



THÉORIE & TECHNIQUES DE COMILATION / INTERPRÉTATION

Samir El Khattabi
Ecole Centrale de Lille
samir.elkhattabi@centralelille.fr



PARTIE 1

PRÉSENTATION GÉNÉRALE

Introduction
Modèle de compilation
Structure & Phases d'un compilateur
Cousins des compilateurs

Introduction à la Compilation

La compilation a pour objectif principal la traduction d'un programme source (écrit dans un langage source) en un programme cible (écrit dans un autre langage).

Un compilateur doit également détecter les erreurs d'écriture dans le programme source.

Les principes et techniques de développement de compilateurs sont si fondamentaux qu'ils sont réutilisables dans d'autres domaines :

- les langages de programmation,
- l'architecture des machines,
- l'algorithme & génie logiciel.

Peu de techniques sont nécessaires à la mise en place de traducteurs pour une grande variété de langages et de machines.

... Domaines d'application

1

Ecriture d'outils de développement ou de transport de logiciel :

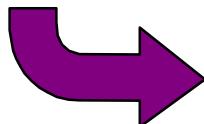
- **Compilateur produisant du code machine,**
- **Traducteur d'un langage de haut-niveau en un autre,**
- **Interpréteur.**

2

Ecriture d'outils d'échanges entre applications hétérogènes

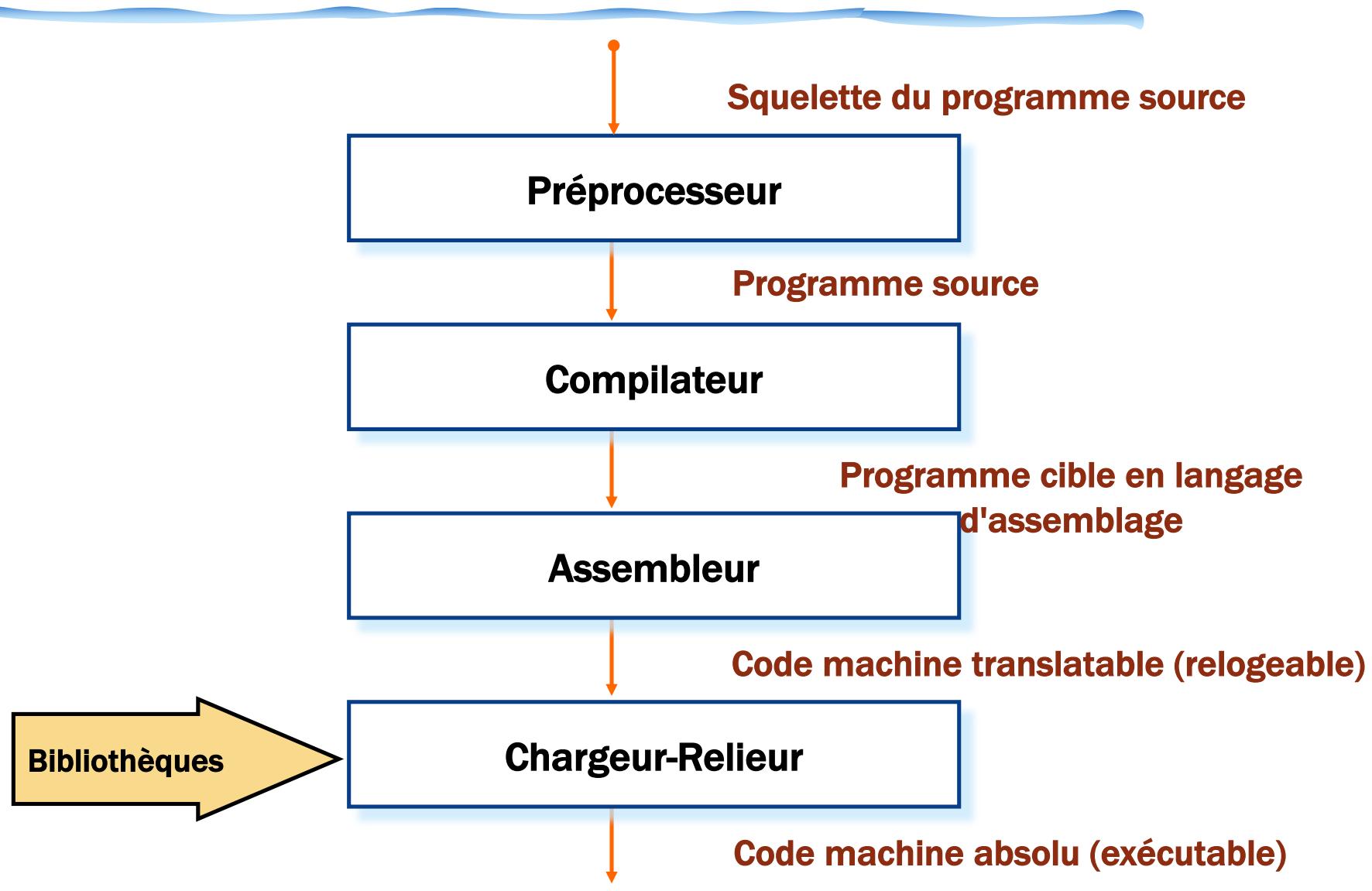
Exploitation de tous types de structures de données :

- **parseur,**
- **Macro-processeur,**
- **Processeur et extension du langage,**
- **Interpréteur du questionnement (Query Interpreter : SQL).**



ADOPTER UNE DEMARCHE METHODOLOGIQUE

Modèle de compilation



Préprocesseur (1)

Un préprocesseur **produit** le programme source qui sera en entrée d'un compilateur après avoir effectué des **traitements précédant la compilation**.

Plusieurs fonctions possibles :

- Inclusion de fichiers :
 - ▶ #include du C ou %include du Pascal,
- Macro-expansion ou macro-définition :
 - ▶ abréviation pour des constructions longues,
- Extension du langage par macros intrinsèques :
 - ▶ le langage Equel est un langage de requêtes de base de données intégré au langage C,

Préprocesseur (2)

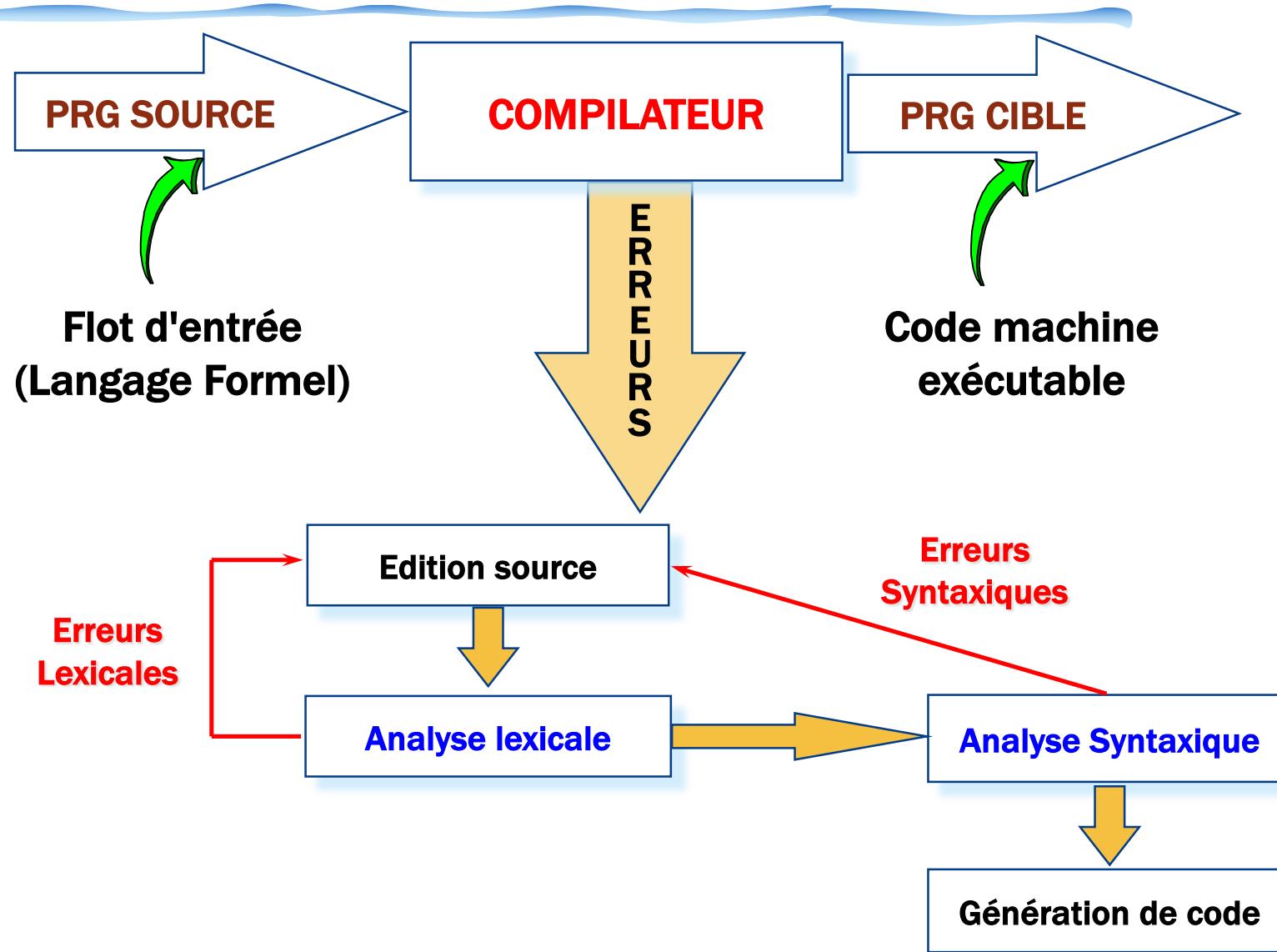
Un macros-processeur traite deux sortes d'instructions :

- Les définitions : balise de définition, nom de la macro, les arguments éventuels et son corps,

```
#define CHECK(sts, msg) if ((sts)==-1) {perror(msg); exit(sts);}
```

- Les appels : nom de la macro et ses arguments effectifs qui vont prendre place dans le corps à la place des arguments formels. Le code obtenu par substitution remplace l'appel de la macro.

Compilateur



Assembleur

Un assembleur a comme point d'entrée un programme en langage d'assemblage ou **code à trois adresses** et produit du **code machine** translatable.

Le langage d'assemblage est une version **mnémonique** du code machine :

- on utilise des noms pour désigner les opérations et non pas les codes binaires.

Assemblage en deux étapes

La 1^o étape, **repère** les identificateurs qui font référence à des adresses mémoire et les **stocke** dans une table des symboles (différente de celle d'un compilateur) :

- Cette étape implique une **allocation** d'emplacements mémoire pour les nouveaux identificateurs,
- La connaissance de la **taille du type** intervient à ce moment.

La 2^o étape, traduit toute opération en son code binaire (suite de bits représentant l'opération dans le langage machine).

Exemple : b := a + 2

MOV a, R1	0001 01 00 00000000	* Charger dans R1
ADD #2, R1	0011 01 10 00000010	* Ajouter à R1
MOV R1, b	0010 01 00 00000100	* Ranger depuis R1

Chargeur - Relieur

L'**édition de liens** se fait en deux étapes, le **chargement** et **reliure** d'où l'appellation **chargeur-relier** :

- Fonction de **chargement** qui consiste à **translate** les adresses **translatables** dans le code translatable par rapport à une adresse A de référence.
- Fonction de **reliure** qui consiste à **constituer un programme unique** à partir de plusieurs fichiers générés par compilations séparées, et/ou plusieurs d'entre eux sont des fichiers ou routines de bibliothèques fournis par le système d'exploitation :

`/usr/lib/libld.a`

La séparation de la fonction de reliure d'un compilateur est le choix d'un **socle commun (code intermédiaire)** à plusieurs langages permet de réutiliser les programmes (**bibliothèques d'objets**), et d'utiliser le langage de programmation le plus adapté (**compilation séparée**).

... La compilation par analyse et synthèse

Un compilateur procède selon deux étapes :

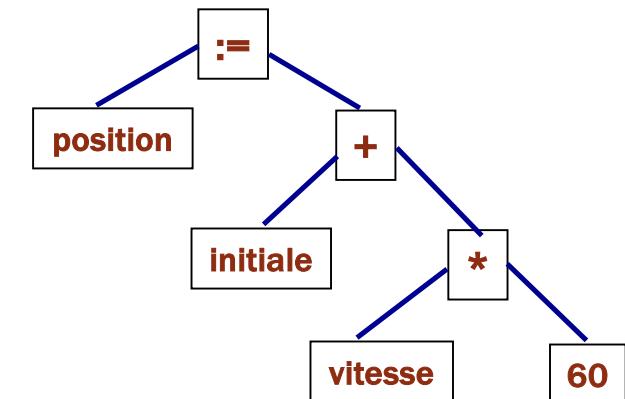
- analyser : partitionner le programme source en éléments constituants et en créer une **représentation intermédiaire**,
- synthétiser : construire le programme cible depuis cette représentation intermédiaire.

Structure de données arborescente pour la représentation intermédiaire :

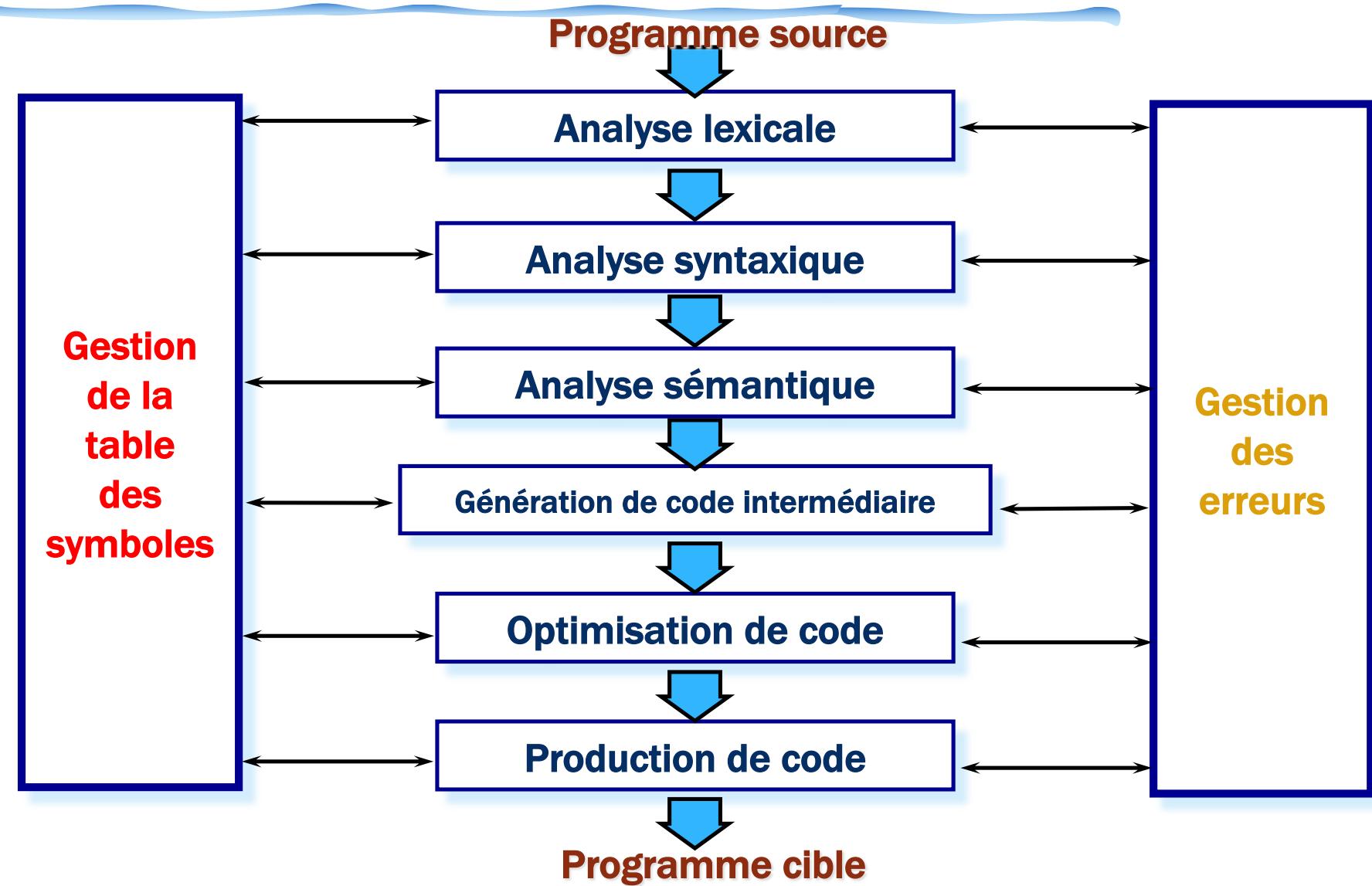
- L'analyse permet de déterminer les opérations spécifiées dans le langage source et de les conserver sous une forme hiérarchique.
- La représentation intermédiaire utilise donc les arbres (abstraits).
- Un nœud représente une opération, et les fils de ce nœud les arguments de cette opération.

Exemple :

position := initiale + vitesse * 60



Structure/Phases d'un compilateur



Compilateur & dépendance matérielle

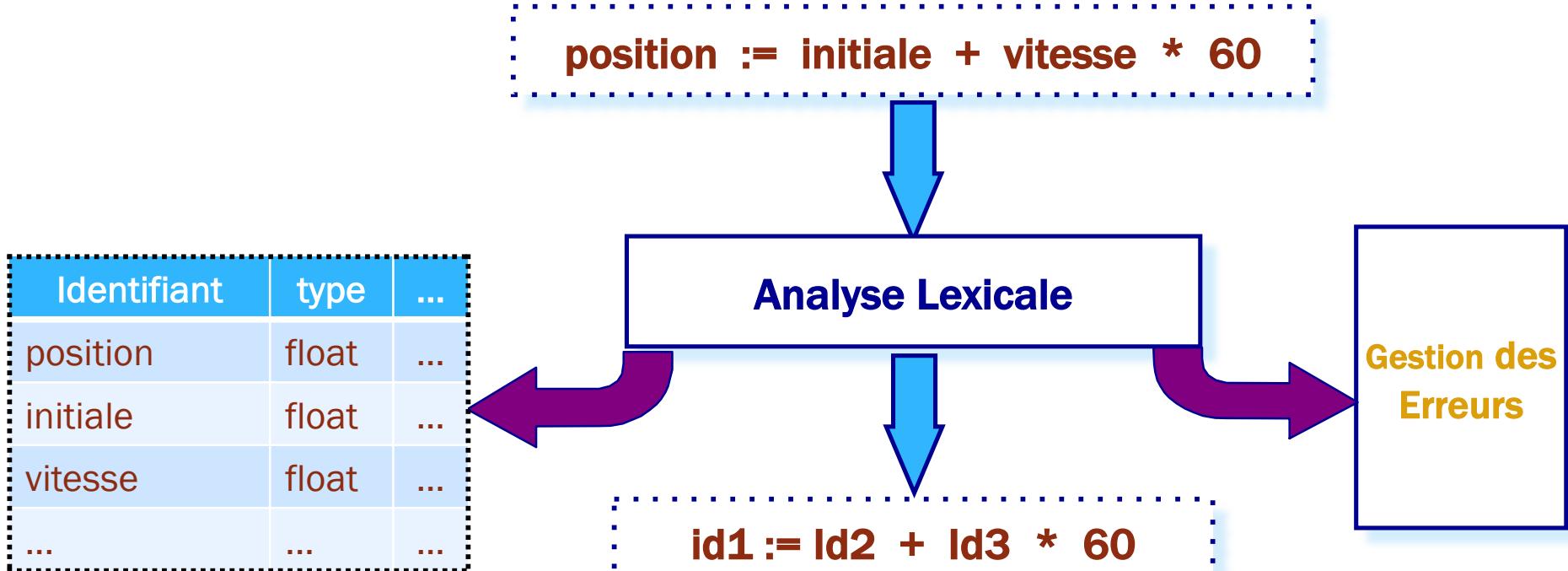
Un compilateur est constitué de deux parties :

- La partie **frontale** constituée des phases de 1 à 4 ne dépend pas de la machine mais uniquement du langage intermédiaire :
 - ▶ Analyses lexicale, syntaxique & sémantique.
 - ▶ Génération de code intermédiaire.
- La partie **finale** constituée des phases 5&6 dépend du langage cible et donc de la machine d'exécution :
 - ▶ Optimisation & Production de code.

La partie frontale d'un compilateur est **inchangée** quand on change de machine.

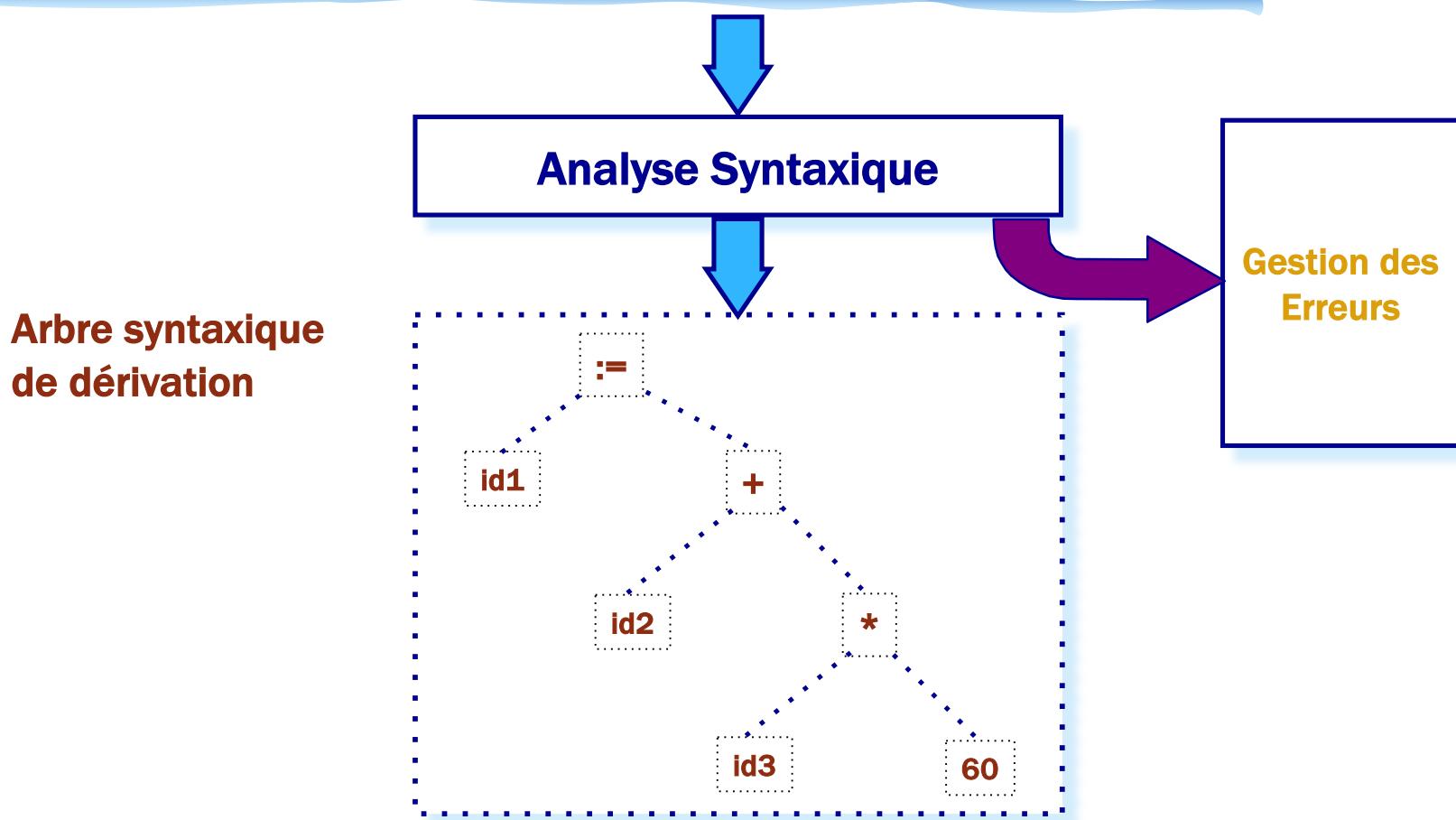
La partie finale peut être **commune** à plusieurs langages.

Phase 1 : Analyse Lexicale



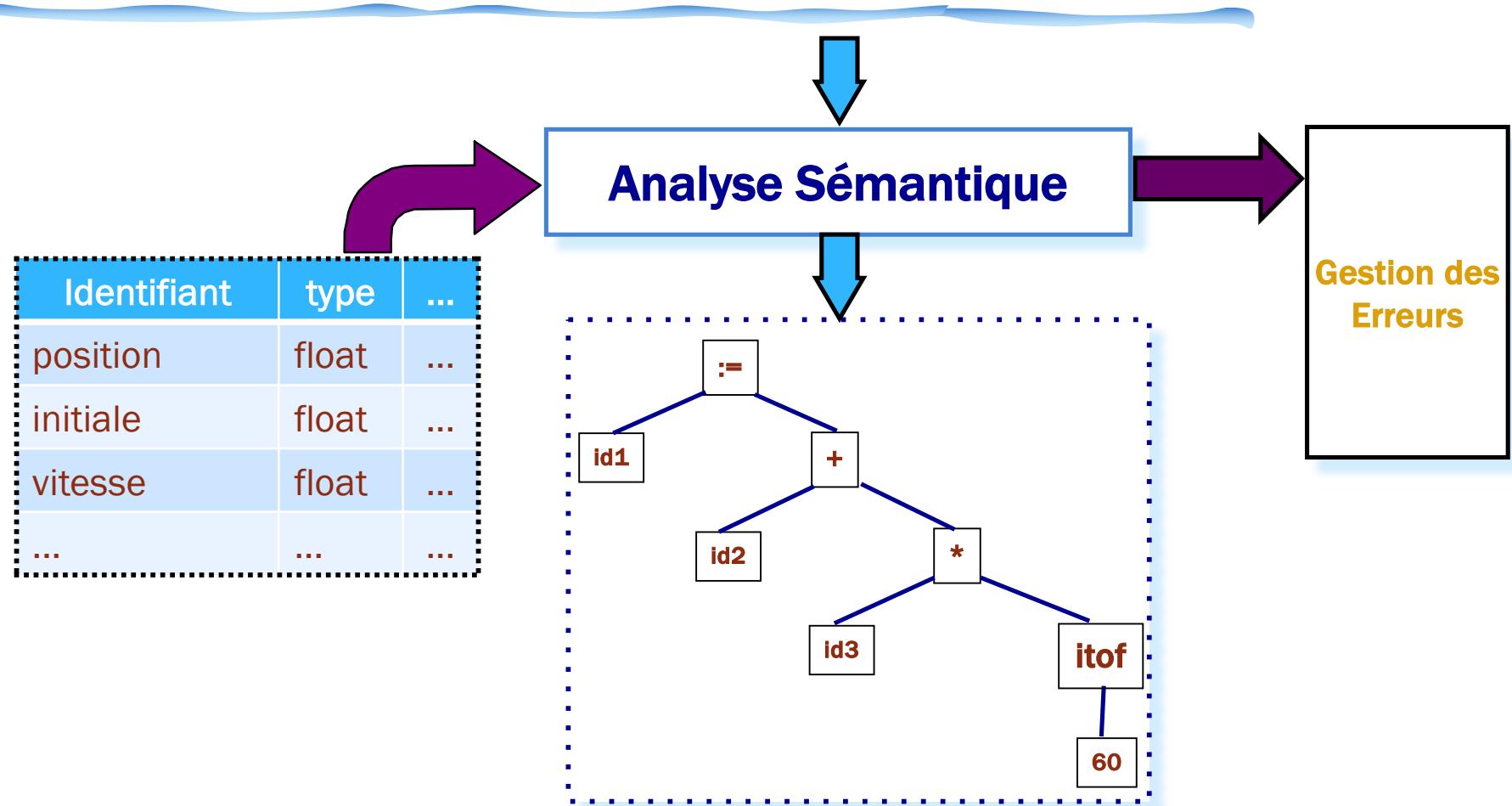
L'analyse lexicale consiste à lire un programme source et à le découper en entités lexicales appelées **unités lexicales (lexèmes)** en appliquant des **règles lexicales**.

Phase 2 : Analyse Syntaxique



L'analyse syntaxique (hiérarchique ou grammaticale) consiste à regrouper les lexèmes en **unités (structures) syntaxiques** en appliquant des **règles syntaxiques**.

Phase 3 : Analyse Sémantique



L'analyse sémantique **contrôle** la cohérence du type, du nombre et de la visibilité des opérandes.

Phase 4 : Génération de Code Intermédiaire

La génération de code intermédiaire consiste à produire un code en langage “machine abstraite” (pseudo-assembleur ou **code à trois adresses**) avec allocation des variables temporaires et utilisation de la notation **post-fixée** (**machine à pile**).

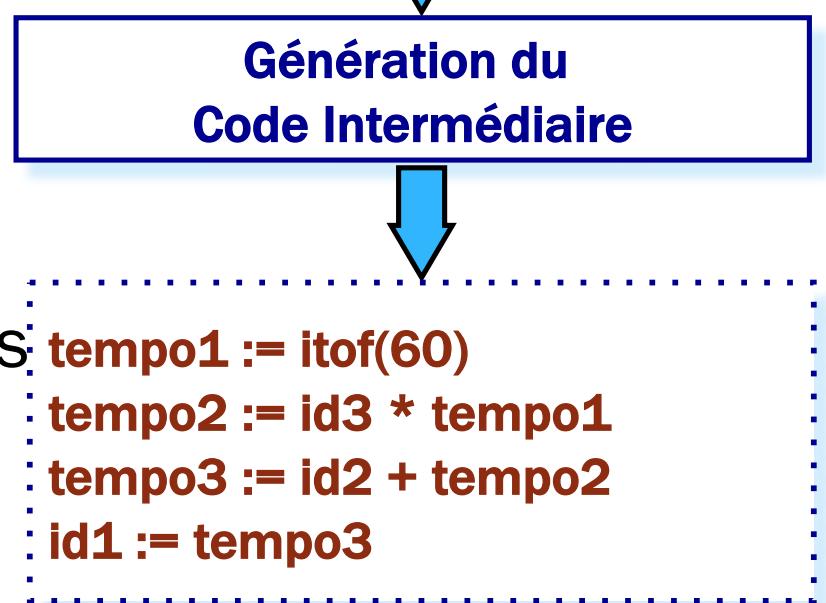
Il ne devrait plus y avoir d'erreurs.

Le code intermédiaire est :

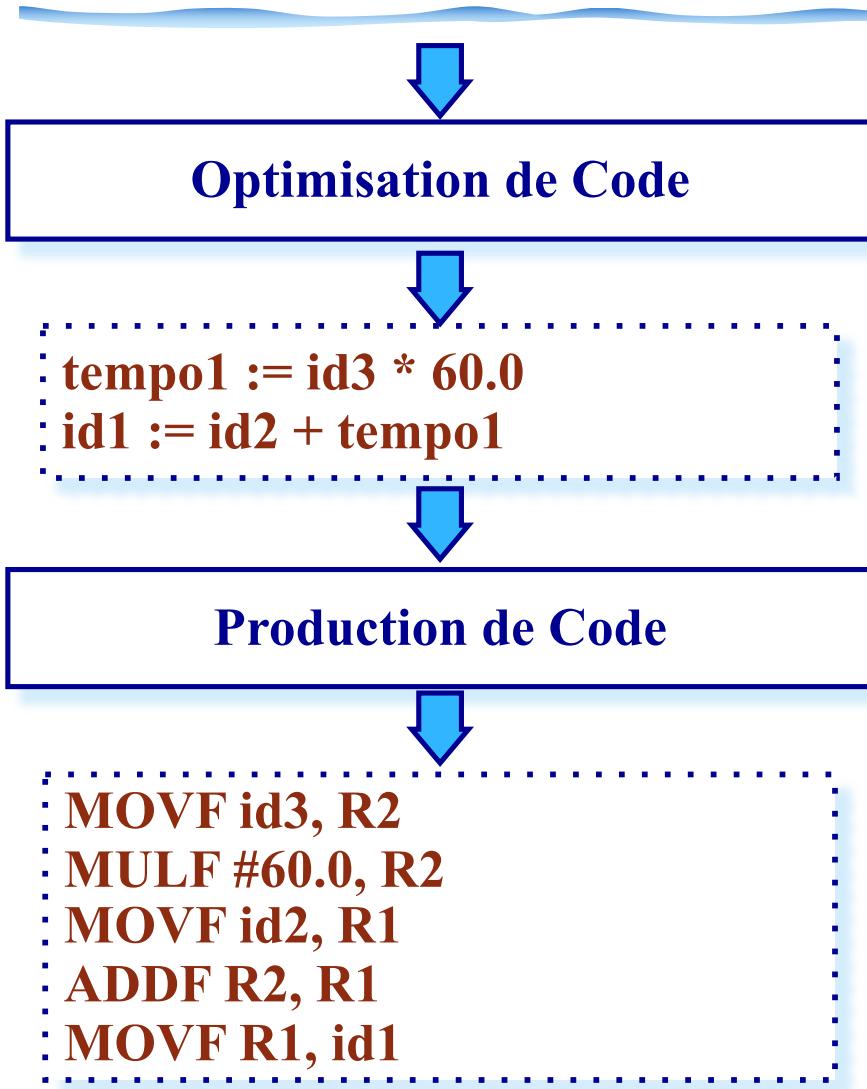
- facile à produire,
- facile à traduire.

Propriétés du code à trois adresses:

- 1 Opération (affectation),
- Var temporaire (valeur),
- Moins de 3 opérandes.



Phases 5&6 : Optimisation et Production de Code



Optimisation :

- Réduction du nombre de variables et des opérations statiques.
- Exploitation des caractéristiques de la machine cible (microprocesseur).

Traduction en langage machine translatable ou non

:

- Dépendance de la machine.
- Assignment des variables aux registres.

Table des symboles (1)

La **table de symboles** sert à **mémoriser** les **identificateurs** utilisés par le programme source ainsi qu'une collection d'informations sur celui-ci (**attributs**) :

- Nom du symbole,
- Type de données du symbole,
- Emplacement mémoire du symbole et sa taille,
- Portée du symbole : région du code où l'identificateur est valide,
- Pour les symboles procédure/fonction :
 - ▶ Nombre d'arguments,
 - ▶ Type de chaque argument
 - ▶ Mode de passage de chaque argument,
 - ▶ et éventuellement le type retourné pour une fonction.

Table des symboles (2)

Chaque symbole sera représenté par un **n-uplet** (**struct** en langage C) contenant les champs informationnels relatifs aux attributs de celui-ci.

Les symboles seront **insérés** dans une **table dynamique** (ou une **liste chaînée**).

La gestion de cette table est construite autour de deux traitements principaux : **insertion & recherche**. La recherche (**adresse du symbole**) permet de rendre un traitement de mise à jour simple.

La table des symboles est utilisée par les différentes phases de compilation :

- L'**analyse lexicale** **crée** les identificateurs non-définis.
- L'**analyse sémantique** **valide** le type d'un identificateur sachant que l'**analyse syntaxique** **a renseigné** ce champ.

Gestion des erreurs

Toute phase de compilation peut **générer** des erreurs. En cas d'erreur, la compilation **doit se poursuivre** et continuer à **déetecter** les autres erreurs.

La majeure partie des erreurs est traitée lors des phases **d'analyse syntaxique et sémantique**.

Analyse syntaxique → Erreurs structurelles.

Analyse sémantique → Erreurs de sens,
Compatibilité des opérandes.

L'**analyse lexicale** peut également générer des erreurs lorsqu'elle rencontre des caractères ne formant pas une entité lexicale.

Interpréteur – cousin du compilateur

Un interpréteur réalise lui-même l'opération reconnue, au lieu de générer le code cible pour son exécution ultérieure.

Il se substituent à la **machine d'exécution** en simulant le langage source : **machine virtuelle**.

La **partie finale** d'un compilateur est remplacée par un **module d'interprétation**.

Domaines :

- langages informatiques : Basic, Forth, Lisp, ...
- langage de requêtes d'accès à une BdD,
- langage de commandes : Shells UNIX, DCL de VMS, COMMAND de DOS,

Propriétés :

- Mise au point des programmes sources : messages d'erreurs précis, accès interactifs aux variables, modifications à effet immédiat,
- Exécution plus lente des programmes.



PARTIE 2

BASES THÉORIQUES

Généralités sur les langages
Grammaire formelle
Dérivation : arbres syntaxiques

Langage formel – grammaire

Un **langage** se définit en -- grande -- partie par sa **grammaire** (ou syntaxe).

Une **grammaire** **décrit** la **structure hiérarchique** des constructions du langage (souvent de programmation).

Exemple : **if (expression) instruction else instruction**

Unités lexicales : "if", : "else", : "(", : ")"

Unités syntaxiques : **expression, instruction**

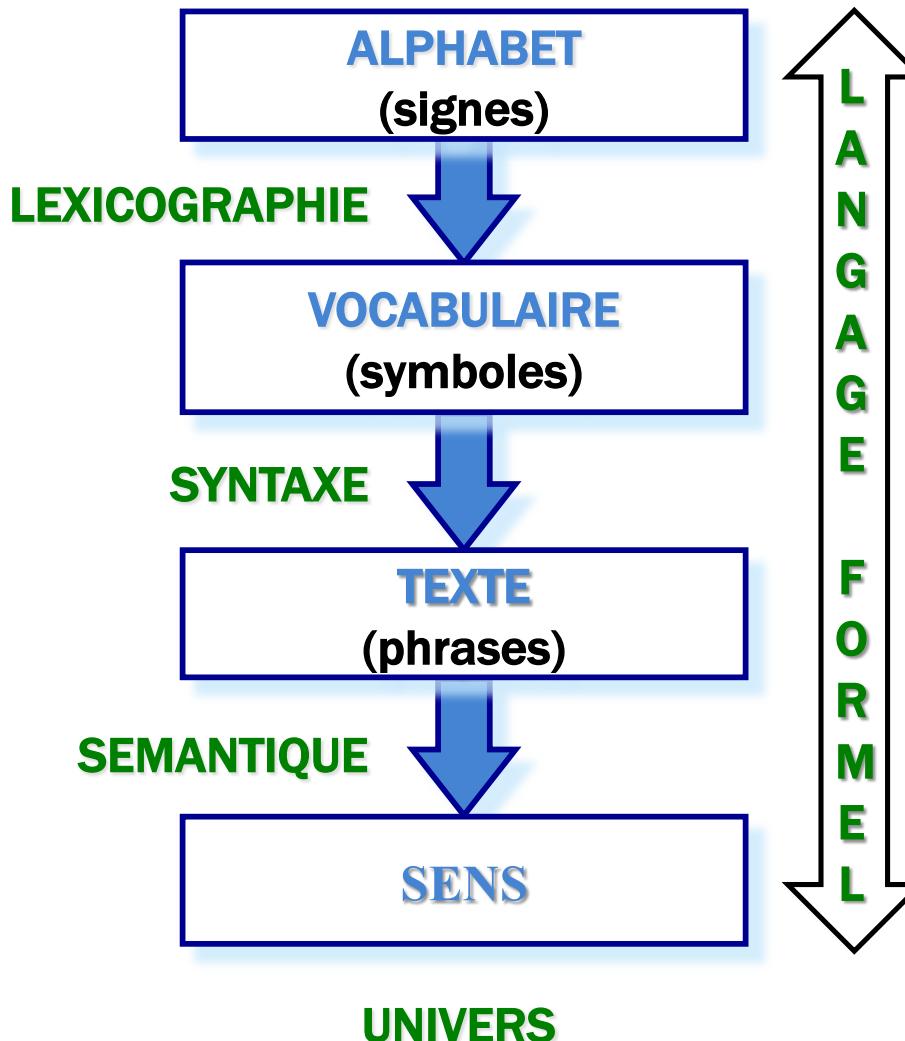
Règle syntaxique :

instr → if (expr) instr else instr

Cet exemple utilise une **règle de production** indiquant une des formes possibles d'une instruction. Une instruction peut avoir la forme ...

Cette **règle** est une séquence formée d'**unités lexicales (symboles terminaux)** et de **constructions syntaxiques (symboles non terminaux)**.

Langage formel – définition / terminologie



Définition des grammaires génératives

Un **langage $L(G)$** généré par une **grammaire G** est un ensemble de toutes les **phrases possibles**.

Dit autrement : toutes les **formes syntaxiques** constituées uniquement des **unités lexicales** (symboles terminaux).

CHOMSKY (1957)

Grammaire formelle – terminologie

Le **vocabulaire** utilisé par un langage est défini par un ensemble **non vide** de **symboles terminaux**, noté V_T .

Une chaîne (**symbole non-terminal**) est une suite finie de symboles de V_T , éventuellement **vide** (λ est la chaîne vide). L'ensemble V_N est formé des chaînes reconnues.

V^* est l'ensemble de **toutes** les chaînes formées de symboles de V_T (y compris les symboles de V_T et λ).

Grammaire formelle – définition

Une grammaire formelle \mathbf{G} est une description des symboles du langage et de sa syntaxe (**règles syntaxiques**) d'un langage notée $L(\mathbf{G})$.

Une **grammaire** se définit par $\mathbf{G} = (V_T, V_N, S, P)$ avec :

- | | |
|---|----------------------------|
| V_T : ensemble des symboles terminaux | $V^* = V_T \cup V_N$ |
| V_N : ensemble des symboles non terminaux | $\emptyset = V_T \cap V_N$ |
| S : symbole initial (axiome) | $S \in V_N$ |
| P : ensemble de règles de la forme $\alpha \rightarrow \beta$ avec $\alpha, \beta \in V^*$ | |

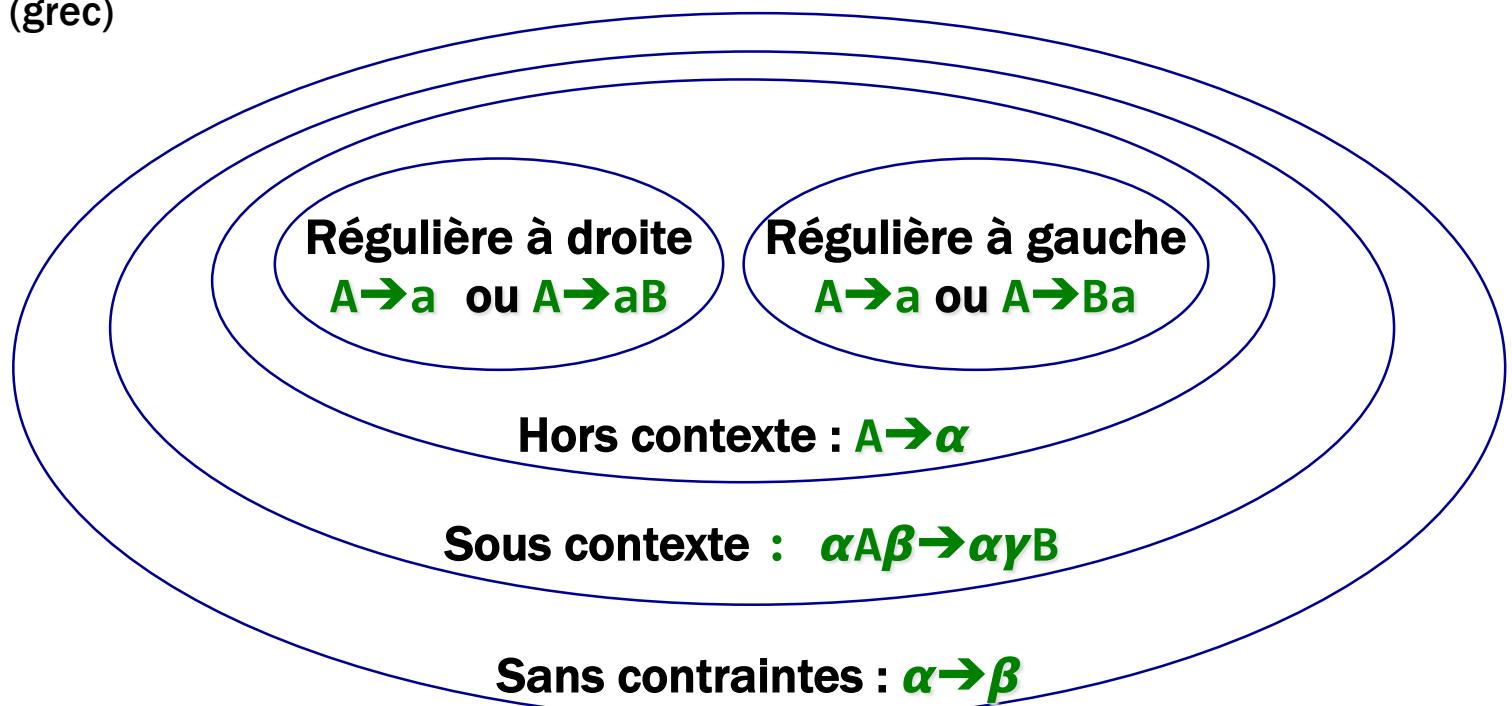
Une **forme syntaxique reconnue** est une chaîne (α) dérivable depuis S ($S \xrightarrow{*} \alpha$)

Classification des Langages (CHOMSKY)

$$L(G) = \{ \alpha / \alpha \in V_T^* \text{ et } S \xrightarrow{*} \alpha \}$$

CONVENTIONS :

- a...z symboles terminaux (minus)
- A...Z symboles non terminaux (majus)
- $\alpha...w$ chaînes (grec)



Grammaire d'un langage - spécifications

Une grammaire non contextuelle se définit par :

1. un ensemble d'**unités lexicales** appelées **symboles terminaux**.
2. Un ensemble de **symboles non terminaux**.
3. Un ensemble de **règles de production** où chaque production possède deux parties : une partie gauche constituée d'un non-terminal, et d'une partie droite de la production.
4. Un **axiome** désigne un symbole non terminal comme symbole de dérivation.

Expression algébrique utilisant un signe entre deux chiffres (exemples : 6-2+5, 5-2, 7) :

- Règles de production :

expr	→ expr + chiffre
expr	→ expr - chiffre
expr	→ chiffre
chiffre	→ 0 1 9

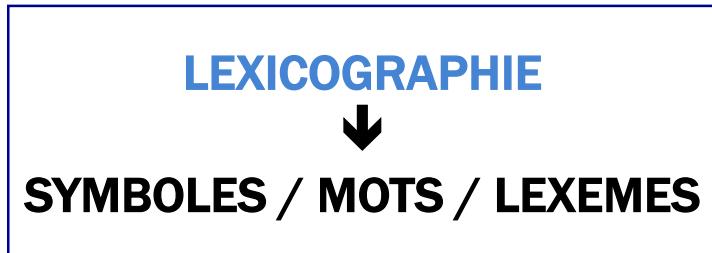
- Ecriture équivalente pour expr :

expr	→ expr + chiffre expr - chiffre chiffre
chiffre	→ 0 1 9

Notation : BACKUS NAUR FORM (BNF)

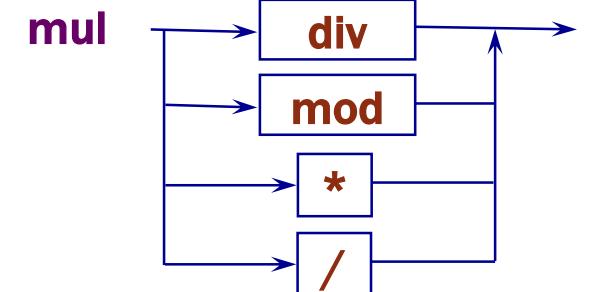
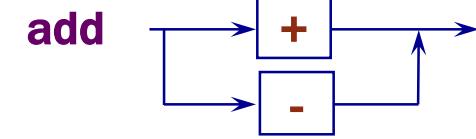
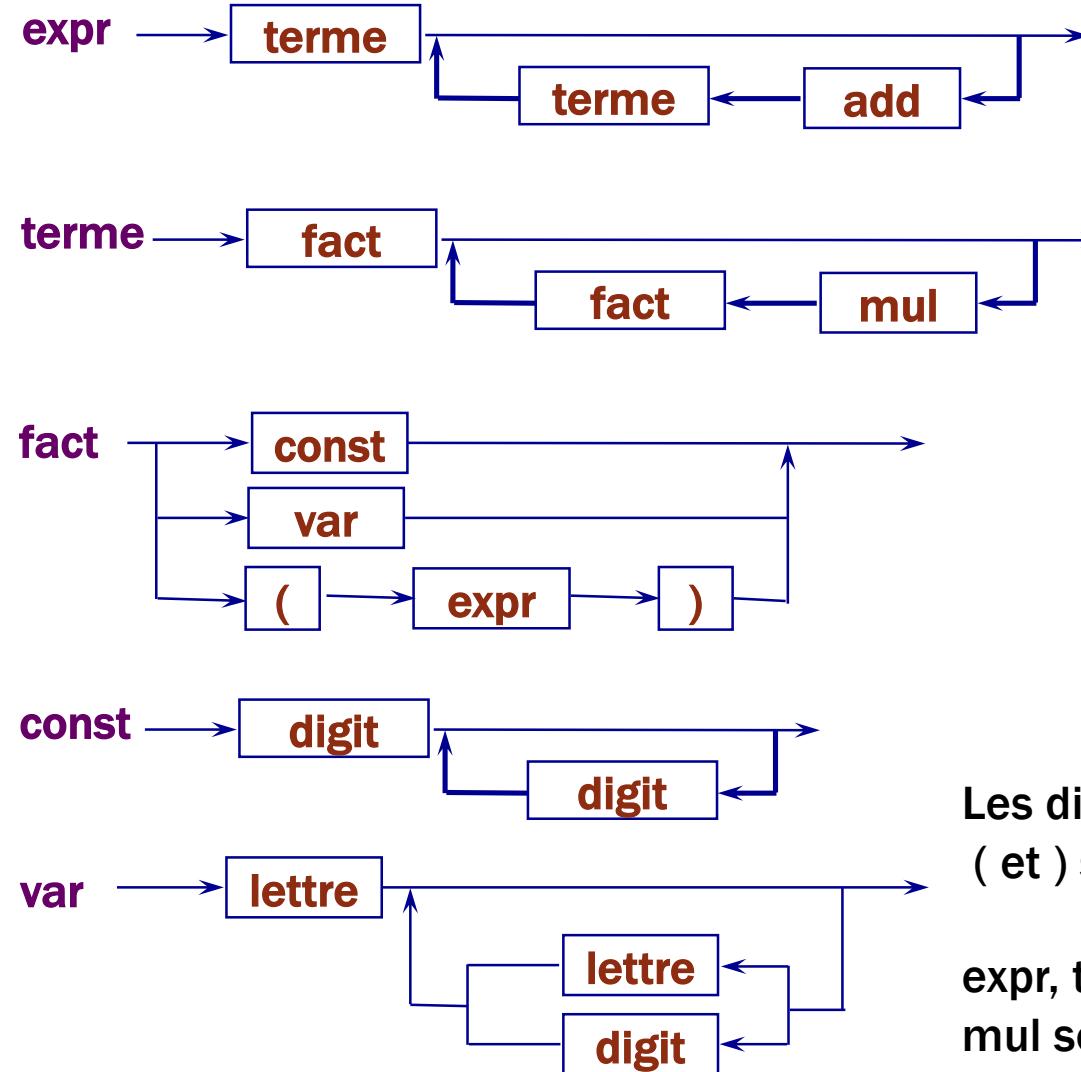


```
<expr> ::= <terme> { <add> <terme> }
<terme> ::= <fact> { <mult> <fact> }
<fact>  ::= <const> | <var> | ( <expr> )
```



```
<add> ::= + | -
<mult> ::= * | / | mod | div
<var>  ::= <lettre> { <lettre> | <digit> }
<const> ::= <digit>{<digit>}
<lettre> ::= a | b | ... | z
<digit> ::= 0 | 1 | ... | 9
```

Notation : Diagramme syntaxique $3 + 5 * 6$



Les digits, les lettres, **+**, **-**, *****, **/**, mod, div, (et) sont tous des **symboles terminaux**.

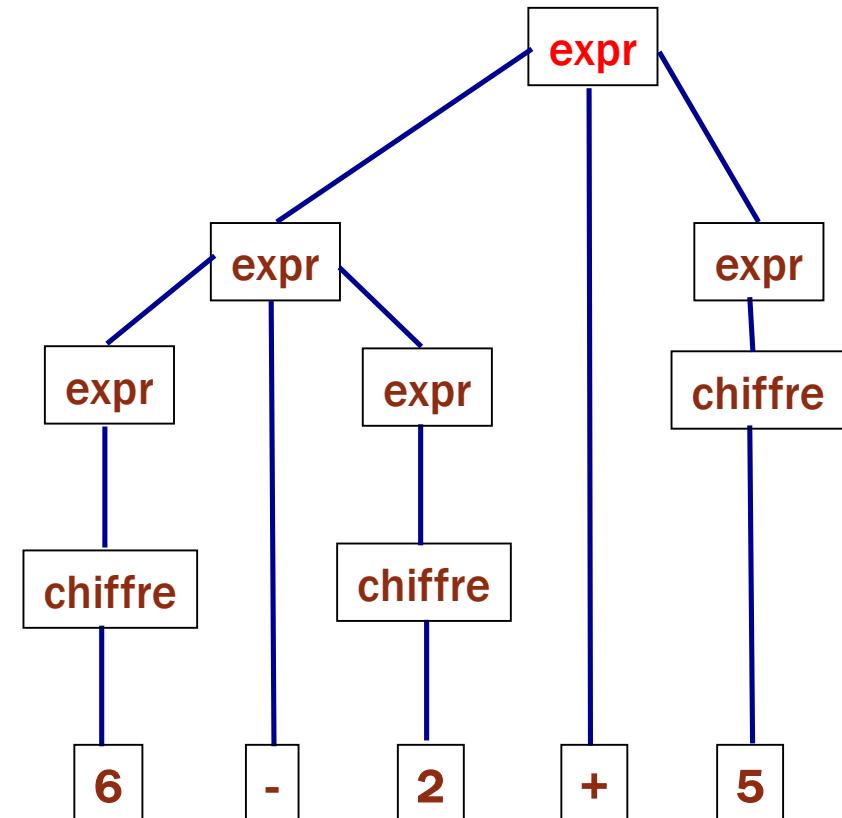
expr, **terme**, **fact**, **const**, **var**, **add** et **mul** sont des **symboles non terminaux**.

Dérivation

Une grammaire **dérive** des chaînes en commençant par l'**axiome** et **en remplaçant** (et ce de manière répétée) un non-terminal par la partie droite d'une des productions le définissant.

Les chaînes d'unités lexicales dérivables à partir de l'axiome forment le langage engendré par la grammaire.

Exemple : 6 - 2 + 5

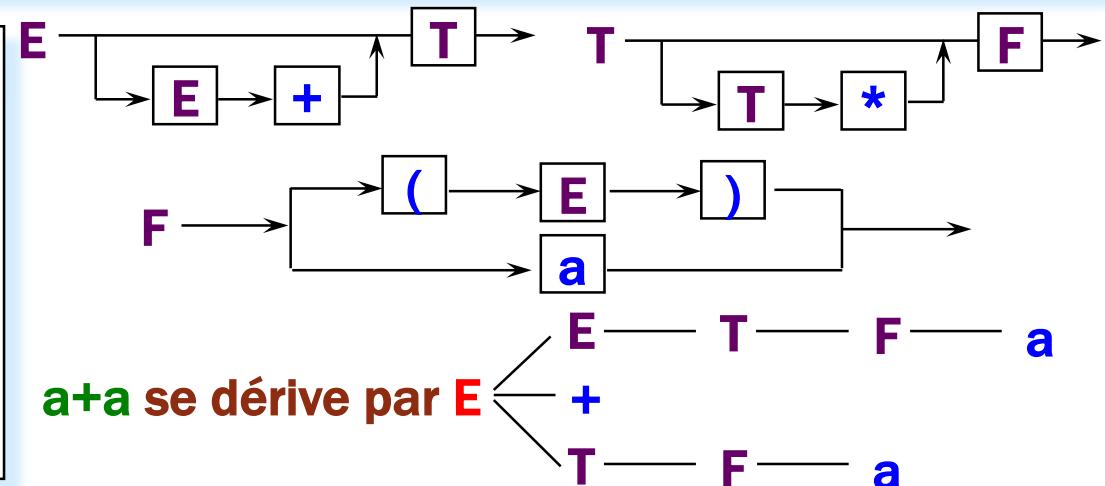


Arbre de dérivation

Soit D , un **arbre de dérivation** de la grammaire $G = (V_T, V_N, X_i, P)$ alors :

1. La **racine** de D est l'axiome S
2. Les **nœuds** de $D \in V_T \cup V_N \cup \{ \lambda \}$
3. Si les sous arbres D_i de racines X_i sont des descendants directs de S alors :
 - $(S \rightarrow X_1 \dots X_k) \in P$
 - $X_i \in V_N \Rightarrow D_i$ est un **arbre de dérivation** pour G
 - $X_i \in V_T \Rightarrow D_i$ est le nœud X_i
4. Si D_1 est le **seul** sous arbre de D et la racine de D_1 est $\lambda : (S \rightarrow \lambda) \in P$

$G = (\{a, +, *, (,)\}, \{E, T, F\}, E, P)$
 $P:$
 $E \rightarrow E+T$
 $E \rightarrow T$
 $T \rightarrow T*F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow a$



Non unicité de la description d'une grammaire

Non unicité de la grammaire pour un langage :

<entier> ::= <digit> | <entier> <digit>

ou

<entier> ::= <digit> | <digit> <entier>

ou encore

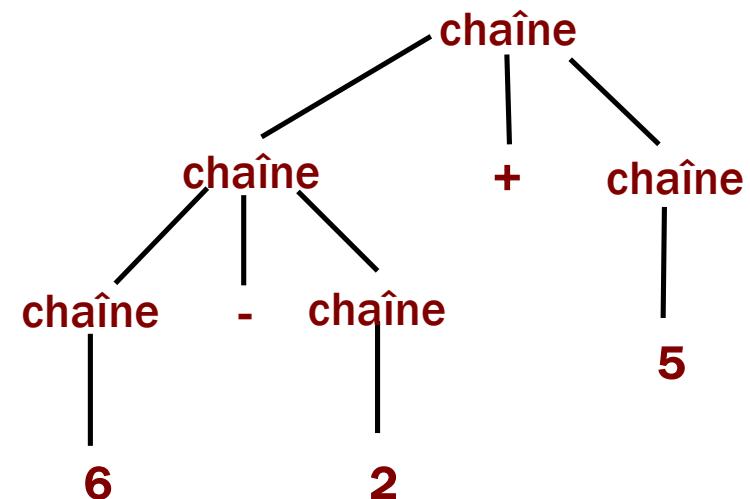
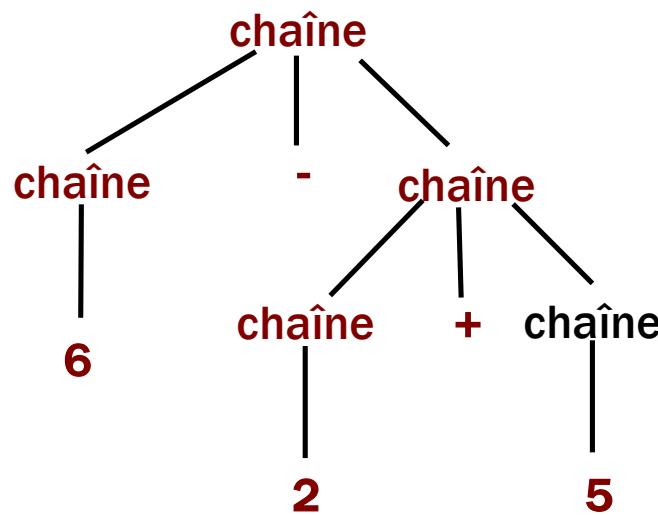
<entier> ::= <digit> { <digit> }

Non unicité de la dérivation

Deux dérивations sont possibles

Remplacer le terme le plus à gauche

Remplacer le terme le plus à droite



Ambiguïté d'une grammaire formelle

Il y a **ambiguïté** dans un langage $L(G)$ s'il existe plusieurs dérivations possibles pour une même phrase

Exemple — considérons les productions suivantes —

$S \rightarrow \text{if } b \text{ then } S \text{ else } S$
 $S \rightarrow \text{if } b \text{ then } S$
 $S \rightarrow a$

L'expression
 $\text{if } b \text{ then if } b \text{ then } a \text{ else } a$
est ambiguë

L'ambiguïté provient du **else**, il faut lever l'ambiguïté en associant le **else** au **then** le plus proche :

$S \rightarrow \text{if } b \text{ then } S_1$
 $S_1 \rightarrow \text{if } b \text{ then } S_2 \text{ else } S_1$
 $S_1 \rightarrow a$
 $S_2 \rightarrow \text{if } b \text{ then } S_2 \text{ else } S_2$
 $S_2 \rightarrow a$

PARTIE 3

OUTILS POUR LA CONCEPTION DES COMPILEURS

Analyseur lexical : lex

Analyseur Syntaxique : yacc

Environnement de programmation

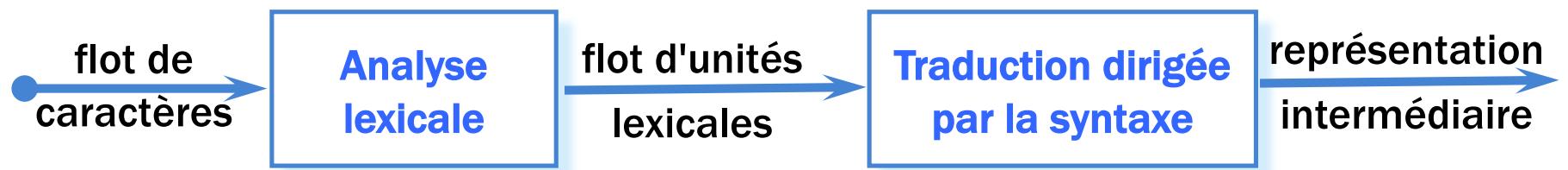
Propos de cette partie

Un langage de programmation est défini par :

- sa syntaxe : structures des programmes écrits dans ce langage,
- sa sémantique : le sens accordé à chaque type de construction selon une structure.

Difficulté de représentation de la sémantique : description souvent informelle.

Etude d'un exemple pour illustrer une démarche de conception d'un compilateur s'appuyant sur la grammaire : la traduction (génération de code en représentation intermédiaire) est dirigée par la syntaxe du langage.



Outils pour la construction des compilateurs

Constructeurs d'analyseurs lexicaux :

- spécification sous forme d'expressions régulières
- l'analyseur généré est un automate à états finis.

Constructeurs d'analyseurs syntaxiques :

- spécification sous forme d'une grammaire non contextuelle,
- phase qui consommait énormément de ressources CPU et humaines,
- algorithmes de génération trop complexes.

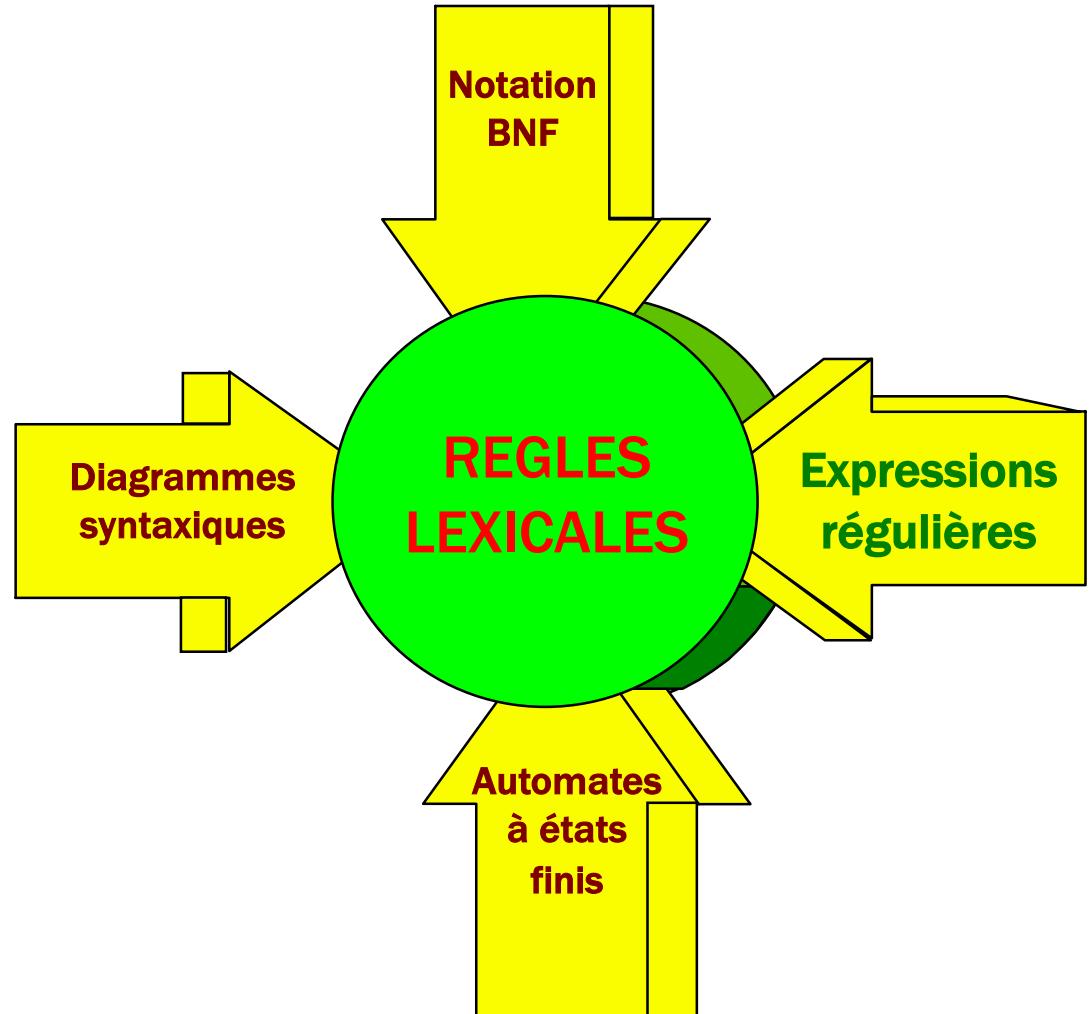
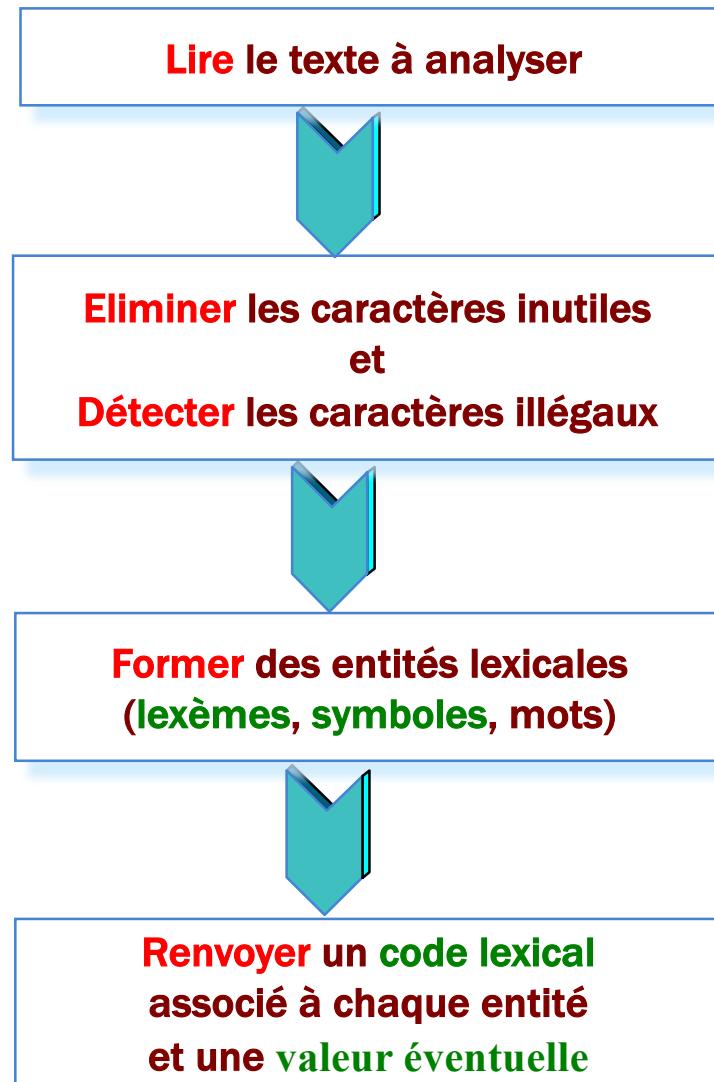
Moteurs de traduction dirigée par la syntaxe :

- collection de procédures qui parcourent l'arbre syntaxique de dérivation tout en produisant du code intermédiaire.

Générateurs automatiques de code :

- collection de règles qui définissent pour chaque opération du langage intermédiaire, sa traduction en code pour la machine cible.
- définition du mode d'adressage : registres, emplacements statiques, emplacement dans une pile,

Analyse lexicale



Expression régulière

Une **expression régulière** décrit un **motif** d'un langage. Les langages sont décrit par des **caractères** : un caractère est une **expression atomique**. Les expressions sont formées par concaténation d'expressions (implicite)

La description des expressions régulières utilise des **méta-caractères** afin de désigner :

- un caractère dit **expression atomique**,
- un **groupement** de caractères dit **expression étendue**,
- la **répétition** d'une expression atomique ou étendue,
- La **disjonction** d'expressions,
- la **position** du motif,
- la **précédence** d'un motif par un autre,

Expression régulière atomique

L'écriture	désigne	Exemple	Commentaire/Remarque
c	le caractère c si c n'est pas un méta-caractère.	x	Le caractère x
\c	le caractère c (si c est un méta caractère) ou un caractère spécial	\n \t	Retour à la ligne Tabulation
"c"	le caractère c, même si celui-ci est un méta-caractère.	["	Le méta-caractère [
[....]	un caractère appartenant à l'ensemble spécifié.	[a-z]	Intervalle de caractères
		[aeiouy]	Énumération de caractères
		[a-zA-Z_0-9]	Intervalle & Énumération
[^....]	un caractère n'appartenant pas à l'ensemble spécifié.	[^ \t]	tout caractère qui n'est pas un espace ni une tabulation
.	tout caractère sauf \n		\. désigne le caractère .
\x...	le code hexadécimal du caractère	\x30	Le caractère 0

Expression régulière étendue & concaténation

La **concaténation** est une écriture naturelle : une suite de caractères. ,

Le regroupement se fait par une mise entre **parenthèses**.

Une **expression étendue** est obtenue soit par concaténation et/ou regroupement d'expressions atomiques et étendues.

Exemple :

(quit) : la chaîne de caractères "quit",

"quit." : la chaîne de caractères "quit.",

Les guillemets **banalisent les métacaractères** mais pas les parenthèses.

Expression régulière – répétition / fermeture

La **fermeture (répétition)** s'applique aussi bien sur des expressions atomiques que étendues.

L'écriture	permet de répéter	Exemple	Commentaire/Remarque
RegExp*	de 0 à une ∞ fois l'expression qui précède.	x*	Toute chaîne de caractères vide ou ne contenant que des x
RegExp+	De 1 à une ∞ fois l'expression qui précède.	[0-9]+	Désigne un entier non signé
RegExp?	de 0 à 1 fois l'expression qui précède.	"-"?	le signe "-" en option
RegExp {n,m}	de n à une m fois l'expression qui précède.	[0-9]{1,2}	Entier < 100
		[0-9]{2}	9 < Entier < 100
		[0-9]{2,2}	
		[0-9]{2, }	Entier > 9

Expression régulière — disjonction / précédence / position

L'écriture	signifie que	Exemple	Commentaire/Remarque
RegExp1 RegExp2	l'on reconnaîtra soit RegExp1, soit RegExp2 .	("+" "-")	Désigne le signe d'un nombre
RegExp1/ RegExp2	l'expression RegExp2 sera reconnue si elle est précédée de l'expression RegExp1.	1/0	Reconnait 0 s'il est précédé de 1
^RegExp	RegExp sera reconnue si elle est en début de chaîne.	^A	La chaîne commence par "A"
RegExp\$	RegExp sera reconnue si elle est en fin de chaîne.	es\$	La chaîne se termine par "es"

Génération d'un analyseur lexical - le langage lex

lex est un langage de spécification d'analyseur lexical.

Un programme en lex, définit un ensemble de motifs qui sont appliqués au flux analysé.

Chaque motif est représenté sous la forme d'une expression régulière.

Lorsque un motif est reconnu, un traitement associé est exécuté.

Le translateur lex permet de traduire un programme écrit en lex en un programme en C équivalent et qui définit la fonction int yylex(void) qui constitue l'analyseur lexical.

Corps d'un programme sous "lex/flex"

```
%{  
<DECLARATIONS C>  
%}  
  
<DECLARATIONS lex>  
%%  
    <REGLES LEXICALES>  
%%  
  
<ROUTINES C>
```

Les définitions placées entre **%{ et %}** sont recopiés par **lex** en **tête du fichier résultat** sans modification.

Une déclaration **lex**, paramètre l'analyseur.
Une règle associe un traitement à une expression régulière reconnue (motif).
Ce bloc définit l'analyseur ; **ylex()**

Les routines sont recopiés par **lex** dans en **fin du fichier résultat** sans modification.

$\langle \text{règles} \rangle ::= \langle \text{règle} \rangle \quad \quad \langle \text{règle} \rangle ; \langle \text{règles} \rangle$	$\langle \text{règle} \rangle ::= \langle \text{RegExp} \rangle \quad \langle \text{TAB} \rangle \quad \langle \text{action} \rangle$
$\langle \text{action} \rangle ::= \langle \text{act} \rangle \quad \quad \{ \quad \langle \text{suite-act} \rangle \quad \}$	$\langle \text{suite-act} \rangle ::= \langle \text{act} \rangle ; \quad \quad \langle \text{act} \rangle ; \langle \text{suite-act} \rangle$

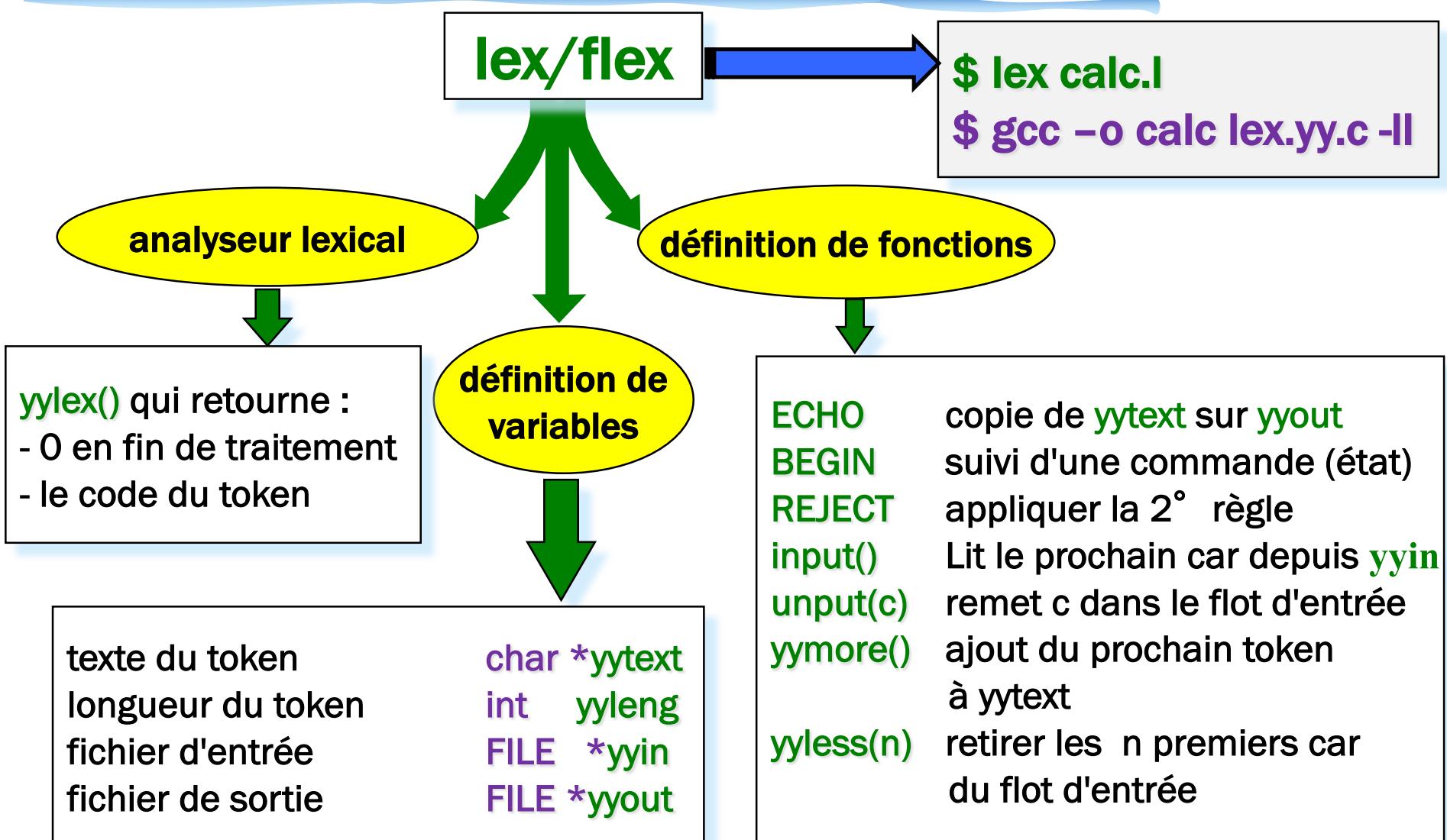
Exemple

```
entier          [0-9]+  
nl              \n  
sortie          (quit|exit|ciao|bye)  
  
%%  
{entier}        ECHO;  
{nl}            ECHO;  
{sortie}        return 0;  
.                ;  
%%  
  
main () {  
    yylex();  
}
```

lex construit un automate de reconnaissance des motifs décrits :

- adéquation avec expressions rationnelles,
- déterminisme,
- minimalité,
- pas de mémorisation de l'état précédent.

“lex/flex”



Un autre exemple / exercice

```
%%  
    int compt=0;  
[ \t] ;  
\n    {compt=0};  
[^ \t\n] {compt++; if (compt==2) ECHO;}  
%%  
  
int main (int argc, char **argv) {  
    if (argc<2) {  
        printf("usage : %s\\  
              filenameToBeAnalysed\n", argv[0]);  
        exit(EXIT_FAILURE);  
    }  
    if (yyin=fopen(argv[1],"r")) yylex();  
    else {  
        printf("Erreur d'ouverture de %s\n", argv[1]);  
        exit(EXIT_FAILURE);  
    }  
    exit(EXIT_SUCCES);  
}
```

Implémentez ce code et testez le :

```
$ lex exemple.l  
$ gcc lex.yy.c -o exemple  
$ ./exemple
```

Que fait ce programme ?

Inspirez vous de ce code et écrivez l'équivalent de la fameuse commande **wc** d'UNIX.

Exemple fil conducteur : calculatrice

Réalisation d'une calculatrice en mode interprété (partie frontale) :

- Analyse lexicale suivie d'une analyse syntaxique et une interprétation de l'expression algébrique.
- Une expression algébrique infixée $5+2*3$ sera traduite en notation post-fixée $5\ 2\ 3\ *\ +$ où les opérateurs apparaissent après leurs opérandes. Ainsi l'évaluation est immédiate (pile).

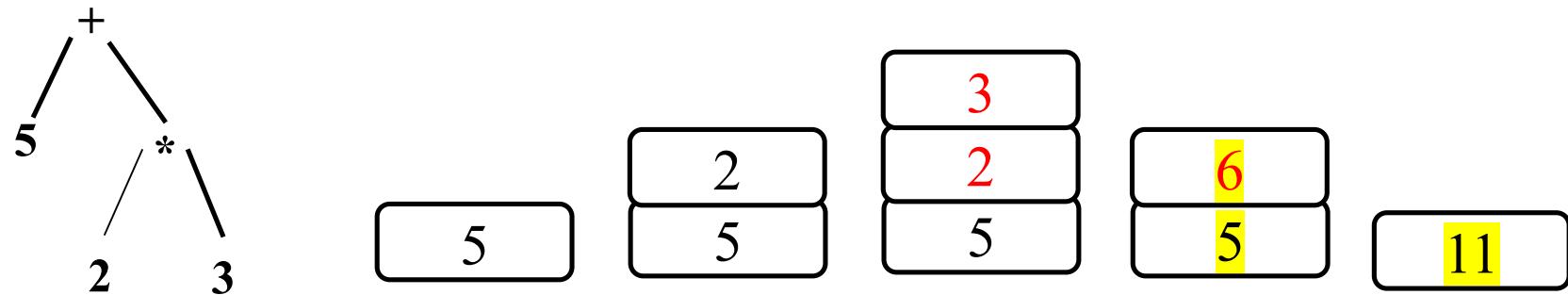
La réalisation se fera de manière progressive

- Réalisation d'un programme qui traduit des expressions formées uniquement de chiffres et des opérateurs '+' et '-'.
- Amélioration de cette version par la prise en compte des constructions lexicales :
 - ▶ Nombres,
 - ▶ Accumulateurs,
 - ▶ identificateurs de variables et mots-clés

Exercice : calc.l (calc V0.1 & V0.2)

Ecrire un code **lex** qui reconnaît uniquement les **chiffres** et les **opérateurs '+' et '-'** et afficher les unités reconnues : **ECHO, yytext** (version 0.1).

Transformer cette version pour reconnaître des **entiers** (version 0.2).



GDR (polonaise) : 5 2 3 * +

Un nombre : on empile

Un opérateur : on dépile deux valeurs (opérandes), on fait l'opération, on empile le résultat

L'analyse syntaxique

Construire un arbre syntaxique à partir des unités lexicales :
Une définition rigoureuse de la grammaire est fondamentale



Depuis l'axiome, **créer** une chaîne identique à la chaîne d'entrée
à analyser en procédant par **dérivations successives** :

Remplacer un symbole non terminal par une de ses parties droites
Attention au problème du choix de la partie droite :
Traiter de gauche à droite (on s'intéresse au symbole courant,
les symboles de droite sont à venir)

Technique de traitement des tokens :

- **substituer** le token courant à la règle,
- **générer** une nouvelle cible,
- **lire** le prochain token,

Génération d'un analyseur syntaxique – le langage yacc

yacc (Yet Another C Compiler) est un langage d'écriture d'**analyseur syntaxique**.

Un code yacc définit la **grammaire** du langage en spécifiant son **axiome** et les **règles de production**.

Une règle de production décrit une (plusieurs) **dérivation(s)** possibles de l'**unité syntaxique**.

A chaque dérivation d'une unité syntaxique, un **traitement** est associé : **traduction dirigée par la syntaxe**.

yacc (translateur) permet de traduire un programme écrit en yacc en un programme en **C** équivalent et qui définit la fonction **int yyparse(void)** qui constitue l'**analyseur syntaxique**.

Corps d'un programme “yacc/bison” ...

La structure d'un code yacc est très **similaire** à celle de lex.

La partie déclarative pour yacc est **plus riche** que celle de lex.

```
%{  
<DECLARATIONS C>  
%}  
  
<DECLARATIONS yacc>  
%%  
<REGLES SYNTAXIQUES>  
%%  
  
<ROUTINES C>
```

Les définitions placées entre **%{ et %}** sont recopier par yacc en **tête du fichier résultat** sans modification.

Une déclaration yacc, paramètre l'analyseur syntaxique et une règle associe un traitement à une **unité syntaxique reconnue**. Ce bloc définit l'analyseur ; **yyparse()**

Les routines sont recopier par yacc dans en **fin du fichier résultat** sans modification.

... Corps d'un programme “yacc/bison”

Une déclaration yacc est une ligne de la forme :

%mot-clé <développement>

%start expr // axiome

%token RC NBR // déclaration des tokens renvoyés par yylex()

Grammaire yacc pour la spécification des règles syntaxiques d'un langage :

<règles> ::= <règle> ';' | <règle> ';' <règles>

<règle> ::= <unité syntaxique> ':' <dérivations>

<dérivations> ::= <dérivation> | <dérivation> '|' <dérivations>

<dérivation> ::= <séquence d'unités lexicales et syntaxiques & traitements>

<traitement> ::= '{' <code C> '}'

Exemple

```
%start list
%token NBR

%%
list : // vide
      | list expr '\n'
      | list '\n'
      ;
expr : NBR '+' NBR { printf("addition\n"); }
      | NBR '-' NBR { printf("soustraction\n"); }
      ;
%%

main () {
    yyparse();
}
```

yacc construit un analyseur (**yyparse()**) de reconnaissance d'une suite d'**expressions algébriques** simples (addition/soustraction de deux nombres), séparées par des **retours à la ligne**.

yyparse() fait appel à **yylex()** qui fournit les tokens : **NBR**, **'+'**, **'-'**, **'\n'**

Génération d'un analyseur syntaxique – le compilateur yacc/bison

Un analyseur syntaxique fait appel à un analyseur lexical.

yacc génère un fichier résultat nommé **y.tab.c**.

L'option **-d** permet de générer le fichier **y.tab.h**. Ce fichier est inclus par l'analyseur lexical.

bison réutilise le nom du fichier source pour les fichiers générés.

```
$ yacc [-d] calc.y  
$ flex calc.l  
$ gcc y.tab.c yy.lex.c -o calc -ll -ly
```

```
$ bison [-d] calc.y  
$ flex calc.l  
$ gcc calc.tab.c yy.lex.c -o calc -ll -ly
```

Options de compilation :

- d** générer **y.tab.h**
- t** créer le symbole **yydebug** pour le mode **debug**

Valeur sémantique

yacc définit le type **YYSTYPE** qui vaut **int** par défaut. Il est ainsi possible d'attribuer une **valeur sémantique** à un token (unité lexicale) reconnue par l'intermédiaire de la variable **yylval**.

YYSTYPE peut être redéfini par une déclaration **%union** en déclarant une liste des **types sémantiques** manipulés par le langage. Il faut ensuite attribuer un type sémantique à chaque unité :

%token permet d'attribuer un type à une unité lexicale parmi les types définis par **YYSTPE**.

%type permet d'attribuer un type à une unité syntaxique parmi les types définis par **YYSTPE**.

Registres

yacc utilise des **registres** de type **YYSTYPE** pour désigner la valeur sémantique d'une unité lexicale ou syntaxique.

A chaque unité de la règle est associé un registre **\$x**, où **x** est le numéro d'ordre de celle-ci dans la règle.

L'unité syntaxique développée (dérivée) es valeurs utilise le registre **\$\$**.

```
...
expr   : NBR '+' NBR { $$ = $1 + $2 ; }
        | NBR '-' NBR { $$ = $1 - $2 ; }
;
...
...
```

la valeur d'un token
est mémorisée dans
yylval (par **yylex()**)

Que définit yacc ?

yacc définit	comme	Commentaire/Remarque
<code>yyparse()</code>	analyseur syntaxique.	<code>return</code> attribue une valeur de retour
<code>YYSTYPE</code>	type pour les valeurs sémantiques des unités lexicales et syntaxiques.	Ce type est défini par <code>%union</code> et utilise par <code>%token</code> et <code>%type</code> <code>YYSTYPE</code> est défini <code>int</code> par défaut.
<code>yylval</code>	valeur sémantique renseignée par l'analyseur lexical	Doit être déclaré pour <code>lex</code> : <code>extern int yylval;</code>
<code>error</code>	Unité syntaxique qui permet de reconnaître une erreur syntaxique et de la traiter par le langage.	<code>list</code> : // vide <code>list error '\n'</code> ...
<code>yyerror()</code>	traitement invoqué en cas d' erreur syntaxique .	Ce traitement peut être redéfini : <code>void yyerror (char *....) {.....}</code>
<code>yyerrok</code>	traitement à invoquer pour repartir avec une pile d'analyse syntaxique propre.	<code>list</code> : // vide <code>list error '\n' { yyerrok; }</code> ...
<code>YYERROR</code>	macro. Son invocation assimile une erreur sémantique à une erreur syntaxique .	

Exercice – calc.y (calc v0.3 & v0.4)

Ecrire un code **yacc** pour la version 0.2 de **calc.l** pour obtenir la version 0.3. Cette version **déclare** les **tokens** reconnus par **lex** et utilise **yylval** pour fournir une valeur sémantique aux **registres**. Il ne faut pas oublier d'inclure **y.tab.h** dans **calc.l**.

Ajouter à cette version **26 accumulateurs** reconnus par les **lettres minuscules** (version 0.4).

Exercice – calc v0.5 & v0.6

Modifier la version 0.4 pour traiter les réels avec 26 accumulateurs reconnus par ses lettres majuscules (version 0.5). Cette version redéfinit YYSTYPE par %union. Les tokens doivent être typés. expr doit l'être aussi par %type. L'analyseur lexical renseignera le champ adéquat dans yyval selon la valeur sémantique de l'unité reconnue.

Comment différencier le cas des entiers de celui des réels ?

Modifier la version 0.5 pour rendre les erreurs syntaxiques acceptables en les traitant par une règle syntaxique : error et yyerrok afin de les considérer comme valide et de ne pas interrompre l'interprétation (version 0.6).

calc – v0.1 à v0.6

Version	Module	Description
0.1	LEX	Règles lexicales de reconnaissance des chiffres, opérateurs '+' et '-' , RC, espaces, ...
0.2	LEX	Règles lexicales de reconnaissance des entiers
0.3	YACC	Règles syntaxiques de reconnaissance des expressions algébriques & Evaluation à base de registres
	LEX	Valeur sémantique d'un nombre : <code>yyval</code>
0.4	LEX/YACC	Utilisation d'un tableau d'accumulateurs (entiers) désignés par les minuscules
0.5	LEX	Règles lexicales de reconnaissance des réels
	YACC	Typage des expressions algébriques (<code>YYSTYPE</code> et <code>%type</code>) pour l'évaluation expressions réelles
	LEX/YACC	Utilisation d'un tableau d'accumulateurs (réels) désignés par les majuscules
0.6	YACC	Reconnaissance des erreurs syntaxiques : <code>error</code> , <code>yyerror()</code> & <code>yyerrok</code>

Associativité des opérateurs

La majorité des langages de programmation définissent des **opérateurs** algébriques, de logique, de comparaison, binaires, ...

Quand le **même opérateur** est utilisé dans une écriture, il y'a indécision sur l'interprétation à donner. On parle d'**associativité** (c'est la même notion qu'en mathématiques).

Ainsi, les opérateurs arithmétiques sont associatifs à gauche :

- $9+5+2$ est équivalent à $(9+5)+2$,
- $9-5-2$ est équivalent à $(9-5)-2$,
- L'exponentiation est associative à **droite**,
- L'opérateur d'affectation en C est associatif à droite. Ainsi, $a=b=c$ est traitée de la même manière que $a=(b=c)$.

yacc — Associativité

yacc utilise	pour	Exemple
%left	déclarer un opérateur associatif à gauche .	%left '+' '-' %left '*' '/'
%right	déclarer un opérateur associatif à droite .	%right '^'
%nonassoc	déclarer un opérateur non associatif .	%nonassoc UMINUS
%prec	établir une préférence d'un opérateur dans une règle syntaxique par rapport à toutes les autres. Ce mécanisme n'est pas une déclaration mais une invocation par une règle syntaxique.	... %prec UMINUS ...

Priorité des opérateurs

Quand **différents opérateurs** sont utilisés dans une écriture, il y'a indécision sur l'interprétation à donner. On parle de **priorité entre opérateurs** (c'est la même notion qu'en mathématiques).

- En arithmétique, la **multiplication** et la **division** ont une priorité **supérieure** à l'**addition** et la **soustraction**.
- L'**affectation** est l'opérateur le **moins** prioritaire.

Il est donc nécessaire de définir la priorité entre opérateurs pour lever les **ambiguïtés** d'interprétation. Généralement, on dresse un **tableau ordonné des priorités** des opérateurs. yacc utilise l'**ordre de déclaration** pour déterminer la priorité entre opérateurs.

Exercice – calc v0.7 & v0.8

Modifier la version 0.6 pour traiter les opérateurs '*' et '/' ainsi que le signe (ne pas traiter le signe par l'analyseur lexical) pour avoir la version 0.7.

La version 0.7 ne gère pas le cas de la division par zéro. Il faut utiliser YYERROR pour provoquer une erreur (version 0.8).

Version	Module	Description
0.7	LEX	Règles lexicales de reconnaissance des signes '*' et '/'
	YACC	Associativité & priorités Règle syntaxique pour les nombres signés : précédence UMINUS
0.7.1	LEX/YACC	Reconnaissance des parenthèses
0.8	YACC	Assimiler les erreurs sémantiques à des erreurs syntaxiques : YYERROR

calc v0.9

Cette version va introduire les **identificateurs** de variables à la place des accumulateurs. Ce qui nécessite la **mise en place d'une table de symboles** (gérée en liste chaînée) :

- Définition d'un type "**typeSymbol_t**" pour qualifier le type de la variable manipulée :
 - ▶ Variable indéfinie : **UNDEF**
 - ▶ Variable entière : **IVAR**
 - ▶ Variable réelle : **FVAR**
- Définition d'un type "**symbol_t**" avec les attributs :
 - ▶ **Nom** du symbole,
 - ▶ **Le type** du symbole,
 - ▶ **Une valeur entière** (**int**) pour le type **IVAR** ou **une valeur réelle** (**double**) pour le type **FVAR**,
 - ▶ Un pointeur sur le **symbole suivant** : **liste chaînée de symboles**.

Gestion de la table des symboles – symbol.h (1)

```
/**\n * \struct      symbol_t\n * \brief       Définition du type de données "symbole"\n */\ntypedef struct symbol {\n    char* name;           // Nom du symbole : identifiant\n    short type;          // Type du symbole : IVAR, FVAR, UNDEF\n    union {\n        int   iVal;        // Selon le type du symbole :\n        double dVal;       // une valeur entière si le type est IVAR\n                           // une valeur réelle si le type est FVAR\n    } U;\n    struct symbol* next;  // Pointeur sur symbole suivant\n} symbol_t;\n/**\n * \typedef     pSymbol_t\n * \brief       Définition du type de données "pointeur sur symbole"\n */\ntypedef struct symbol* pSymbol_t;\n/**\n * \def         SYMBOL_NULL\n * \brief       Définition du symbole nul\n */\n#define SYMBOL_NULL ((pSymbol_t) 0)
```

Gestion de la table des symboles – symbol.h (2)

```
/**  
 * \fn    symbol_t* installSymbol (char* tokenName, short tokenType)  
 * \brief Insérer un nouveau symbole en tête de la liste des symboles  
 * \return pointeur sur symbole inséré  
 */  
symbol_t* installSymbol (char* tokenName, short tokenType) ;  
/**  
 * \fn    symbol_t* lookUpSymbol (const char *tokenName)  
 * \brief Rechercher un symbole dans la liste des symboles  
 * \return pointeur sur symbole recherché ou NULL si non trouvé  
 */  
symbol_t* lookUpSymbol (const char* tokenName) ;
```

Gestion de la table des symboles – symbol.c

```
/**  
 * \var      _symbolList  
 * \brief    Liste des symboles  
 */  
  
static pSymbol_t _symbolList = SYMBOL_NULL ;  
  
symbol_t* installSymbol (char* tokenName, short tokenType) {  
    symbol_t *mySp = (symbol_t *) malloc(sizeof(symbol_t));  
    mySp->type = tokenType;  
    mySp->name = (char *) malloc(strlen(tokenName)+1);  
    strcpy(mySp->name, tokenName);  
    mySp->next = _symbolList;  
    _symbolList = mySp;  
    return mySp;  
}  
symbol_t* lookUpSymbol (const char* tokenName) {  
    symbol_t *mySp = _symbolList;  
    for ( ; mySp != SYMBOL_NULL; mySp = mySp->next)  
        if (strcmp(mySp->name, tokenName) == 0) return mySp;  
    return SYMBOL_NULL;          // token non trouvé  
}
```

calc v0.9

Version	Module	Description
0.9	LEX	Règles lexicales de reconnaissance des identificateurs (TS)
	SYMBOL	Gestion des symboles : définition du type symbol_t , de installSymbol() et lookUpSymbol()
	YACC	Définition de YYSTYPE (%union) : typage des tokens & des unités syntaxiques
	YACC	Règles syntaxiques d'accès aux variables, affectation de variables

Cette version nécessite :

- La modification du type **YYSTYPE** ainsi que le typage des unités lexicales et syntaxiques.
- La modification des valeurs sémantiques des tokens,
- La mise en place des règles lexicales pour les identificateurs.
- La mise en place des règles syntaxiques pour traiter identificateurs (accès, affectation).

Exercice – calc v0.9

Version	Module	Description
1.0	LEX/YACC	Reconnaissance de opérateurs logiques & de comparaison

Améliorer la version 0.9 pour reconnaître les opérateurs logiques et de comparaison (version 1.0)



PARTIE 4

UN COMPILATEUR PAR L'EXEMPLE

Hoc v1.0

Le point de départ de cette version est la dernière version de **calc** (càd 0.9) : sans les opérateurs logiques et de comparaison.

L'objectif est d'utiliser le **type générique (void *)** pour typer la sémantique.

Les améliorations à apporter :

- Traitement des nombres **comme** des symboles,
- Classification des symboles (**classSymbole_t**),
- Typage des valeurs sémantiques (**typeSymbole_t**),
- Harmonisation de la valeur (lexicale) d'un token défini par **yacc** et **symbol.h**,
- Installation des symboles **constants**.

Généricité en C – void *

Le type **void *** est un pointeur mémoire (adresse de début) sur une donnée quelconque (type générique) : la taille mémoire pointée et sa structure sont inconnues.

déclarations		Exemple
void *g; type_t *pv; type_t v;	// Echange de pointeurs g = (void *) pv; g = (void *) &v; pv = (type_t *) g;	void * g; int i; int *pi=&i; ... g = (void *) pi; g = (void *) &i;
	// Echange de valeurs pointées *pv = *(type_t *) g; // v ne peut être affectée directement // par la valeur pointée par g memcpy((void*)&v, g, sizeof(type_t)); memcpy((void*)pv, g, sizeof(type_t));	void * g; double d; double *pd; ... pd = (double *)g; *pd = *(double *)g; memcpy((void*)&d, g, sizeof(double));
void *g, *aux; int v;	v = (int)*(void **) g; aux = *(void **) g;	Un déréférencement de g pour les entiers

Hoc v1.0 – TS : Classe & type

Cette version nécessite la mise en place d'une gestion de symboles plus avancée :

- Définition d'un type "**classSymbol_t**" pour qualifier le symbole manipulé : **DAT**, **CST**, **VAR**, **PRG**,
- Définition d'un type "**typeSymbol_t**" pour qualifier le type du symbole manipulé : **UNDEF**, **ENTIER**, **REEL**,
- Définition d'un type "**symbol_t**" avec les attributs :
 - ▶ **Nom** du symbole,
 - ▶ La **classe** et le **type** du symbole,
 - ▶ Un **pointeur générique** :
 - ⇒ sur la **valeur** avec le type adéquat (**VAL**, **CST**, **VAR**)
 - ⇒ sur une **fonction** (**PRG**),
 - ▶ La **taille mémoire** occupée par la valeur,
 - ▶ Un **pointeur sur le symbole suivant** : liste chaînée de symboles.

Hoc v1.0 – TS : symbol.h

```
/**  
 * \struct generic  
 * \brief Définition du type de données générique  
 */  
typedef void * generic;  
/**  
 * \typedef symbol_t  
 * \brief Définition du type de données "symbole" de type struct  
 */  
typedef struct symbol {  
    char *name;           // Nom du symbole : identifiant  
    short clas;          // Classe du symbole : DAT, CST, VAR  
    short type;          // Type du symbole : ENTIER, REEL, UNDEF  
    generic pValue;       // Pointeur générique sur la valeur du symbole  
    short size;          // Taille mémoire occupée par la valeur  
    struct symbol *next; // Pointeur sur symbole suivant  
} symbol_t;
```

Hoc v1.0 – TS : symbol.c – installSymbol()

```
/**  
 * \fn     symbol_t* installSymbol (char * tokenName, short tokenClas, short tokenType,  
 *                                 short tokenSize, generic pValue)  
 * \brief  Insérer un nouveau symbole en tête de la liste des symboles  
 * \return Pointeur sur le symbole inséré */  
symbol_t* installSymbol (char* tokenName, short tokenClas, short tokenType,  
                        short tokenSize, generic tokenPtrValue) {  
    symbol_t *mySp = (symbol_t *) malloc(sizeof(symbol_t));  
    mySp->clas = tokenClas;  
    mySp->type = tokenType;  
    mySp->size = tokenSize;  
    mySp->pValue = tokenPtrValue;  
    mySp->name = NULL;                                // Cas d'un symbole sans nom  
    if (tokenName != NULL) {  
        mySp->name = (char *) malloc(strlen(tokenName)+1);  
        strcpy(mySp->name, tokenName); }  
    mySp->next = _symbolList;  
    _symbolList = mySp;  
    return mySp;  
}
```

Hoc v1.0 – TS : déclaration des constantes

```
/**  
 * \var    _consts[]  
 * \brief Déclaration des variables (constantes mathématiques) prédéfinies  
 */  
static struct {  
    char *cName; short cType;  
    double cValue; char* cDesc;}  
  
_consts[] = {  
    "PI",      REEL,   3.14159265358979323846, "Archimede's constant",  
    "E",       REEL,   2.71828182845904523536, "Euler's constant",  
    "GAMMA",   REEL,   0.57721566490153286060,  "Another Euler's constant",  
    "DEG",     REEL,   57.2957795130823208768, "360/2*Pi",  
    "PHI",     REEL,   1.61803398874989484820, "Golden ratio",  
    "MAXSTACK", ENTIER, 256,                      "Taille de la pile d'exe",  
    "MAX_PROG", ENTIER, 2000,                     "Taille de la machine d'exe",  
    NULL,      0,      0,                          NULL  
}
```

Hoc v1.0 – TS : defSymbols.c – *installDefaultSymbols()*

```
/**  
 * \fn      void installDefaultSymbols (void)  
 * \brie flnstalle les symboles par défaut dans la table des symboles :  
 *          <UL><LI>Constantes : PI, E, ...</LI>  
 *          <LI>Fonctions mathématiques : sin(), cos(), ...  
 *          </UL> */  
void installDefaultSymbols (void) {  
    int * pInt;  
    double * pFlo;  
    for (int i = 0; _consts[i].cName!=NULL; i++)  
        if (_consts[i].cType==ENTIER) {  
            pInt = (int *) malloc (sizeof(int));  
            *pInt = (int) _consts[i].cValue;  
            installSymbol(_consts[i].cName, CST, _consts[i].cType, sizeof(int), pInt);  
        }  
        else if (_consts[i].cType==REEL) {  
            pFlo = (double *) malloc (sizeof(double));  
            *pFlo = _consts[i].cValue;  
            installSymbol(_consts[i].cName, CST, _consts[i].cType, sizeof(double), pFlo);  
    }  
}
```

Hoc v1.0 – Les nombres sont des symboles – hoc.l

```
%{  
int installIntSymbol() {  
    int *pValue = (int *)malloc (sizeof(int));  
    *pValue = atoi(yytext);  
    yyval.symb = installSymbol("", DAT, ENTIER, sizeof(int), pValue);  
    return (yyval.symb)->type;  
}  
int installDoubleSymbol() {  
    double *pValue = (double *)malloc (sizeof(double));  
    *pValue = atof(yytext);  
    yyval.symb = installSymbol("", DAT, REEL, sizeof(double), pValue);  
    return (yyval.symb)->type; }  
...  
entier      [0-9]+  
reel        ([0-9]+\.?)|([0-9]*\.[0-9]+)  
%  
{entier}    return installIntSymbol();  
{reel}      return installDoubleSymbol();  
...}
```

installIntSymbol() et
installDoubleSymbol()
implémentent des
traitements similaires à
installDefaultSymbols()

Hoc v1.0 – Les nombres sont des symboles – hoc.y

```
%{ ...
int isFloat = 0; /* Expression entière ou réelle*/
%}
/* YYSTYPE : Définition des valeurs sémantiques (yyval renseignée par lex) */
%union {
    int      entier;
    double   reel;
    pSymbol_t symb;
}
/* Tokens fictifs : déclaration des classes */
%token DAT CST VAR
/* Tokens nombres */
%token <symb> ENTIER REEL
/* Token pour identifiant non reconnu */
%token <symb> UNDEF
/* Tokens pour les variables : passage de UNDEF vers IVAR/FVAR */
%token <symb> IVAR FVAR
/* Typage des unités syntaxiques */
%type <reel> expr assgn
```

Hoc v1.0 – Assiguation d'une variable – hoc.y

```
%%
...
assgn :
    | IVAR AFF expr { $$ = *(int *) $1->pValue = $3; }
    | FVAR AFF expr { $$ = *(double *) $1->pValue = $3; isFloat=1; }
    | UNDEF AFF expr {
        $1->clas = VAR;
        $1->type= (isFloat) ? FVAR : IVAR ;
        $1->size = (isFloat) ? sizeof(double) : sizeof(int) ;
        if (isFloat) {
            $1->pValue = (double*) malloc (sizeof(double));
            $$ = *(double *) $1->pValue = $3;
        } else {
            $1->pValue= (int*) malloc(sizeof(int));
            $$ = *(int *) $1->pValue = $3;
        }
    }
;
```

Le type de la variable
IVAR/FVAR est défini à
posteriori par l'analyseur
syntaxique

Hoc v1.0 – Accès à une variable – hoc.y

```
%%
...
expr :
| ENTIER { $$ =*(int *) $1->pValue; }
| REEL   { memcpy( (generic)&$$, $1->pValue, $1->size); isFloat=1; }
| IVAR    { $$=*(int *) $1->pValue; }
| FVAR    { $$ =*(double *) $1->pValue; isFloat=1; }

...
/* Appel de fonction prédefinie */
| PREDEF PO expr PF { $$ = ( *($1->pValue) ) ($3); isFloat=1; }
```

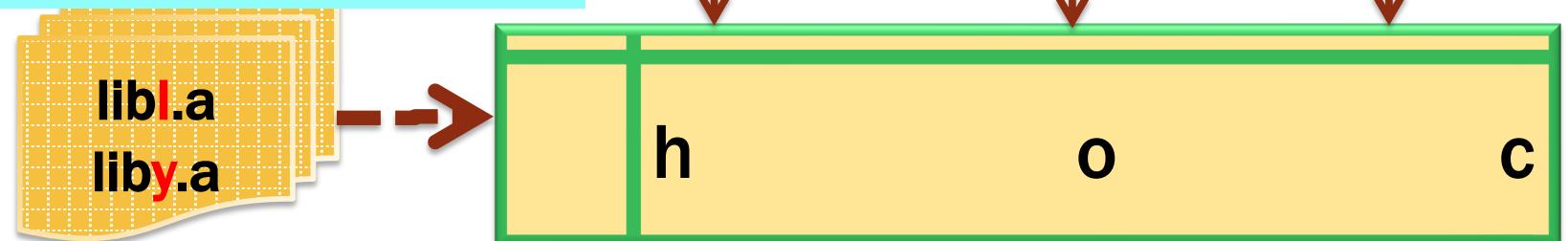
Structure & environnement de développement d'un langage interpréte

yyparse() : Analyseur syntaxique écrit en yacc

yylex() : Analyseur lexical écrit en lex

Implémentation des différents traitements relatifs à l'application développée :

- **symbol.c/defSymbols.c** implémentent une **gestion de symboles** utilisée par tout compilateur/interpréteur,
- **error.c** est une autre implémentation courante pour la **gestion des erreurs**.



hoc v1.1 – Mise en place d'un fichier d'inclusion unique – hoc.h

Chaque fichier d'implémentation (**file.c**) inclut :

- son propre fichier de spécifications : **file.h**,
- des fichiers de spécifications standards : **unistd.h**, **stdlib.h**, **stdio.h**, **string.h**, **math.h**, **errno.h**,
- des fichiers de spécifications définis dans le cadre de l'application : **symbol.h**, **hoc.tab.h**, (**io.h**, **code.h**, ...)

Chaque fichier de spécifications (**file.h**) définit un symbole de type **_FILE_H_**, on obtient ainsi :

- **_SYMBOL_H_, _IO_H_, _CODE_H_, _CODE_HOC_H_, ...**

Un fichier de spécifications **global et unique** (**hoc.h**) donnera le **séquencement des inclusions** à faire par chaque fichier d'implémentation, en exploitant le statut (**défini/non-défini**) de ces **symboles** grâce aux directives du préprocesseur C.

Ainsi, chaque fichier d'implémentation devra inclure **hoc.h**.

Hoc v1.1 – hoc.h

```
#ifndef _HOC_H_
#define _HOC_H_

#ifndef _HOC_L_
#include "symbol.h"
#include "hoc.tab.h"
#endif

#ifndef _HOC_Y_
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "symbol.h"
#endif
#ifndef _SYMBOL_C_
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "symbol.h"
#include "hoc.tab.h"
#endif
#endif /* _HOC_H_ */
```

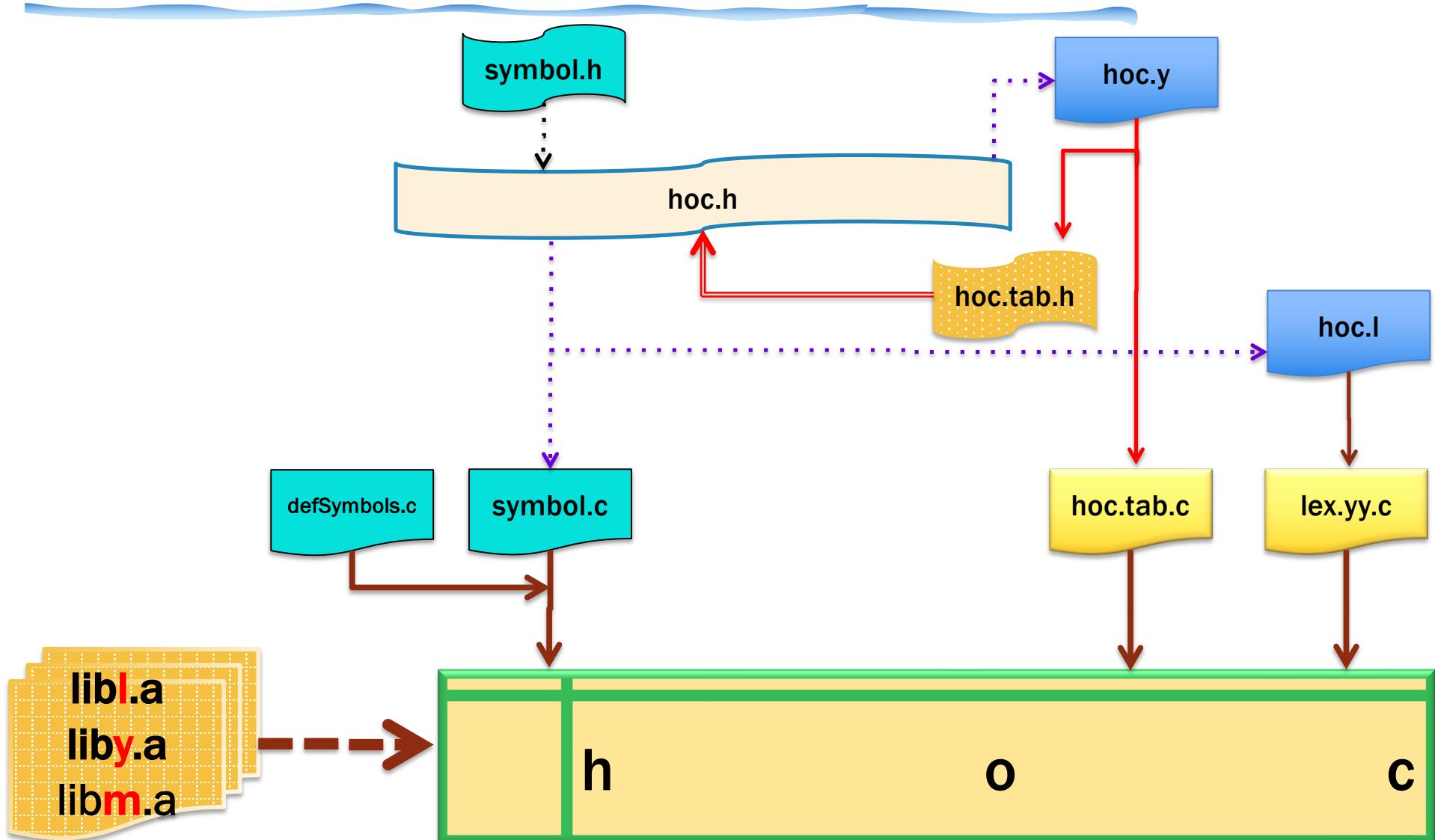
// inclusion par **hoc.l**

// inclusion par **hoc.y**

// inclusion par **symbol.c**

On utilise le même principe pour **hoc.l** et **hoc.y**

hoc v1.1



Hoc – v1.0 & v1.1

Version	Module	Description
1.0	LEX	Reconnaissance de nombres (entiers et réels), d'opérateurs algébriques, de parenthèses, d'identificateurs (constantes, variables, fonctions), RC et élimination des espaces et de tout caractère non spécifié (n'appartient pas au langage)
	YACC	Reconnaissance et évaluation d'expressions algébriques avec variables entières/réelles
	SYMBOL	Transformation du type symbol_t pour gérer la valeur avec le type générique Transformation de installSymbol() pour supporter le type générique Déclaration de constantes : installDefaultSymbols()
1.0.1	SYMBOL	Transformation du type symbol_t pour différencier un pointeur de valeur d'un pointeur de fonction Adaptation de installSymbol() Déclaration de fonctions prédéfinies : installDefaultSymbols()
	YACC	Adaptation des écritures d'accès à pValue en le distinguant de pFct Déclaration de PRG et PREDEF Ecriture d'une règle pour appel de fonction prédéfinie
1.1		Utilisation d'un SEUL fichier de spécifications : hoc.h

Hoc V1.2 – Gestion des erreurs

On distingue trois sortes d'erreurs : lexicale, syntaxique et sémantique.

Une fonction d'affichage par sorte d'erreur avec coloration différenciée.

Cette version utilisera une **table de messages** d'erreurs, numérotés avec des **formats** différenciés. Chaque message utilisera un **style prédéfini** (couleur de fond, couleur du texte, effet).

Hoc v1.2 – Couleurs

	style		Couleur TEXTE		Couleur FOND	S	C F	C T	
0	RESET	30	Noir	40	Noir	T	0 0	0 X	R ô l e
1	Gras	31	Rouge	41	Rouge	Y	U N	U T	
2	Gris	32	Vert	42	Vert	L	L D	L	
3	Italique	33	Jaune	43	Jaune		6 47	30	Couleur par défaut
4	Souligné	34	Bleu	44	Bleu		7 44	37	Résultat entier
5	Clignotant	35	Magenta	45	Magenta		7 42	37	Résultat réel
6	Normal	36	Cyan	46	Cyan		6 47	31	Erreur syntaxique
7	Fond inversé	37	Blanc/gris	47	Blanc/gris		6 47	35	Erreur sémantique
							6 47	36	Erreur lexicale
							7 41	37	DEBUG AS
							7 45	37	DEBUG sémantique
							7 36	37	DEBUG AL

`printf("\033[%im", paramètre)`

`printf("\033[%i;%im", par1, par2)`

...

`printf("\033[H\033[2J") // efface écran`

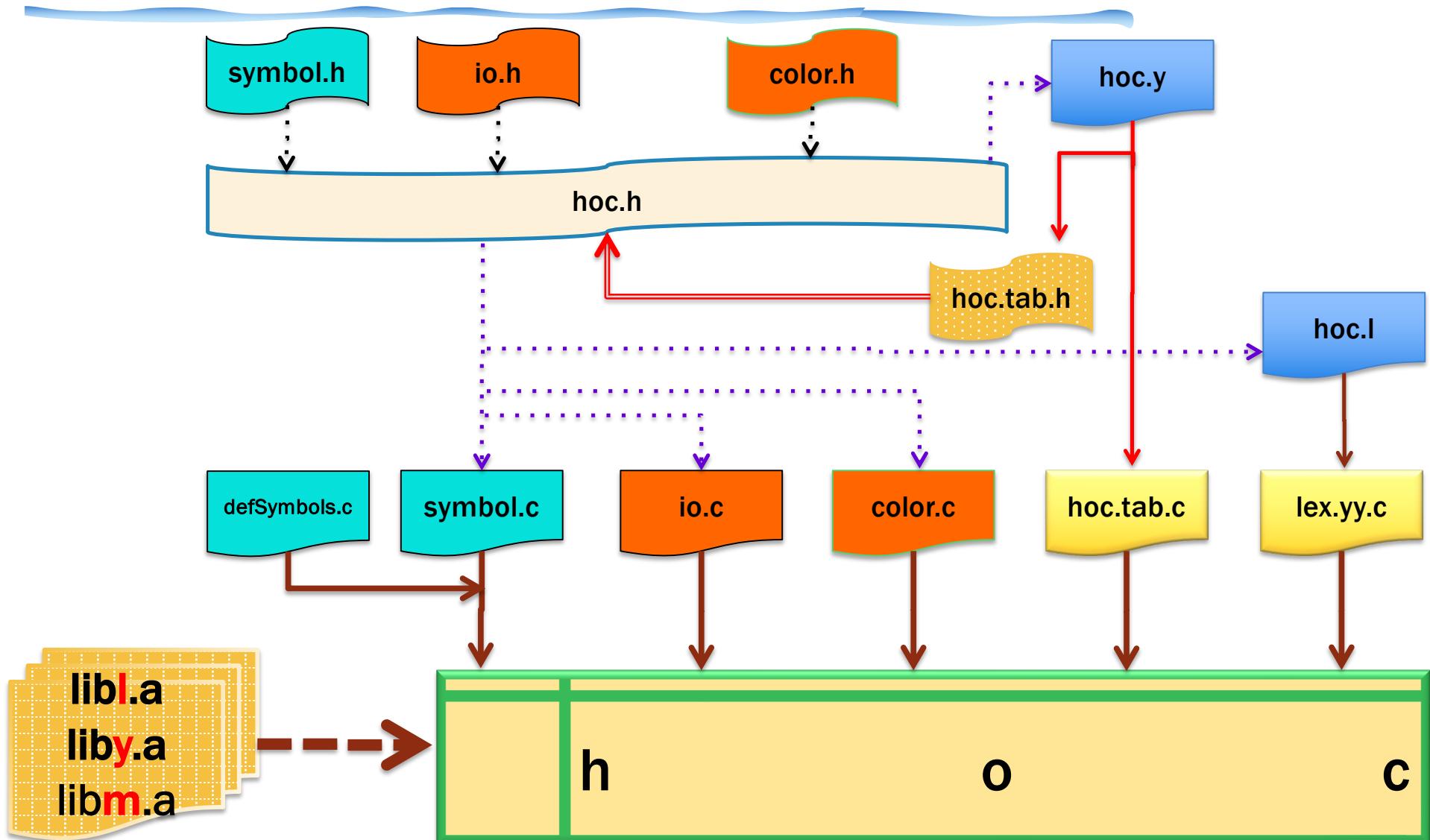
Hoc v1.2 – MAKEFILE

```

YFLAGS=-d
SRCS=hoc.l hoc.y hoc.h symbol.h symbol.c defSymbols.c color.h color.c io.h io.c msgFmt.c
OBJS=hoc.o hoc.l.o symbol.o color.o io.o
CC=gcc
hoc: $(OBJS)
    gcc $(OBJS) -o hoc -ll -ly -lm
    make clean
hoc.o: hoc.h symbol.h io.h hoc.y
    bison $(YFLAGS) hoc.y
    mv hoc.tab.c hoc.c
    gcc -c hoc.c
hoc.l.o: hoc.h hoc.tab.h symbol.h io.h hoc.l
    flex -l hoc.l
    gcc -c lex.yy.c -o hoc.l.o
    rm lex.yy.c
symbol.o: hoc.h hoc.tab.h symbol.h symbol.c defSymbols.c
    gcc -c symbol.c
color.o: hoc.h color.h color.c
    gcc -c color.c
io.o: hoc.h symbol.h color.h io.h io.c msgFmt.c
    gcc -c io.c
clean:
    rm -f $(OBJS) hoc.tab.h hoc.c lex.yy.c *~
edit:
    gedit $(SRC) &

```

hoc v1.2



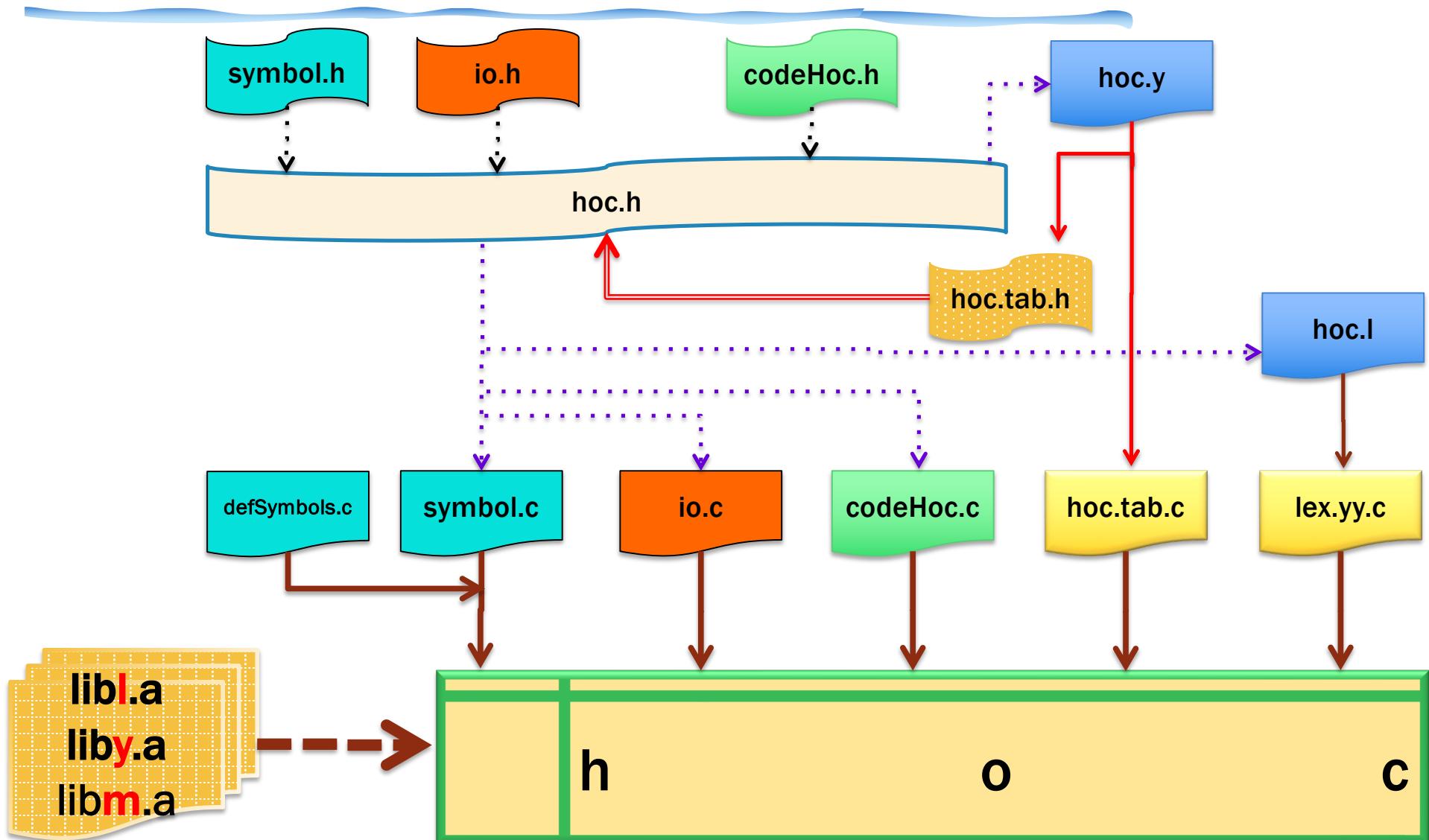
Hoc v1.2

Version	Module	Description
1.2		Gestion des erreurs avec affichage de messages formatés ,
	COLOR	Coloration et taggage des messages : définition de styles par défaut
	IO	Déclaration d'une table de messages formatés & un style prédéfini (<code>msgFmt.c</code>)
		Importation et adaptation de <code>prompt()</code> et <code>printExpr()</code>
		Importation et adaptation de <code>yyerror()</code>
		Ecriture de <code>lexError()</code> et <code>exeError()</code>
	MAKEFILE	Mise en place de makefile

Hoc – v1.3 & v1.4

Version	Module	Description
1.3	LEX	Les opérateurs deviennent des symboles : classe PRG et le type équivaut au token (ADD, SUB, MUL, DIV)
	CODE	Implémentation des fonctions de traitement des opérations algébriques
	YACC	Typage des tokens opérateurs algébriques Création de l'US opAlg de type symb englobant tout les opérateurs algébriques (casse les priorités). Appel des fonctions de traitement des opérations algébriques
	SYMBOL	Installation des opérations algébriques comme des symboles
1.4	IO/DBG	Traçabilité & débogage : normalisation des formats d'affichage

hoc v1.3/1.4

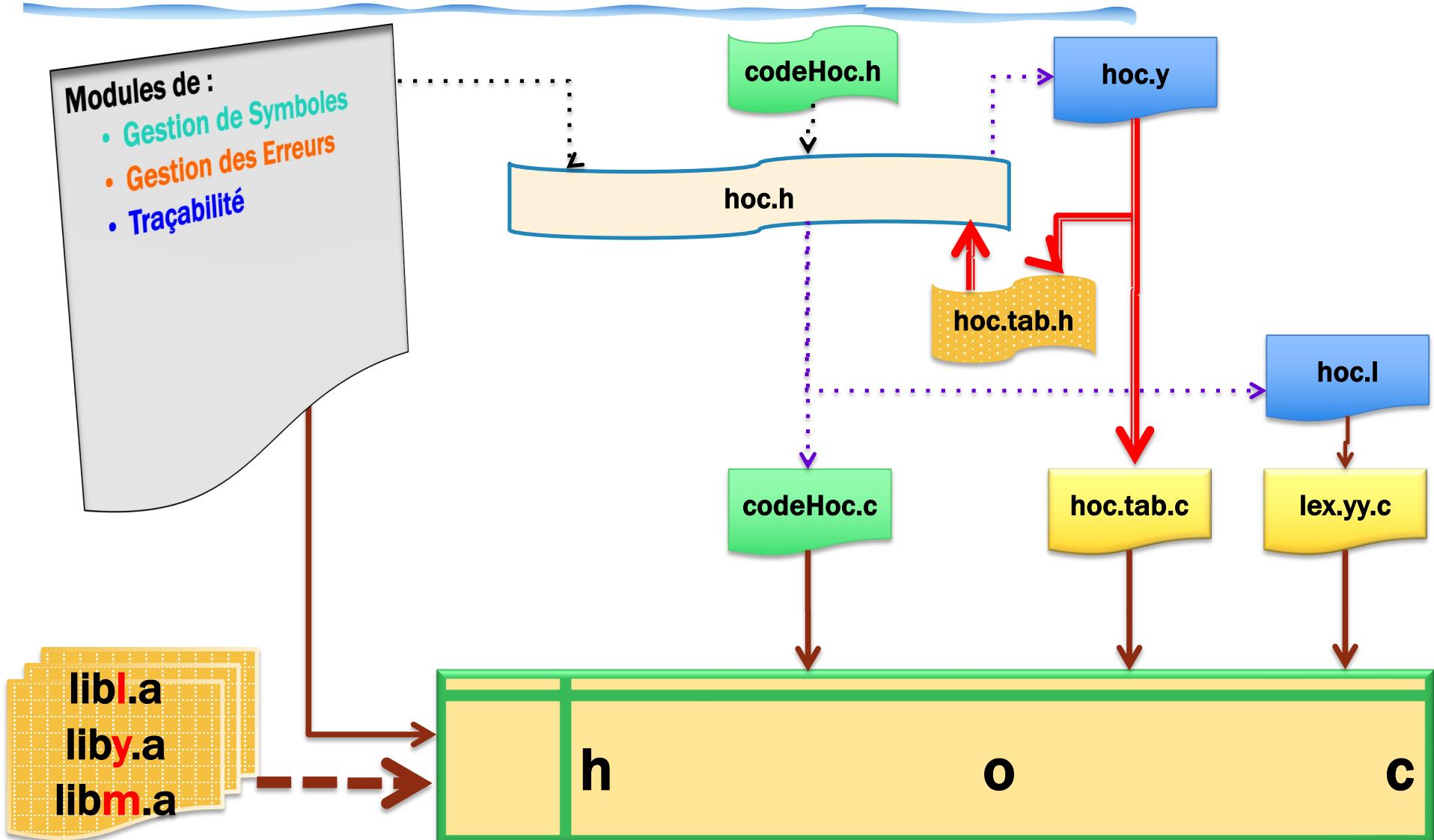


Hoc v1.5

Il faut apporter quelques **améliorations** à la version 1.4, pour obtenir la version 1.5 :

- Réécrire la fonction `printSymbolList()` pour répondre au token `PR_TS` :
 - ▶ Il faut utiliser des messages formatés avec coloration en éliminant les macros d'affichage,
 - ▶ Changer le résultat (affichage) en fournissant un résultat ressemblant à celui fourni en commentaire (voir code, suite fonction).
- Unifier le code AL/TS/AS : `installIntSymbol()`, `InstallDoubleSymbol()` avec `installDefaultSymbols`, traitement d'une affectation.
- Unifier/factoriser le traitement des variables & nombres par l'AS en introduisant une unité syntaxique.
- Générer une librairie pour la gestion de symboles et des IO.

Hoc v1.5



Génération de code intermédiaire & Interprétation

A chaque règle syntaxique reconnue, un code intermédiaire en **langage abstrait** sera généré. Le code généré est équivalent à la règle reconnue. Généralement, il s'agit de :

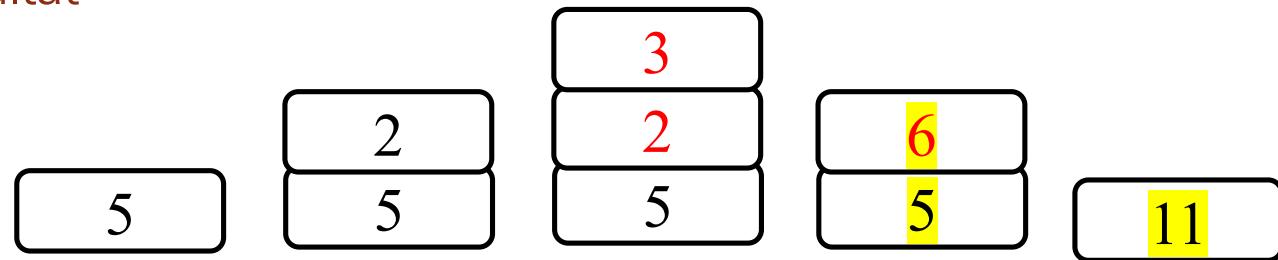
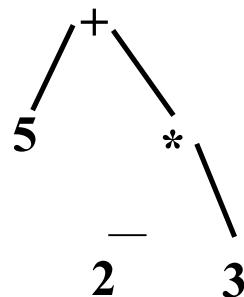
- stocker l'adresse d'un symbole,
- stocker l'adresse d'une fonction qui réalise le traitement attendu,

Le code généré sera placé dans une **table de code**.

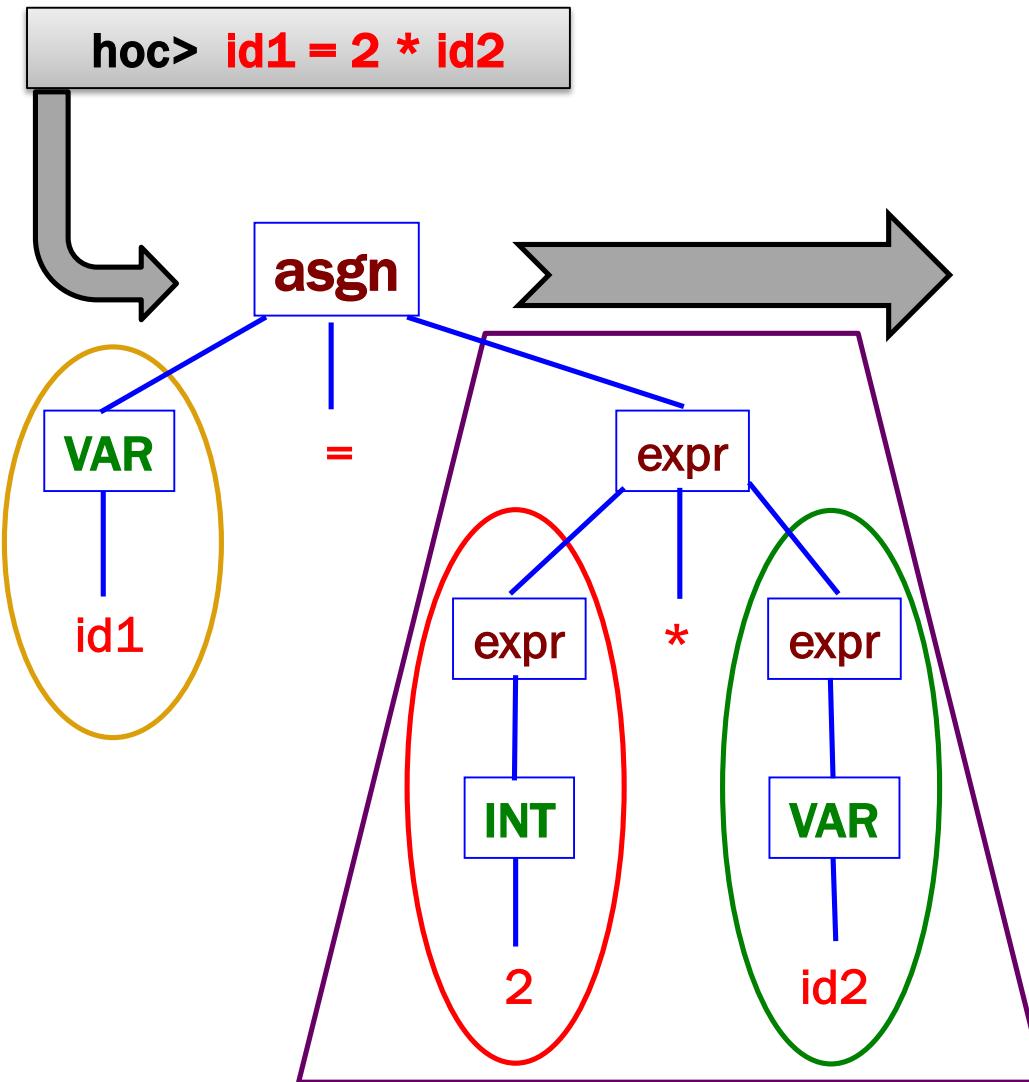
L'exécution du code intermédiaire utilisera une **pile de données**.

Rappel Evaluation par pile :

- Un nombre : on empile
- Un opérateur : on dépile deux valeurs (opérandes), on fait l'opération, on empile le résultat



Exemple – Génération de code intermédiaire



arrêter
affecter la variable id1
empiler variable id1
multiplier id2 par 2
évaluer la variable id2
empiler variable id2
empiler constante de valeur 2

Table de code générée

Hoc – Génération de code intermédiaire

Transformation de la version précédente :

- avant : évaluation immédiate
- après : génération de code intermédiaire en langage abstrait et exécution du code généré.

Génération de code :

- Pour les nombres, le code associé sera de ranger le pointeur sur le symbole :
 - ▶ `nbrPush()`
- Il en est de même pour les variables avec une opération d'évaluation ou d'assignation :
 - ▶ `varPush()`, `evalVar()`, `assignVar()`
- Pour les fonctions prédéfinies, on range l'adresse de la fonction (après avoir rangé l'adresse du symbole paramètre) :
 - ▶ `predef()`
- Pour les opérations, on considère que la pile de données contient les opérandes nécessaires pour effectuer l'opération. On dépile alors les opérandes, on effectue l'opération et on empile le résultat.
 - ▶ `add()`, `sub()`, `mul()`, `myDiv()`, `power()`, `negate()`,

Hoc v2.0 – Table de code intermédiaire (à générer)

La génération du code intermédiaire nécessite **une table** qui sera constituée d'**adresses** de symboles ou d'**adresses** de fonctions :

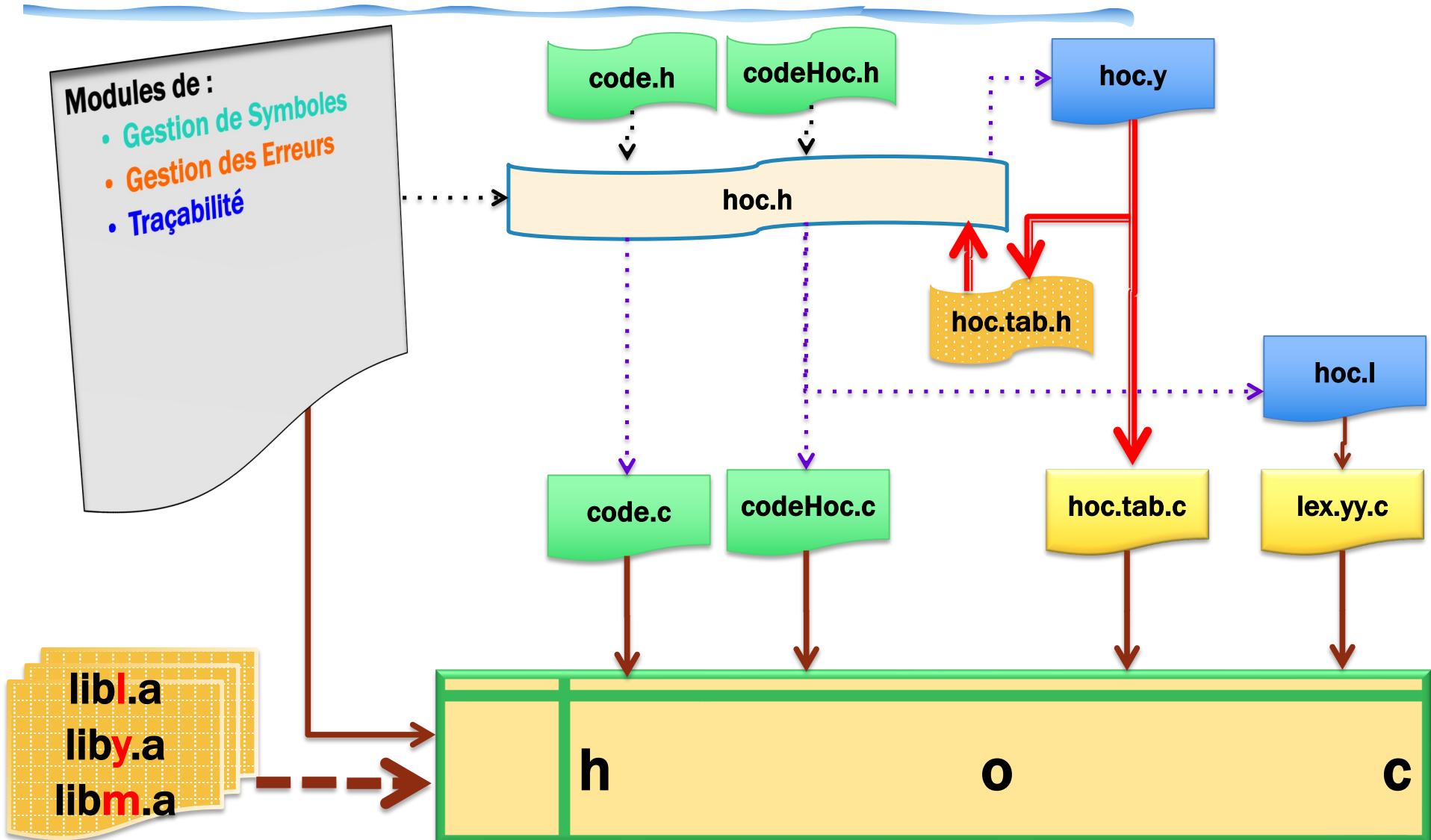
déclaration	Description
<code>typedef int (*instr_t)();</code>	Définition d'une Instruction machine
<code>#define STOP (instr_t) 0</code>	Instruction nulle
<code>#define MAX_PROG 2000</code>	Taille de la table
<code>instr_t prog [MAX_PROG];</code>	Table du code intermédiaire
<code>instr_t *progPtr;</code>	Pointeur sur la première case libre de la table : utilisé pendant la génération pour placer une instruction de code.
<code>instr_t *PC;</code>	Pointeur sur la prochaine instruction à exécuter : table : utilisé pendant l' interprétation pour repérer l'instruction à exécuter.
<code>instr_t *code (instr_t);</code>	Fonction qui range une instruction dans la table
<code>void execute (instr_t *);</code>	Fonction qui interprète un code rangé prog

Hoc v2.0 – Table de code intermédiaire (à générer)

L'interprétation du code généré nécessite **une pile de données** qui contiendra soit la valeur d'un résultat de calcul ou d'un pointeur sur un symbole :

déclaration	Description
<pre>typedef union { double value; symbol_t pSymb;} data_t;</pre>	Définition du type de données utilisées par la pile : 1. value : valeur obtenue suite à un traitement 2. pSymb : adresse du symbole donnant accès à la valeur
<pre>#define MAXPILE 256</pre>	Taille de la pile de données
<pre>data_t pile [MAXPILE];</pre>	Pile de données
<pre>data_t *pilePtr;</pre>	Pointeur sur la première case libre de la pile (sommet de la pile)
<pre>data_t *pop (void);</pre>	Fonction qui dépile une donnée
<pre>void push (data_t);</pre>	Fonction qui empile une donnée

Hoc v2.0



Hoc – v2.1 & v2.2

Améliorer la version 2.0, et obtenir la 2.1 en modifiant la gestion d'erreurs : lexique, syntaxe, sémantique, exécution.

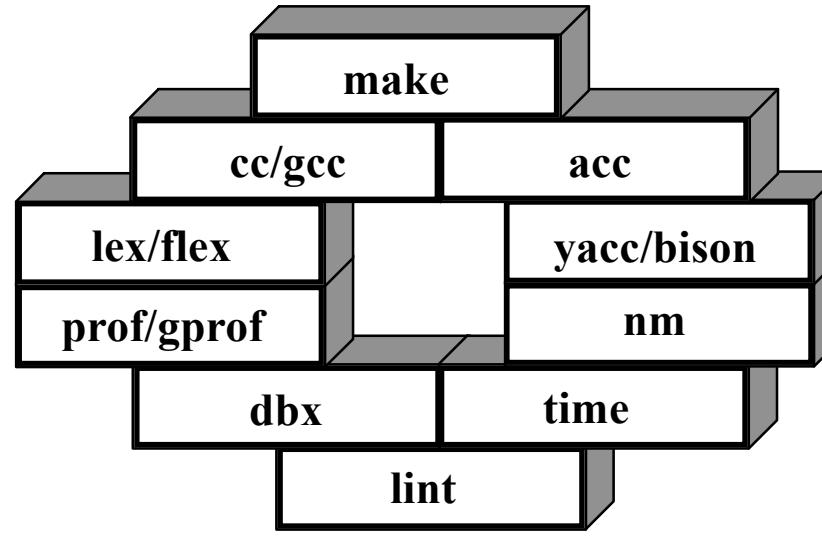
Il y aura donc une codification et une catégorisation de l'erreur.

Implémenter le déboggage pour chaque étape de la compilation comme options de compilation de votre langage.

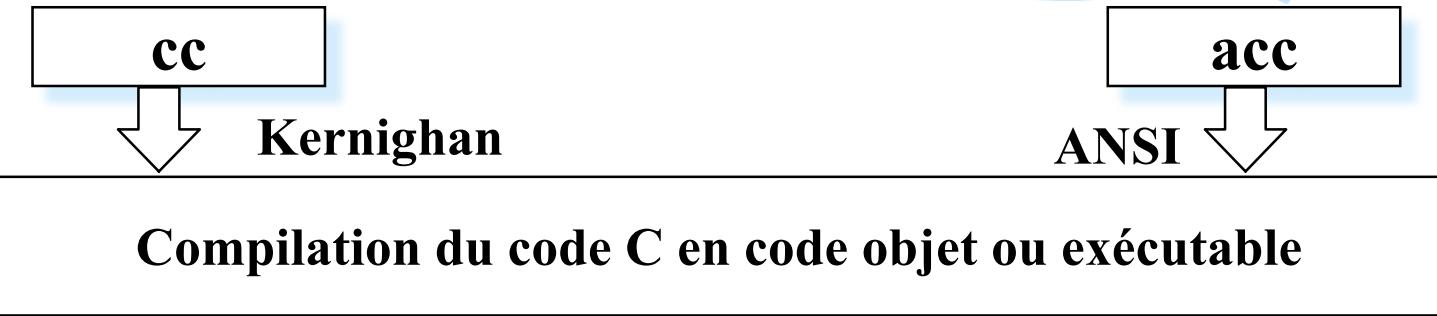
Version	Module	Description
2.1	?	Utilisation la variable myError comme code d'erreur. La détection d'erreur se fait aux différentes étapes, ce qui permet de positionner l'erreur et d'invoquer la bonne fonction d'affichage de l'erreur. semanticError() n'est pas définie !
2.2	?	Déboggage de l'AL, l'AS, la génération de code et de son exécution.

PARTIE 5

OUTILS DE COMPILEATION EN UNIX



Compilation des sources C



Options de compilation :

- g : préparation pour le deboggeur
- c : compilation sans édition de lien (code objet)
- o : compilation avec lien (code exécutable)
- O : optimisation du code en fonction de la machine
- p : ajoute les infos pour l'analyseur de performances
- time : indique le temps consommé par chaque étape de compilation

Exemple :

- | | |
|--------------------------|-------------------------------------|
| acc -c prog.c | génération du code objet prog.o |
| acc -o exec prog.o lib.o | génération du code exécutable exec |
| cc -O prog.c | génération du code exécutable a.out |

Analyse fine des sources C (1)



Raisons :

- Tout est permis en C (portabilité portée à son maximum).
- Peu de contrôles sont effectués.
- Problèmes de normalisation : caractères signés ou non, ordre d'évaluation des expressions

Inconvénient :

- trop bavard.

Options :

- -a : signaler l'affectation du type long à d'autres types
- -b : détecter les blocs du code non atteints après un break
- -c: vérifier la portabilité du code
- -h : utiliser l'heuristique dans la recherche des erreurs
- -x : lister les variables externes non utilisées

Analyse fine des sources C (2)

lint

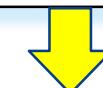
alint

Résultats :

- variables ou fonctions inutilisées
- variables et fonctions mal utilisées
- instructions inaccessibles
- plusieurs fins de fonction ou différents types de return
- vérification des types
 - ▶ avant les opérateurs -> et . on doit pointer sur une structure
 - ▶ disjonction des types enum
 - ▶ vérification des types d'arguments des fonctions
- conversion long dans un int
- constructions ou structures douteuses
- problèmes de priorité des expressions
- redéclaration de variables globales/locales

Déverminage (debug)

dbx



stop at n	: point d'arrêt à la ligne n
cont	: continuer l'exécution
print j	: afficher la valeur de la variable j
trace j	: afficher les valeurs successives de la variable j
stop in toto if i==0	: arrêt conditionnel
status	: afficher les points de contrôle
delete n°	: effacer un point de contrôle
delete all	: effacer tous les points de contrôle
clear	: effacer les points d'arrêt
run	: exécution
next	: exécution pas à pas
where	: ligne ayant provoquée une erreur d'exécution
quit	: quitter le débogueur
help	: afficher les commandes disponibles
help cde	: afficher l'aide sur la commande cde

Chronométrage du temps d'exécution

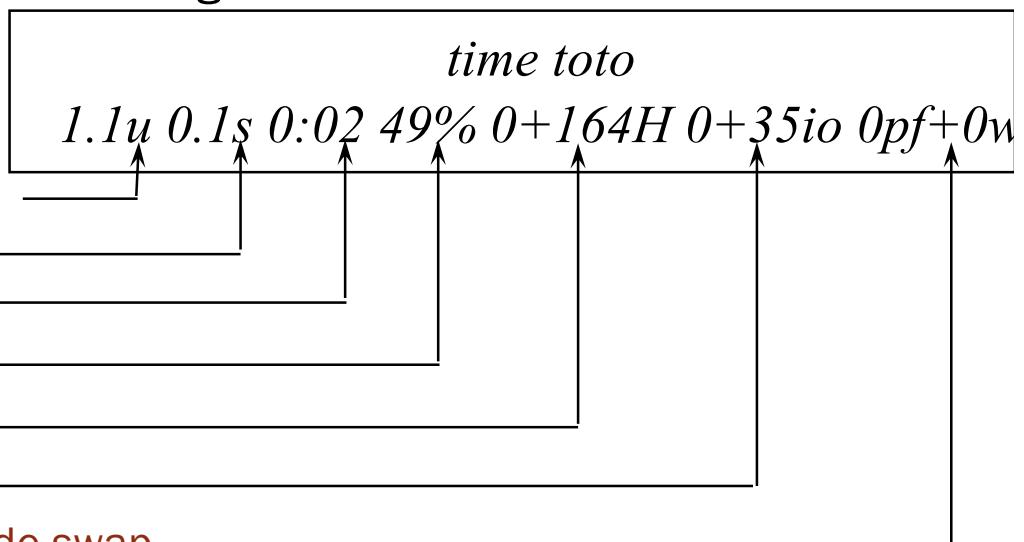
time

Mesure du temps d'exécution d'une commande

La commande time fournit des mesures sur le temps d'exécution en secondes.
Cette commande ne prend pas en compte la charge réelle de la machine.

Résultats :

- Temps d'exécution de la commande _____
- Temps passé dans le noyau _____
- Durée totale _____
- % de la durée totale _____
- Occupation de la mémoire _____
- Nombre d'E/S _____
- Nombre de page mémoire+Nombre de swap _____



Liste des symboles d'un objet

nm



Affiche la liste des symboles définis

dans un fichier objet avec leurs caractéristiques

Syntaxe :

- nm options file

Catégories :

- U : symbole de type indéfini
- D : symbole de type data.
- T : symbole de type texte (Noms de fonctions).
- C : variables générales.

Résultats :

- Pour chaque symbole, la commande renvoie sa valeur (blanc si indéfinie) et sa catégorie.

Analyse des performances (1)

prof/gprof



Fournit une trace détaillée du temps consommé par chaque procédure

Utilisation et Syntaxe :

- Le programme doit être compilé avec l'option -p (-pg pour gprof)
- Lancement du programme → Création d'un fichier trace mon.out (gmon.out pour gprof)
- Lancement de la commande prof/gprof suivie du nom du programme
 - ▶ Comparaison de la table des symboles et mon.out pour fournir les statistiques.

Résultats :

- %time : % du temps CPU passé dans la procédure.
- cumsecs : temps d'exécution de la procédure et ces sous-procédures.
- #call : nombre d'appels.
- ms/call : temps moyen par appel.
- name : nom de la procédure.

Analyse des performances (2)

prof/gprof

Précautions et Intérêts :

- Dépendance de la charge du système.
- Pas de prise en compte de ces résultats dans l'absolu.
- Comparaison de l'efficacité des procédures → localiser les procédures à optimiser.

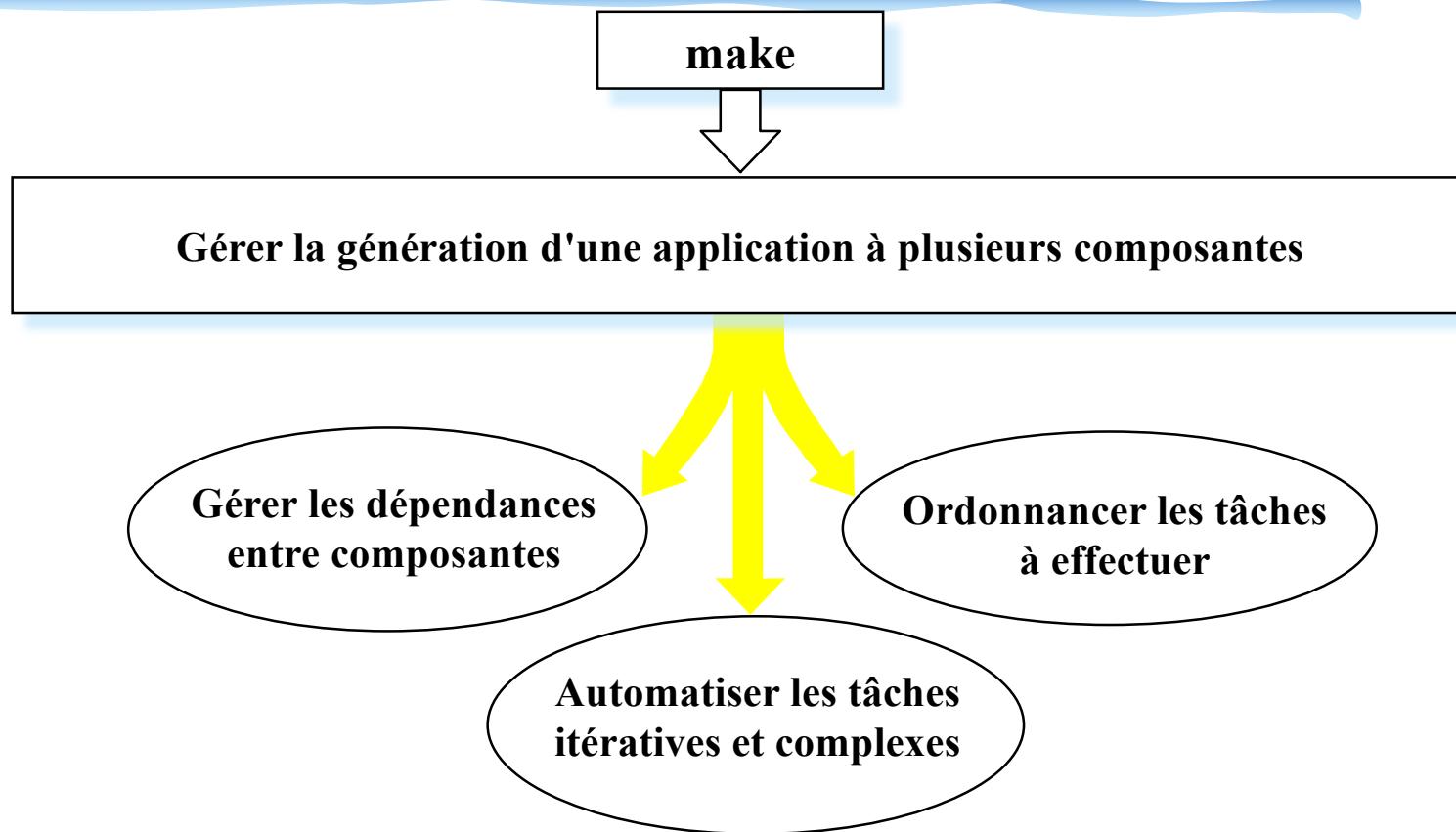
Technique :

- Plusieurs essais.
- Lancement à la même date pour avoir une charge identique (exécuter la nuit par exemple).
- Faire des moyennes.

Différence entre gprof et prof :

- Analyse plus détaillée.
- Graphe de dépendances des différentes procédures.
- Pour chaque procédure, la commande fournit :
 - ▶ la procédure appelante
 - ▶ les procédures appelées avec le temps passé dans chacune d'elles.

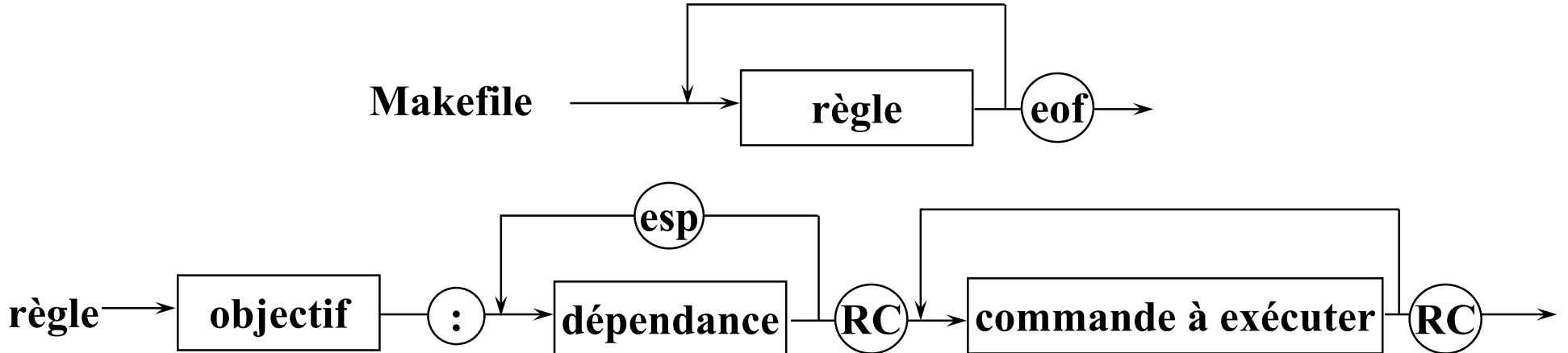
Utilitaire de compilation “make” (1)



make travaille selon un ensemble de règles définies dans un fichier utilisateur (en général Makefile ou makefile) et un autre ensemble de règles par défaut (/usr/include/make/default.mk)

- Par défaut un make nom-exe équivaut à cc -o nom-exe nom-exe.c

Utilitaire de compilation “make” (2)



Principe de Fonctionnement : Si la date de l'une des dépendances est postérieure à celle de l'objectif alors il y a exécution des commandes.

Si la liste des dépendances est vide, l'objectif est alors atteint.

Exemple :

toto : toto.c

cc -o toto toto.c

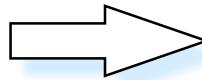
Si toto est à jour alors arrêt (toto is up to date) sinon lancement de la commande cc

Utilitaire de compilation “make” (3)

Ordonnancement des tâches :

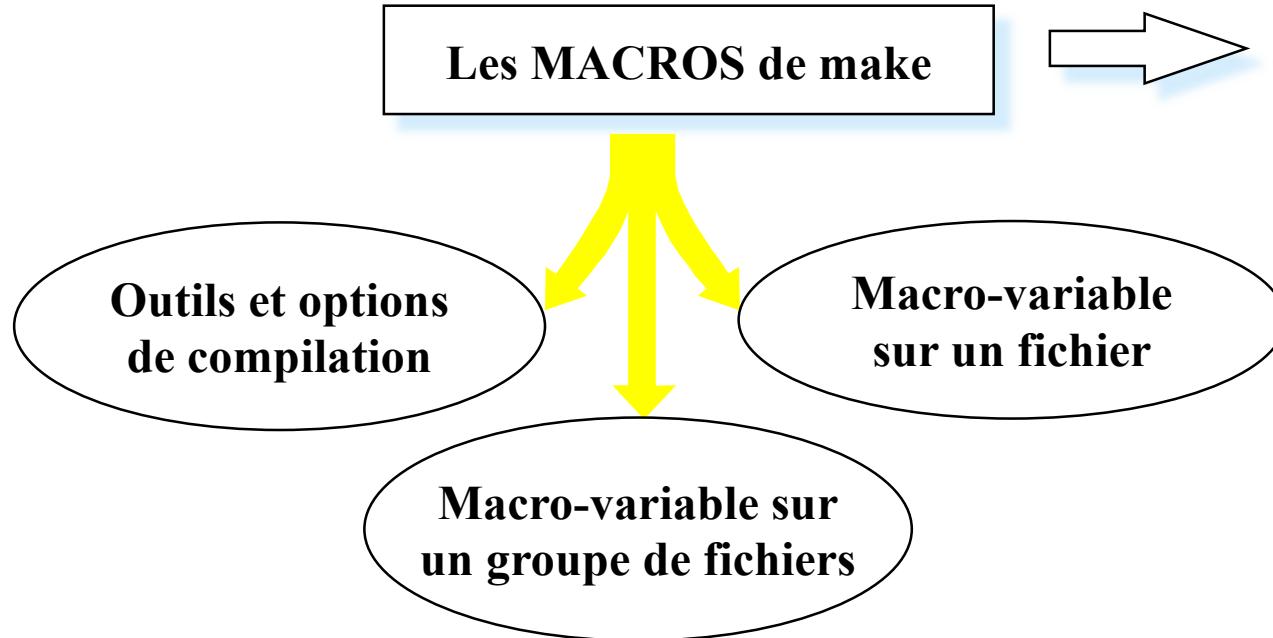
- make réalise un ordonnancement des tâches avant de les exécuter
- Test réalisé sur les fichiers (dépendances) :
 - ▶ test : est-ce que le fichier n'existe pas ou le fichier existe mais n'est pas à jour

```
f: f1 f2  
      touch f  
f1 : x y  
      touch f1  
f2 :  
      touch f2  
x :  
      touch x  
y :  
      touch y
```



```
Si test sur f alors  
  Si test sur f1 alors  
    Si test sur x alors  
      touch x (création de x)  
    Si test sur y alors  
      touch y (création de y)  
      touch f1 (création de f1)  
  Si test sur f2 alors  
    touch f2 (création de f2)  
touch f (création de f)
```

Utilitaire de compilation “make” (4)



```
FLAGS = -O  
CFLAGS = -O  
toto : x.o y.o z.o  
cc $(FLAGS) -o toto x.o y.o z.o  
titi : x.o y.o z.o  
cc -o titi x.o y.o z.o
```

Utilisation d'une Macro-Variable
en option de compilation : FLAGS

Utilisation d'une option de
compilation : CFLAGS

Utilitaire de compilation “make” (5)

MACROS PREDEFINIES :

- CC : Compilateur utilisé
- CFLAGS : Options de CC
- CPPFLAGS : Options du pré-processeur C
- LDFLAGS : Options de l'édition de liens (link)
- TARGET_ARCH : Architέcture cible (sun4c, sparc, vax, 386i, alpha , ...)

VALEURS PAR DEFAUT :

- CC = cc
- LINT = lint
- COMPILE.c = \$(CC) \$(CFLAGS) \$(CPPFLAGS) -c
- LINK.c = \$(CC) \$(CFLAGS) \$(CPPFLAGS) \$(LDFLAGS)

EXEMPLE :

- CFLAGS = -O
- toto : x.o def.h
- \$(LINK.c) -o toto x.o
- X.O:
- \$(COMPILE.c) x.c