultative) : Cette partie commence par ***%{*** et finit par ***%}*** chacun de ces délimiteurs devant être écrit sur une ligne ne contenant rien d'autre. Toutes les lignes de code C écrites entre ces deux délimiteurs seront directement recopié par Lex au début du fichier source lex.yy.c. Elle sert essentiellement à insérer des directives de pré-compilation C (#ifdef, ...), à inclure des bibliothèques et des sources extérieurs (#include, ...), et à déclarer des variables et constantes globales. On peut aussi y définir les fonctions et procédures qui

serviront dans le code associé aux expressions (troisième partie) et/ou dans le code utilisateur (quatrième partie).

3.2. Définitions (partie facultative) : Cette partie contient les définitions opératoires (lignes commençant par %) et les définitions d'expressions. Les définitions opératoires contiennent notamment les déclarations de configurations.

3.3. Règles (=productions) : Cette partie, qui est la seule **obligatoire**, commence par le délimitateur ***%%*** écrit seul dans sa ligne. Elle contient une suite (éventuellement vide) de règles (une ligne par règle) qui sont des expressions rationnelles augmentées : l'expression rationnelle est suivie, après un (ou plusieurs) espace(s) et/ou tabulation(s), du code C qui lui est associé et qui sera exécuté chaque fois qu'un motif correspondant à cette expression est reconnu. Elle décrit les actions à effectuer lors de la détection des différents éléments lexicaux décrits dans la partie précédente. Chacune des entrées contient deux blocs entre accolades :

- Le premier bloc entre accolades contient le nom de l'élément lexical considéré ;
- Le second bloc entre accolades contient le code C de l'action correspondante à exécuter.

3.4. Code utilisateur (partie facultative) : Cette partie commence par le délimiteur ***%%*** seul dans sa ligne. Toutes les lignes de code C écrites après seront directement recopiées par Lex dans le fichier lex.yy.c. Cette partie permet notamment de redéfinir les fonctions yywrap() et main().

Exemple

Voici le texte source pour créer l'analyseur lexical d'un langage comportant les nombres et les identificateurs définis comme d'habitude, les mots réservés *si, alors, sinon, tantque, faire* et *rendre* et les opérateurs doubles ==, *!=, <= et* >=. Les unités lexicales correspondantes sont respectivement représentées par les constantes conventionnelles *IDENTIF, NOMBRE, SI, ALORS, SINON, TANTQUE, FAIRE, RENDRE, EGAL, DIFF, INFEG, SUPEG*, définies dans un fichier *unitesLexicales.h*.

```
%{
        #include "unitesLexicales.h"
%}
%chiffre [0-9]
%lettre [A-Za-z]
%%
[" "\t\n] { ECHO; /* rien */ }
{chiffre}+ { ECHO; valNombre = atoi(yytext); return NOMBRE; };
si { ECHO; return SI; }
alors { ECHO; return ALORS; }
sinon { ECHO; return SINON; }
tantque { ECHO; return TANTQUE; }
fairerendre { ECHO; return FAIRE; }
{lettre}({lettre}|{chiffre})* {
ECHO; strcpy(valIdentif, yytext); return IDENTIF; }
"==" { ECHO; return EGAL; }
"!=" { ECHO; return DIFF; }
"<=" { ECHO; return INFEG; }
">=" { ECHO; return SUPEG; }
. { ECHO; return yytext[0]; }
%%
int valNombre;
char valIdentif[256];
int yywrap(void) {
return 1;
}
```