Techniques et outils pour la compilation

Henri Garreta
Faculté des Sciences de Luminy - Université de la Méditerranée
Janvier 2001

Table des matières

1 Introduction

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. Fondamentalement, c'est une traduction : un texte écrit en Pascal, C, Java, etc., exprime un algorithme et il s'agit de produire un autre texte, spécifiant le même algorithme dans le langage d'une machine que nous cherchons à programmer.

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 ; à la limite, 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.

Bien sûr, les questions qui nous intéresseront ici seront plus complexes que la simple lecture d'un nombre. Mais il faut comprendre que le domaine d'application des principes et méthodes de l'écriture de compilateurs contient bien d'autres choses que la seule production de programmes exécutables. Chaque fois que vous aurez à écrire un programme lisant des expressions plus compliquées que des nombres vous pourrez tirer profit des concepts, techniques et outils expliqués dans ce cours.

1.1 Structure de principe d'un compilateur

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*[1]. Voici les phases dans lesquelles se décompose le travail d'un compilateur, du moins d'un point de vue logique[2] (voyez la figure 1) :

Analyse lexicale Dans cette phase, les caractères isolés qui constituent le texte source sont regroupés pour former des *unités lexicales*, qui sont les mots du langage.
L'analyse lexicale opère sous le contrôle de l'analyse syntaxique ; elle apparaît comme une sorte de fonction de « lecture améliorée », qui fournit un mot lors de chaque appel.

Analyse syntaxique Alors que l'analyse lexicale reconnaît les mots du langage, l'analyse syntaxique en reconnaît les phrases. Le rôle principal de cette phase est de dire si le texte source appartient au langage considéré, c'est-à-dire s'il est correct relativement à la grammaire de ce dernier.

Analyse sémantique La structure du texte source étant correcte, il s'agit ici de vérifier certaines propriétés sémantiques, c'est-à-dire relatives à la signification de la phrase et de ses constituants :
- les identificateurs apparaissant dans les expressions ont-ils été declarés ?
- les opérandes ont-ils les types requis par les opérateurs ?
- les opérandes sont-ils compatibles ? n'y a-t-il pas des conversions à insérer ?
- les arguments des appels de fonctions ont-ils le nombre et le type requis ?

etc.

Génération de code intermédiaire Après les phases d'analyse, certains compilateurs ne produisent pas directement le code attendu en sortie, mais une représentation intermédiaire, une sorte de code pour une machine abstraite. Cela permet de concevoir indépendamment les premières phases du compilateur (constituant ce que l'on appelle sa *face avant*) qui ne dépendent que du langage source compilé et les dernières phases (formant sa *face arrière*) qui ne dépendent que du langage cible ; l'idéal serait d'avoir plusieurs faces avant et plusieurs faces arrière qu'on pourrait assembler librement[3].

[1] Conseil : le texte source a probablement été composé à l'aide d'un éditeur de textes qui le montre sous forme de pages faites de plusieurs lignes mais, pour ce que nous avons à en faire ici, prenez l'habitude de l'imaginer comme s'il était écrit sur un long et mince ruban, formant une seule ligne.

[2] C'est une organisation logique ; en pratique certaines de ces phases peuvent être imbriquées, et d'autres absentes.

[3] De la sorte, avec n faces avant pour n langages source et m faces arrière correspondant à m machines cibles, on disposerait automatiquement de $n \times m$ compilateurs distincts. Mais cela reste, pour le moment, un fantasme d'informaticien.

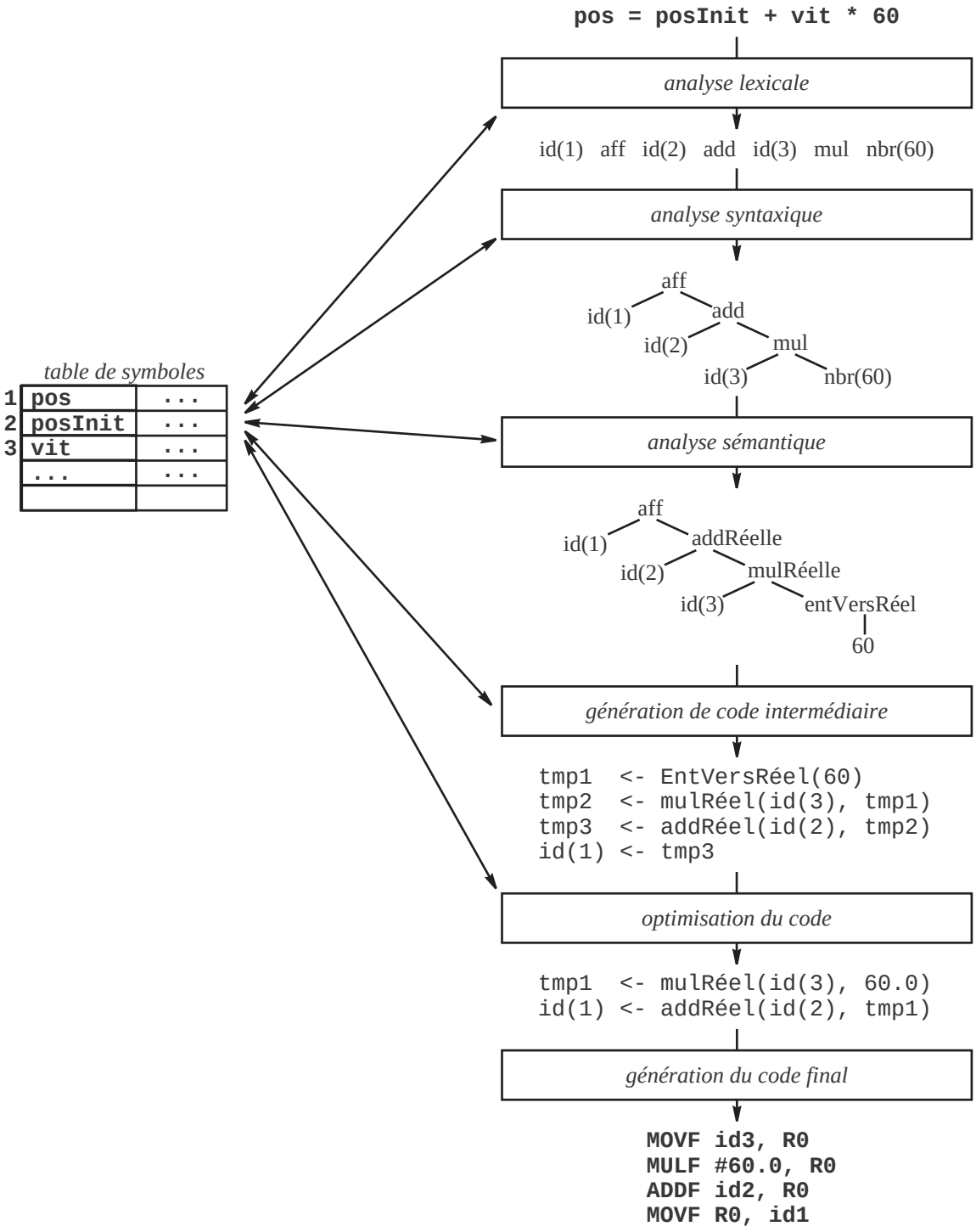


FIG. 1 – Phases logiques de la compilation d'une instruction

Optimisation du code Il s'agit généralement ici de transformer le code afin que le programme résultant s'exécute plus rapidement. Par exemple

- détecter l'inutilité de recalculer des expressions dont la valeur est déjà connue,
- transporter à l'extérieur des boucles des expressions et sous-expressions dont les opérandes ont la même valeur à toutes les itérations
- détecter, et supprimer, les expressions inutiles

etc.

Génération du code final Cette phase, la plus impressionnante pour le néophyte, n'est pas forcément la plus difficile à réaliser. Elle nécessite la connaissance de la machine cible (réelle, virtuelle ou abstraite), et notamment de ses possibilités en matière de registres, piles, etc.

2 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.

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.

A propos d'une unité lexicale reconnue dans le texte source on doit distinguer quatre notions importantes :
- l'*unité lexicale*, représentée généralement par un code conventionnel ; pour nos dix exemples `+`, `=`, `<=`, `++`, `if`, `while`, `123`, `-5`, `i` et `vitesse_du_vent`, ce pourrait être, respectivement[4] : `PLUS`, `EGAL`, `INFEGAL`, `PLUSPLUS`, `SI`, `TANTQUE`, `NOMBRE`, `NOMBRE`, `IDENTIF`, `IDENTIF`.
- le *lexème*, qui est la chaîne de caractères correspondante. Pour les dix exemples précédents, les lexèmes correspondants sont : `"+"`, `"="`, `"<="`, `"++"`, `"if"`, `"while"`, `"123"`, `"-5"`, `"i"` et `"vitesse_du_vent"`
- éventuellement, un *attribut*, qui dépend de l'unité lexicale en question, et qui la complète. Seules les dernières des dix unités précédentes ont un attribut ; pour un nombre, il s'agit de sa valeur (123, -5) ; pour un identificateur, il s'agit d'un renvoi à une table dans laquelle sont placés tous les identificateurs rencontrés (on verra cela plus loin).
- le *modèle* qui sert à spécifier l'unité lexicale. Nous verrons ci-après des moyens formels pour définir rigoureusement les modèles ; pour le moment nous nous contenterons de descriptions informelles comme :
  - pour les caractères spéciaux simples et doubles et les mots réservés, le lexème et le modèle coïncident,
  - le modèle d'un nombre est « une suite de chiffres, éventuellement précédée d'un signe »,
  - le modèle d'un identificateur est « une suite de lettres, de chiffres et du caractère '_', commençant par une lettre ».

Outre la reconnaissance des unités lexicales, les analyseurs lexicaux assurent certaines tâches mineures comme la suppression des caractères de décoration (blancs, tabulations, fins de ligne, etc.) et celle des commentaires (généralement considérés comme ayant la même valeur qu'un blanc), l'interface avec les fonctions de lecture de caractères, à travers lesquelles le texte source est acquis, la gestion des fichiers et l'affichage des erreurs, etc.

Remarque. La frontière entre l'analyse lexicale et l'analyse syntaxique n'est pas fixe. D'ailleurs, l'analyse lexicale n'est pas une obligation, on peut concevoir des compilateurs dans lesquels la syntaxe est définie à partir des caractères individuels. Mais les analyseurs syntaxiques qu'il faut alors écrire sont bien plus complexes que ceux qu'on obtient en utilisant des analyseurs lexicaux pour reconnaître les mots du langage.

Simplicité et efficacité sont les raisons d'être des analyseurs lexicaux. Comme nous allons le voir, les techniques pour reconnaître les unités lexicales sont bien plus simples et efficaces que les techniques pour vérifier la syntaxe.

2.1 Expressions régulières

2.1.1 Définitions

Les *expressions régulières* sont une importante notation pour spécifier formellement des modèles. Pour les définir correctement il nous faut faire l'effort d'apprendre un peu de vocabulaire nouveau :

Un *alphabet* est un ensemble de symboles. Exemples : $\{0, 1\}$, $\{A, C, G, T\}$, l'ensemble de toutes les lettres, l'ensemble des chiffres, le code ASCII, etc. On notera que les *caractères blancs* (c'est-à-dire les espaces, les

[4] Dans un analyseur lexical écrit en C, ces codes sont des pseudo-constantes introduites par des directives `#define`

tabulations et les marques de fin de ligne) ne font généralement pas partie des alphabets[5].

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`, `ACCAGTTGAAGTGGACCTTT`, `Bonjour`, `2001`. On note ε la chaîne vide, ne comportant aucun caractère.

Un *langage* sur un alphabet Σ est un ensemble de chaînes construites sur Σ. Exemples triviaux : $\emptyset$, le langage vide, $\{\varepsilon\}$, le langage réduit à l'unique chaîne vide. Des exemples plus intéressants (relatifs aux alphabets précédents) : l'ensemble des nombres en notation binaire, l'ensemble des chaînes ADN, l'ensemble des mots de la langue française, etc.

Si x et y sont deux chaînes, la *concaténation* de x et y, notée xy, est la chaîne obtenue en écrivant y immédiatement après x. Par exemple, la concaténation des chaînes `anti` et `moine` est la chaîne `antimoine`. Si x est une chaîne, on définit $x^0 = \varepsilon$ et, pour $n > 0$, $x^n = x^{n-1}x = xx^{n-1}$. On a donc $x^1 = x$, $x^2 = xx$, $x^3 = xxx$, etc.

Les *opérations sur les langages* suivantes nous serviront à définir les expressions régulières. Soient L et M deux langages, on définit :

dénomination	notation	définition
l'union de L et M	$L \cup M$	$\{ x \mid x \in L \text{ ou } x \in M\}$
la concaténation de L et M	LM	$\{ xy \mid x \in L \text{ et } y \in M\}$
la fermeture de Kleene de L	L^*	$\{ x_1x_2...x_n \mid x_i \in L, n \in \mathbb{N} \text{ et } n \geq 0\}$
la fermeture positive de L	L^+	$\{ x_1x_2...x_n \mid x_i \in L, n \in \mathbb{N} \text{ et } n > 0\}$

De la définition de LM on déduit celle de $L^n = LL \dots L$.

Exemples. On se donne les alphabets $L = \{A, B, \dots Z, a, b, \dots z\}$, ensemble des lettres, et $C = \{0, 1, \dots 9\}$, ensemble des chiffres. En considérant qu'un caractère est la même chose qu'une chaîne de longueur un, on peut voir L et C comme des langages, formés de chaînes de longueur un. Dans ces conditions :

- $L \cup C$ est l'ensemble des lettres et des chiffres,
- LC est l'ensemble des chaînes formées d'une lettre suivie d'un chiffre,
- L^4 est l'ensemble des chaînes de quatre lettres,
- L^* est l'ensemble des chaînes faites d'un nombre quelconque de lettres ; ε en fait partie,
- C^+ est l'ensemble des chaînes de chiffres comportant au moins un chiffre,
- $L(L \cup C)^*$ est l'ensemble des chaînes de lettres et chiffres commençant par une lettre.

Expression régulière. Soit Σ un alphabet. Une expression régulière r sur Σ est une formule qui définit un langage $L(r)$ sur Σ, de la manière suivante :

1. ε est une expression régulière qui définit le langage $\{\varepsilon\}$
2. Si $a \in \Sigma$, alors a est une expression régulière qui définit le langage[6] $\{a\}$
3. Soient x et y deux expressions régulières, définissant les langages $L(x)$ et $L(y)$. Alors
   - $(x)|(y)$ est une expression régulière définissant le langage $L(x) \cup L(y)$
   - $(x)(y)$ est une expression régulière définissant le langage $L(x)L(y)$
   - $(x)^*$ est une expression régulière définissant le langage $(L(x))^*$
   - (x) est une expression régulière définissant le langage $L(x)$

La dernière règle ci-dessus signifie qu'on peut encadrer une expression régulière par des parenthèses sans changer le langage défini. D'autre part, les parenthèses apparaissant dans les règles précédentes peuvent souvent être omises, en fonction des opérateurs en présence : il suffit de savoir que les opérateurs $*$, concaténation et $|$ sont associatifs à gauche, et vérifient

$$\textit{priorité}(*) > \textit{priorité}(\text{ concaténation }) > \textit{priorité}(|)$$

Ainsi, on peut écrire l'expression régulière `oui` au lieu de `(o)(u)(i)` et `oui|non` au lieu de `(oui)|(non)`, mais on ne doit pas écrire `oui`* au lieu de `(oui)`*.

[5] Il en découle que les unités lexicales, sauf mesures particulières (apostrophes, quillemets, etc.), ne peuvent pas contenir des caractères blancs. D'autre part, la plupart des langages autorisent les caractères blancs *entre* les unités lexicales.

[6] On prendra garde à l'abus de langage qu'on fait ici, en employant la même notation pour le caractère a, la chaîne a et l'expression régulière a. En principe, le contexte permet de savoir de quoi on parle.

Définitions régulières. Les expressions régulières se construisent à partir d'autres expressions régulières ; cela amène à des expressions passablement touffues. On les allège en introduisant des *définitions régulières* qui permettent de donner des noms à certaines expressions en vue de leur réutilisation. On écrit donc

$d_1 \to r_1$
$d_2 \to r_2$
$\dots$
$d_n \to r_n$

où chaque d_i est une chaîne sur un alphabet disjoint de Σ [7], distincte de $d_1, d_2, \dots d_{i-1}$, et chaque r_i une expression régulière sur $\Sigma \cup \{d_1, d_2, \dots d_{i-1}\}$.

Exemple. Voici quelques définitions régulières, et notamment celles de *identificateur* et *nombre*, qui définissent les identificateurs et les nombres du langage Pascal :

lettre → A | B | ... | Z | a | b | ... | z
chiffre → 0 | 1 | ... | 9
identificateur → *lettre* (*lettre* | *chiffre*)*

chiffres → *chiffre chiffre**
fraction-opt → . *chiffres* | ε
exposant-opt → (E (+ | - | ε) *chiffres*) | ε
nombre → *chiffres fraction-opt exposant-opt*

Notations abrégées. Pour alléger certaines écritures, on complète la définition des expressions régulières en ajoutant les notations suivantes :

- soit x une expression régulière, définissant le langage $L(x)$; alors $(x)^+$ est une expression régulière, qui définit le langage $(L(x))^+$,
- soit x une expression régulière, définissant le langage $L(x)$; alors $(x)?$ est une expression régulière, qui définit le langage $L(x) \cup \{ \varepsilon \}$,
- si $c_1, c_2, \dots c_k$ sont des caractères, l'expressions régulière $c_1|c_2|\dots|c_k$ peut se noter `[`$c_1c_2\dots c_k$`]`,
- à l'intérieur d'une paire de crochets comme ci-dessus, l'expression c_1–c_2 désigne la séquence de tous les caractères c tels que $c_1 \leq c \leq c_2$.

Les définitions de *lettre* et *chiffre* données ci-dessus peuvent donc se réécrire :

lettre → `[A–Za–z]`
chiffre → `[0–9]`

2.1.2 Ce que les expressions régulières ne savent pas faire

Les expressions régulières sont un outil puissant et pratique pour définir les unités lexicales, c'est-à-dire les constituants élémentaires des programmes. Mais elles se prêtent beaucoup moins bien à la spécification de constructions de niveau plus élevé, car elles deviennent rapidement d'une trop grande complexité.

De plus, on démontre qu'il y a des chaînes qu'on ne peut pas décrire par des expressions régulières. Par exemple, le langage suivant (supposé infini)

$$\{ \texttt{a}, \texttt{(a)}, \texttt{((a))}, \texttt{(((a)))}, \dots \}$$

ne peut pas être défini par des expressions régulières, car ces dernières ne permettent pas d'assurer qu'il y a dans une expression de la forme `((` ... `((a))` ... `))` autant de parenthèses ouvrantes que de parenthèses fermantes. On dit que les expressions régulières « ne savent pas compter ».

Pour spécifier ces structures équilibrées, si importantes dans les langages de programmation (penser aux parenthèses dans les expressions arithmétiques, les crochets dans les tableaux, *begin...end*, {...}, *if...then...*, etc.) nous ferons appel aux grammaires non contextuelles, expliquées à la section 3.1.

[7] On assure souvent la séparation entre Σ^* et les noms des définitions régulières par des conventions typographiques.

2.2 Reconnaissance des unités lexicales

Nous avons vu comment spécifier les unités lexicales ; notre problème maintenant est d'écrire un programme qui les reconnaît dans le texte source. Un tel programme s'appelle un *analyseur lexical*.

Dans un compilateur, le principal client de l'analyseur lexical est l'analyseur syntaxique. L'interface entre ces deux analyseurs est une fonction *int uniteSuivante(void)*[8], qui renvoie à chaque appel l'unité lexicale suivante trouvée dans le texte source.

Cela suppose que l'analyseur lexical et l'analyseur syntaxique partagent les définitions des constantes conventionnelles définissant les unités lexicales. Si on programme en C, cela veut dire que dans les fichiers sources des deux analyseurs on a inclus un fichier d'entête (fichier « .h ») comportant une série de définitions comme[9] :

```
#define IDENTIF  1
#define NOMBRE   2
#define SI       3
#define ALORS    4
#define SINON    5
```

etc.

Cela suppose aussi que l'analyseur lexical et l'analyseur syntaxique partagent également une variable globale contenant le lexème correspondant à la dernière unité lexicale reconnue, ainsi qu'une variable globale contenant le (ou les) attribut(s) de l'unité lexicale courante, lorsque cela est pertinent, et notamment lorsque l'unité lexicale est `NOMBRE` ou `IDENTIF`.

On se donnera, du moins dans le cadre de ce cours, quelques « règles du jeu » supplémentaires :
- l'analyseur lexical est « glouton » : chaque lexème est le plus long possible[10] ;
- seul l'analyseur lexical accède au texte source. L'analyseur syntaxique n'acquiert ses données d'entrée autrement qu'à travers la fonction *uniteSuivante* ;
- l'analyseur lexical acquiert le texte source un *caractère à la fois*. Cela est un choix que nous faisons ici ; d'autres choix auraient été possibles, mais nous verrons que les langages qui nous intéressent permettent de travailler de cette manière.

2.2.1 Diagrammes de transition

Pour illustrer cette section nous allons nous donner comme exemple le problème de la reconnaissance des unités lexicales `INFEG`, `DIFF`, `INF`, `EGAL`, `SUPEG`, `SUP`, `IDENTIF`, respectivement définies par les expressions régulières `<=`, `<>`, `<`, `=`, `>=`, `>` et $lettre(lettre|chiffre)^*$, *lettre* et *chiffre* ayant leurs définitions déjà vues.

Les *diagrammes de transition* sont une étape préparatoire pour la réalisation d'un analyseur lexical. Au fur et à mesure qu'il reconnaît une unité lexicale, l'analyseur lexical passe par divers *états*. Ces états sont numérotés et représentés dans le diagramme par des cercles.

De chaque état e sont issues une ou plusieurs flèches étiquetées par des caractères. Une flèche étiquetée par c relie e à l'état e_1 dans lequel l'analyseur passera si, alors qu'il se trouve dans l'état e, le caractère c est lu dans le texte source.

Un état particulier représente l'état initial de l'analyseur ; on le signale en en faisant l'extrémité d'une flèche étiquetée *debut*.

Des doubles cercles identifient les états finaux, correspondant à la reconnaissance complète d'une unité lexicale. Certains états finaux sont marqués d'une étoile : cela signifie que la reconnaissance s'est faite au prix de la lecture d'un caractère au-delà de la fin du lexème[11].

Par exemple, la figure 2 montre les diagrammes traduisant la reconnaissance des unités lexicales `INFEG`, `DIFF`, `INF`, `EGAL`, `SUPEG`, `SUP` et `IDENTIF`.

Un diagramme de transition est dit *non déterministe* lorsqu'il existe, issues d'un même état, plusieurs flèches étiquetées par le même caractère, ou bien lorsqu'il existe des flèches étiquetées par la chaîne vide ε. Dans le cas

[8] Si on a employé l'outil *lex* pour fabriquer l'analyser lexical, cette fonction s'appelle plutôt *yylex* ; le lexème est alors pointé par la variable globale *yytext* et sa longueur est donnée par la variable globale *yylen*. Tout cela est expliqué à la section 2.3.

[9] Peu importent les valeurs numériques utilisées, ce qui compte est qu'elles soient distinctes.

[10] Cette règle est peu mentionnée dans la littérature, pourtant elle est fondamentale. C'est grâce à elle que 123 est reconnu comme *un* nombre, et non plusieurs, $vitesseVent$ comme *un seul* identificateur, et $force$ comme un identificateur, et non pas comme un mot réservé suivi d'un identificateur.

[11] Il faut être attentif à ce caractère, car il est susceptible de faire partie de l'unité lexicale suivante, surtout s'il n'est pas blanc.

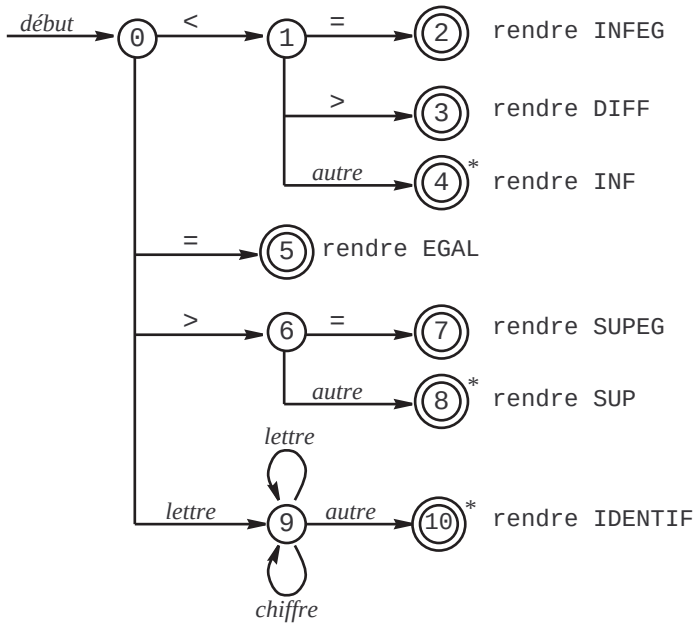


FIG. 2 – Diagramme de transition pour les opérateurs de comparaison et les identificateurs

contraire, le diagramme est dit déterministe. Il est clair que le diagramme de la figure 2 est déterministe. *Seuls les diagrammes déterministes nous intéresseront dans le cadre de ce cours.*

2.2.2 Analyseurs lexicaux programmés « en dur »

Les diagrammes de transition sont une aide importante pour l'écriture d'analyseurs lexicaux. Par exemple, à partir du diagramme de la figure 2 on peut obtenir rapidement un analyseur lexical reconnaissant les unités `INFEG`, `DIFF`, `INF`, `EGAL`, `SUPEG`, `SUP` et `IDENTIF`.

Auparavant, nous apportons une légère modification à nos diagrammes de transition, afin de permettre que les unités lexicales soient séparées par un ou plusieurs blancs[12]. La figure 3 montre le (début du) diagramme modifié[13].

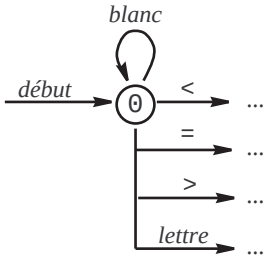


FIG. 3 – Ignorer les blancs devant une unité lexicale

Et voici le programme obtenu :

```
int uniteSuivante(void) {
    char c;
    c = lireCar();                              /* etat = 0 */
```

[12]Nous appelons « blanc » une espace, un caractère de tabulation ou une marque de fin de ligne

[13]Notez que cela revient à modifier toutes les expressions régulières, en remplaçant « `<=` » par « $(blanc)^*$`<=` », « `<` » par « $(blanc)^*$`<` », etc.

```
    while (estBlanc(c))
        c = lireCar();
    if (c == '<') {
        c = lireCar();                          /* etat = 1 */
        if (c == '=')
            return INFEG;                       /* etat = 2 */
        else if (c == '>')
            return DIFF;                        /* etat = 3 */
        else {
            delireCar(c);                       /* etat = 4 */
            return INF;
        }
    }
    else if (c == '=')
        return EGAL;                            /* etat = 5 */
    else if (c == '>') {
        c = lireCar();                          /* etat = 6 */
        if (c == '=')
            return SUPEG;                       /* etat = 7 */
        else {
            delireCar(c);                       /* etat = 8 */
            return SUP;
        }
    }
    else if (estLettre(c)) {
        lonLex = 0;                             /* etat = 9 */
        lexeme[lonLex++] = c;
        c = lireCar();
        while (estLettre(c) || estChiffre(c)) {
            lexeme[lonLex++] = c;
            c = lireCar();
        }
        delireCar(c);                           /* etat = 10 */
        return IDENTIF;
    }
    else {
        delireCar(c);
        return NEANT; /* ou bien donner une erreur */
    }
}
```

Dans le programme précédent on utilise des fonctions auxiliaires, dont voici une version simple :

```
int estBlanc(char c) {
    return c == ' ' || c == '\t' || c == '\n';
}
int estLettre(char c) {
    return 'A' <= c && c <= 'Z' || 'a' <= c && c <= 'z';
}
int estChiffre(char c) {
    return '0' <= c && c <= '9';
}
```

NOTE. On peut augmenter l'efficacité de ces fonctions, au détriment de leur sécurité d'utilisation, en en faisant des macros :

```
#define estBlanc(c) ((c) == ' ' || (c) == '\t' || (c) == '\n')
#define estLettre(c) ('A' <= (c) && (c) <= 'Z' || 'a' <= (c) && (c) <= 'z')
#define estChiffre(c) ('0' <= (c) && (c) <= '9')
```

Il y plusieurs manières de prendre en charge la restitution d'un caractère lu en trop (notre fonction *delireCar*). Si on dispose de la bibliothèque standard C on peut utiliser la fonction *ungetc* :

```
void delireCar(char c) {
    ungetc(c, stdin);
}
char lireCar(void) {
    return getc(stdin);
}
```

Une autre manière de faire permet de se passer de la fonction *ungetc*. Pour cela, on gère une variable globale contenant, quand il y a lieu, le caractère lu en trop (il n'y a jamais plus d'*un* caractère lu en trop). Déclaration :

```
int carEnAvance = -1;
```

avec cela nos deux fonctions deviennent

```
void delireCar(char c) {
    carEnAvance = c;
}
char lireCar(void) {
    char r;
    if (carEnAvance >= 0) {
        r = carEnAvance;
        carEnAvance = -1;
    }
    else
        r = getc(stdin);
    return r;
}
```

Reconnaissance des mots réservés. Les mots réservés appartiennent au langage défini par l'expression régulière $lettre(lettre|chiffre)^*$, tout comme les identificateurs. Leur reconnaissance peut donc se traiter de deux manières :

- soit on incorpore les mots réservés au diagrammes de transition, ce qui permet d'obtenir un analyseur très efficace, mais au prix d'un travail de programmation plus important, car les diagrammes de transition deviennent très volumineux[14],
- soit on laisse l'analyseur traiter de la même manière les mots réservés et les identificateurs puis, quand la reconnaissance d'un « identificateur-ou-mot-réservé » est terminée, on recherche le lexème dans une table pour déterminer s'il s'agit d'un identificateur ou d'un mot réservé.

Dans les analyseurs lexicaux « en dur » on utilise souvent la deuxième méthode, plus facile à programmer. On se donne donc une table de mots réservés :

```
struct {
    char *lexeme;
    int uniteLexicale;
} motRes[] = {
        { "si", SI },
        { "alors", ALORS },
        { "sinon", SINON },
        ...
    };
int nbMotRes = sizeof motRes / sizeof motRes[0];
```

puis on modifie de la manière suivante la partie concernant les identificateurs de la fonction *uniteSuivante* :

```
    ...
    else if (estLettre(c)) {
        lonLex = 0;                          /* etat =  9 */
        lexeme[lonLex++] = c;
```

[14] Nous utiliserons cette solution quand nous étudierons l'outil *lex*, à la section 2.3

```
    c = lireCar();
    while (estLettre(c) || estChiffre(c)) {
        lexeme[lonLex++] = c;
        c = lireCar();
    }
    delireCar(c);                          /* etat = 10 */

    lexeme[lonLex] = '\0';
    for (i = 0; i < nbMotRes; i++)
        if (strcmp(lexeme, motRes[i].lexeme) == 0)
            return motRes[i].uniteLexicale;

    return IDENTIF;
}
...
```

2.2.3 Automates finis

Un automate fini est défini par la donnée de
- un ensemble fini d'états E,
- un ensemble fini de symboles (ou *alphabet*) d'entrée Σ,
- une fonction de transition, $transit : E \times \Sigma \to E$,
- un état ε_0 distingué, appelé *état initial*,
- un ensemble d'états F, appelés *états d'acceptation* ou *états finaux*.

Un automate peut être représenté graphiquement par un graphe où les états sont figurés par des cercles (les états finaux par des cercles doubles) et la fonction de transition par des flèches étiquetées par des caractères : si $transit(e_1, c) = e_2$ alors le graphe a une flèche étiquetée par le caractère c, issue de e_1 et aboutissant à e_2.

Un tel graphe est exactement ce que nous avons appelé *diagramme de transition* à la section 2.2.1 (voir la figure 2). Si on en reparle ici c'est qu'on peut en déduire un autre style d'analyseur lexical, assez différent de ce que nous avons appelé analyseur programmé « en dur ».

On dit qu'un automate fini *accepte* une chaîne d'entrée $s = c_1 c_2 \dots c_k$ si et seulement si il existe dans le graphe de transition un chemin joignant l'état initial e_0 à un certain état final e_k, composé de k flèches étiquetées par les caractères $c_1, c_2, \dots c_k$.

Pour transformer un automate fini en un analyseur lexical il suffira donc d'associer une unité lexicale à chaque état final et de faire en sorte que l'acceptation d'une chaîne produise comme résultat l'unité lexicale associée à l'état final en question.

Autrement dit, pour programmer un analyseur il suffira maintenant d'implémenter la fonction *transit* ce qui, puisqu'elle est définie sur des ensembles finis, pourra se faire par une table à double entrée. Pour les diagrammes des figures 2 et 3 cela donne la table suivante (les états finaux sont indiqués en gras, leurs lignes ont été supprimées) :

	' '	'\t'	'\n'	'<'	'='	'>'	*lettre*	*chiffre*	*autre*
0	0	0	0	1	**5**	6	9	*erreur*	*erreur*
1	**4***	**4***	**4***	**4***	**2**	**3**	**4***	**4***	**4***
6	**8***	**8***	**8***	**8***	**7**	**8***	**8***	**8***	**8***
9	**10***	**10***	**10***	**10***	**10***	**10***	9	9	**10***

On obtiendra un analyseur peut-être plus encombrant que dans la première manière, mais certainement plus rapide puisque l'essentiel du travail de l'analyseur se réduira à répéter « bêtement » l'action `etat = transit[etat][lireCar(` jusqu'à tomber sur un état final. Voici ce programme :

```
#define NBR_ETATS ...
#define NBR_CARS  256

int transit[NBR_ETATS][NBR_CARS];
```

```
int final[NBR_ETATS + 1];

int uniteSuivante(void) {
    char caractere;
    int etat = etatInitial;

    while ( ! final[etat]) {
        caractere = lireCar();
        etat = transit[etat][caractere];
    }
    if (final[etat] < 0)
        delireCar(caractere);
    return abs(final[etat]) - 1;
}
```

Notre tableau `final`, indexé par les états, est défini par
- `final`[e] $= 0$ si e n'est pas un état final (vu comme un booléen, `final`[e] est faux),
- `final`[e] $= U + 1$ si e est final, sans étoile et associé à l'unité lexicale U (en tant que booléen, `final`[e] est vrai, car les unités lexicales sont numérotées au moins à partir de zéro),
- `final`[e] $= -(U + 1)$ si e est final, étoilé et associé à l'unité lexicale U (en tant que booléen, `final`[e] est encore vrai).

Enfin, voici comment les tableaux `transit` et `final` devraient être initialisés pour correspondre aux diagrammes des figures 2 et 3[15] :

```
void initialiser(void) {
    int i, j;

    for (i = 0; i < NBR_ETATS; i++) final[i] = 0;
    final[ 2] = INFEG + 1;
    final[ 3] = DIFF + 1;
    final[ 4] = - (INF + 1);
    final[ 5] = EGAL + 1;
    final[ 7] = SUPEG + 1;
    final[ 8] = - (SUP + 1);
    final[10] = - (IDENTIF + 1);
    final[NBR_ETATS] = ERREUR + 1;

    for (i = 0; i < NBR_ETATS; i++)
        for (j = 0; j < NBR_CARS; j++)
            transit[i][j] = NBR_ETATS;

    transit[0][' '] = 0;
    transit[0]['\t'] = 0;
    transit[0]['\n'] = 0;

    transit[0]['<'] = 1;
    transit[0]['='] = 5;
    transit[0]['>'] = 6;

    for (j = 'A'; j <= 'Z'; j++) transit[0][j] = 9;
    for (j = 'a'; j <= 'z'; j++) transit[0][j] = 9;

    for (j = 0; j < NBR_CARS; j++) transit[1][j] = 4;
    transit[1]['='] = 2;
    transit[1]['>'] = 3;
```

[15] Nous avons ajouté un état supplémentaire, ayant le numéro `NBR_ETATS`, qui correspond à la mise en erreur de l'analyseur lexical, et une unité lexicale `ERREUR` pour signaler cela

```
    for (j = 0; j < NBR_CARS; j++) transit[6][j] = 8;
    transit[6]['='] = 7;

    for (j = 0; j < NBR_CARS; j++) transit[9][j] = 10;
    for (j = 'A'; j <= 'Z'; j++) transit[9][j] = 9;
    for (j = 'a'; j <= 'z'; j++) transit[9][j] = 9;
    for (j = '0'; j <= '9'; j++) transit[9][j] = 9;
}
```

2.3 *Lex*, un générateur d'analyseurs lexicaux

Les analyseurs lexicaux basés sur des tables de transitions sont les plus efficaces... une fois la table de transition construite. Or, la construction de cette table est une opération longue et délicate.

Le programme *lex*[16] fait cette construction automatiquement : il prend en entrée un ensemble d'expressions régulières et produit en sortie le texte source d'un programme C qui, une fois compilé, est l'analyseur lexical correspondant au langage défini par les expressions régulières en question.

Plus précisément (voyez la figure 4), *lex* produit un fichier source C, nommé `lex.yy.c`, contenant la définition de la fonction *int yytext(void)*, qui est l'exacte homologue de notre fonction *uniteSuivante* de la section 2.2.2 : un programme appelle cette fonction et elle renvoie une unité lexicale reconnue dans le texte source.

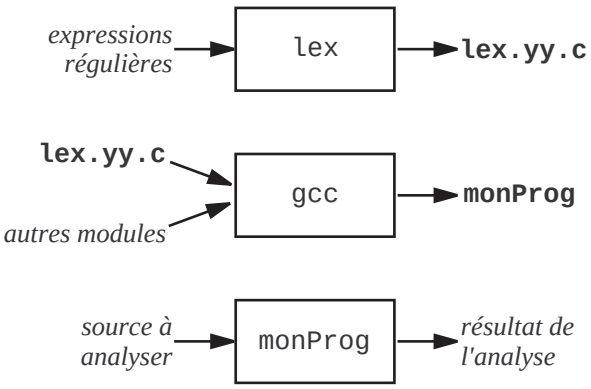


FIG. 4 – Utilisation de *lex*

2.3.1 Structure d'un fichier source pour *lex*

En lisant cette section, souvenez-vous de ceci : *lex* écrit un fichier source C. Ce fichier est fait de trois sortes d'ingrédients :

- des tables garnies de valeurs calculées à partir des expressions régulières fournies,
- des morceaux de code C invariable, et notamment le « moteur » de l'automate, c'est-à-dire la boucle qui répète inlassablement $etat \leftarrow transit\,(etat, caractere)$,
- des morceaux de code C, trouvés dans le fichier source *lex* et recopiés tels quels, à l'endroit voulu, dans le fichier produit.

Un fichier source pour *lex* doit avoir un nom qui se termine par « `.l` ». Il est fait de trois sections, délimitées par deux lignes réduites[17] au symbole `%%` :

```
%{
    déclarations pour le compilateur C
%}
    définitions régulières
```

[16] *Lex* est un programme gratuit qu'on trouve dans le système UNIX pratiquement depuis ses débuts. De nos jours on utilise souvent *flex*, une version améliorée de *lex* qui appartient à la famille GNU.

[17] Notez que les symboles `%%`, `%{` et `%}`, quand ils apparaissent, sont écrits au début de la ligne, aucun blanc ne les précède.

%%
règles
%%
fonctions C supplémentaires

La partie « *déclarations pour le compilateur C* » et les symboles `%{` et `%}` qui l'encadrent peuvent être omis. Quand elle est présente, cette partie se compose de déclarations qui seront simplement recopiées au début du fichier produit. En plus d'autres choses, on trouve souvent ici une directive `#include` qui produit l'inclusion du fichier « `.h` » contenant les définitions des codes conventionnels des unités lexicales (`INFEG`, `INF`, `EGAL`, etc.).

La troisième section « *fonctions C supplémentaires* » peut être absente également (le symbole `%%` qui la sépare de la deuxième section peut alors être omis). Cette section se compose de fonctions C qui seront simplement recopiées à la fin du fichier produit.

Définitions régulières. Les définitions régulières sont de la forme

identificateur *expressionRégulière*

où *identificateur* est écrit au début de la ligne (pas de blancs avant) et séparé de *expressionRégulière* par des blancs. Exemples :

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

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

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

%%

{lettre}{alphanum}*         { return IDENTIF; }
{chiffre}+("."{chiffre}+)?  { return NOMBRE; }
```

Règles. Les règles sont de la forme

expressionRégulière { *action* }

où *expressionRégulière* est écrit au début de la ligne (pas de blancs avant) ; *action* est un morceau de code source C, qui sera recopié tel quel, au bon endroit, dans la fonction *yylex*. Exemples :

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

La règle

expressionRégulière { *action* }

signifie « à la fin de la reconnaissance d'une chaîne du langage défini par *expressionRégulière* exécutez *action* ». Le traitement par *lex* d'une telle règle consiste donc à recopier l'*action* indiquée à un certain endroit de la fonction *yylex*[18]. Dans les exemples ci-dessus, les actions étant toutes de la forme « `return` *unite* », leur signification est claire : quand une chaîne du texte source est reconnue, la fonction *yylex* se termine en rendant comme résultat l'unité lexicale reconnue. Il faudra appeler de nouveau cette fonction pour que l'analyse du texte source reprenne.

A la fin de la reconnaissance d'une unité lexicale la chaîne acceptée est la valeur de la variable *yytext*, de type chaîne de caractères[19]. Un caractère nul indique la fin de cette chaîne ; de plus, la variable entière *yylen* donne le nombre de ses caractères. Par exemple, la règle suivante reconnaît les nombres entiers et en calcule la valeur dans une variable *yylval* :

[18] C'est parce que ces actions sont copiées dans une fonction qu'on a le droit d'y utiliser l'instruction *return*.

[19] La variable *yytext* est déclarée dans le fichier produit par *lex* ; il vaut mieux ne pas chercher à y faire référence dans d'autres fichiers, car il n'est pas spécifié laquelle des déclarations « `extern char *yytext` » ou « `extern char yytext[]` » est pertinente.

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

EXPRESSIONS RÉGULIÈRES. Les expressions régulières acceptées par *lex* sont une extension de celles définies à la section 2.1. Les méta-caractères précédemment introduits, c'est-à-dire (,), |, ∗, +, ?, [,] et − à l'intérieur des crochets, sont légitimes dans *lex* et y ont le même sens. En outre, on dispose de ceci (liste non exhaustive) :

- un point `.` signifie un caractère quelconque, différent de la marque de fin de ligne,
- on peut encadrer par des guillemets un caractère ou une chaîne, pour éviter que les méta-caractères qui s'y trouvent soient interprétés comme tels. Par exemple, `"."` signifie le caractère `.` (et non pas un caractère quelconque), `" "` signifie un blanc, `"[a-z]"` signifie la chaîne `[a-z]`, etc.,
  D'autre part, on peut sans inconvénient encadrer par des guillemets un caractère ou une chaîne qui n'en avaient pas besoin,
- l'expression `[^`*caractères*`]` signifie : tout caractère n'appartenant pas à l'ensemble défini par `[`*caractères*`]`,
- l'expression « `^`*expressionRégulière* » signifie : toute chaîne reconnue par *expressionRégulière* à la condition qu'elle soit au début d'une ligne,
- l'expression « *expressionRégulière*`$` » signifie : toute chaîne reconnue par *expressionRégulière* à la condition qu'elle soit à la fin d'une ligne.

ATTENTION. Il faut être très soigneux en écrivant les définitions et les règles dans le fichier source *lex*. En effet, tout texte qui n'est pas exactement à sa place (par exemple une définition ou une règle qui ne commencent pas au début de la ligne) sera recopié dans le fichier produit par *lex*. C'est un comportement voulu, parfois utile, mais qui peut conduire à des situations confuses.

ECHO DU TEXTE ANALYSÉ. L'analyseur lexical produit par *lex* prend son texte source sur l'entrée standard[20] et l'écrit, avec certaines modifications, sur la sortie standard. Plus précisement :

- tous les caractères qui ne font partie d'aucune chaîne reconnue sont recopiés sur la sortie standard (ils « traversent » l'analyseur lexical sans en être affectés),
- une chaîne acceptée au titre d'une expression régulière n'est pas recopiée sur la sortie standard.

Bien entendu, pour avoir les chaînes acceptées dans le texte écrit par l'analyseur il suffit de le prévoir dans l'action correspondante. Par exemple, la règle suivante reconnaît les identificateurs et fait en sorte qu'ils figurent dans le texte sorti :

```
[A-Za-z][A-Za-z0-9]*        { printf("%s", yytext); return IDENTIF; }
```

Le texte « `printf("%s", yytext)` » apparaît très fréquemment dans les actions. On peut l'abréger en `ECHO` :

```
[A-Za-z][A-Za-z0-9]*        { ECHO; return IDENTIF; }
```

2.3.2 Un exemple complet

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 le fichier `unitesLexicales.h`.

Pour les opérateurs simples on décide que tout caractère non reconnu par une autre expression régulière est une unité lexicale, et qu'elle est représentée par son propre code ASCII[21]. Pour éviter des collisions entre ces codes ASCII et les unités lexicales nommées, on donne à ces dernières des valeurs supérieures à 255.

Fichier `unitesLexicales.h` :

```
#define IDENTIF  256
#define NOMBRE   257
#define SI       258
#define ALORS    259
#define SINON    260
#define TANTQUE  261
```

[20] On peut changer ce comportement par défaut en donnant une valeur à la variable *yyin*, avant le premier appel de *yylex* ; par exemple : `yyin = fopen(argv[1], "r");`

[21] Cela veut dire qu'on s'en remet au « client » de l'analyseur lexical, c'est-à-dire l'analyseur syntaxique, pour séparer les opérateurs prévus par le langage des caractères spéciaux qui n'en sont pas. Ou, dit autrement, que nous transformons l'erreur « caractère illégal », à priori lexicale, en une erreur syntaxique.

```
#define FAIRE    262
#define RENDRE   263
#define EGAL     264
#define DIFF     265
#define INFEG    266
#define SUPEG    267

extern int valNombre;
extern char valIdentif[];
```

Fichier `analex.l` (le source pour *lex*) :

```
%{
#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;
}
```

Fichier `principal.c` (purement démonstratif) :

```
#include <stdio.h>
#include "unitesLexicales.h"

int main(void) {
    int unite;
    do {
        unite = yylex();
        printf(" (unite: %d", unite);
        if (unite == NOMBRE)
            printf(" val: %d", valNombre);
        else if (unite == IDENTIF)
            printf(" '%s'", valIdentif);
        printf(")\n");
```

```
    } while (unite != 0);
    return 0;
}
```

Fichier `essai.txt` pour essayer l'analyseur :

```
si x == 123 alors
    y = 0;
```

Création d'un exécutable et essai sur le texte précédent (rappelons que *lex* s'appelle *flex* dans le monde Linux) ; `$` est le prompt du système :

```
$ flex analex.l
$ gcc lex.yy.c principal.c -o monprog
$ monprog < essai.txt
si (unite: 258)
x (unite: 256 'x')
== (unite: 264)
123 (unite: 257 val: 123)
alors (unite: 259)
y (unite: 256 'y')
= (unite: 61)
0 (unite: 257 0)
; (unite: 59)
$
```

NOTE. La fonction *yywrap* qui apparaît dans notre fichier source pour *lex* est appelée lorsque l'analyseur rencontre la fin du fichier à analyser[22]. Outre d'éventuelles actions utiles dans telle ou telle application particulière, cette fonction doit rendre une valeur non nulle pour indiquer que le flot d'entrée est définitivement épuisé, ou bien ouvrir un autre flot d'entrée.

2.3.3 Autres utilisations de *lex*

Ce que le programme généré par *lex* fait nécessairement, c'est reconnaître les chaînes du langage défini par les expressions régulières données. Quand une telle reconnaissance est accomplie, on n'est pas obligé de renvoyer une unité lexicale pour signaler la chose ; on peut notamment déclencher l'action qu'on veut et *ne pas retourner à la fonction appelante*. Cela permet d'utiliser *lex* pour effectuer d'autres sortes de programmes que des analyseurs lexicaux.

Par exemple, supposons que nous disposons de textes contenant des indications de prix en francs. Voici comment obtenir rapidement un programme qui met tous ces prix en euros :

```
%%
[0-9]+("."[0-9]*)?[" "\t\n]*F(rancs|".")?[" "\t\n] {
            printf("%.2f EUR ", atof(yytext) / 6.55957); }
%%
int yywrap(void) {
    return 1;
}
int main(void) {
    yylex();
}
```

Le programme précédent exploite le fait que tous les caractères qui ne font pas partie d'une chaîne reconnue sont recopiés sur la sortie ; ainsi, la plupart des caractères du texte donné seront recopiés tels quels. Les chaînes reconnues sont définies par l'expression régulière

```
[0-9]+("."[0-9]*)?[" "\t\n]*F(rancs|".")?[" "\t\n]
```

cette expression se lit, successivement :

[22] Dans certaines versions de *lex* une version simple de la fonction *yywrap*, réduite à `{ return 1; }`, est fournie et on n'a pas à s'en occuper.

– une suite d'au moins un chiffre,
– éventuellement, un point suivi d'un nombre quelconque de chiffres,
– éventuellement, un nombre quelconque de blancs (espaces, tabulations, fins de ligne),
– un `F` majuscule obligatoire,
– éventuellement, la chaîne `rancs` ou un point (ainsi, « `F` », « `F.` » et « `Francs` » sont tous trois acceptés),
– enfin, un blanc obligatoire.

Lorsqu'une chaîne s'accordant à cette syntaxe est reconnue, comme « `99.50 Francs` », la fonction *atof* obtient la valeur que [le début de] cette chaîne représente. Il suffit alors de mettre dans le texte de sortie le résultat de la division de cette valeur par le taux adéquat ; soit, ici, « `15.17 EUR` ».

3 Analyse syntaxique

3.1 Grammaires non contextuelles

Les langages de programmation sont souvent définis par des règles *récursives*, comme : « on a une *expression* en écrivant successivement un terme, `'+'` et une *expression* » ou « on obtient une *instruction* en écrivant à la suite `si`, une expression, `alors`, une *instruction* et, éventuellement, `sinon` et une *instruction* ». Les grammaires non contextuelles sont un formalisme particulièrement bien adapté à la description de telles règles.

3.1.1 Définitions

Une *grammaire non contextuelle*, on dit parfois *grammaire BNF* (pour Backus-Naur form[23]), est un quadruplet $G = (V_T, V_N, S_0, P)$ formé de
– un ensemble V_T de *symboles terminaux*,
– un ensemble V_N de *symboles non terminaux*,
– un symbole $S_0 \in V_N$ particulier, appelé *symbole de départ* ou *axiome*,
– un ensemble P de *productions*, qui sont des règles de la forme

$$S \to S_1 S_2 \dots S_k \text{ avec } S \in V_N \text{ et } S_i \in V_N \cup V_T$$

Compte tenu de l'usage que nous en faisons dans le cadre de l'écriture de compilateurs, nous pouvons expliquer ces éléments de la manière suivante :

1. Les symboles terminaux sont les symboles élémentaires qui constituent les chaînes du langage, les phrases. Ce sont donc les unités lexicales, extraites du texte source par l'analyseur lexical (il faut se rappeler que l'analyseur syntaxique ne connaît pas les caractères dont le texte source est fait, il ne voit ce dernier que comme une suite d'unités lexicales).
2. Les symboles non terminaux sont des variables syntaxiques désignant des *ensembles* de chaînes de symboles terminaux.
3. Le symbole de départ est un symbole non terminal particulier qui désigne le langage en son entier.
4. Les productions peuvent être interprétées de deux manières :
   – comme des *règles d'écriture* (on dit plutôt de *réécriture*), permettant d'engendrer toutes les chaînes correctes. De ce point de vue, la production $S \to S_1 S_2 \dots S_k$ se lit « pour produire un S correct [sous-entendu : de toutes les manières possibles] il fait produire un S_1 [de toutes les manières possibles] suivi d'un S_2 [de toutes les manières possibles] suivi d'un … suivi d'un S_k [de toutes les manières possibles] »,
   – comme des *règles d'analyse*, on dit aussi *reconnaissance*. La production $S \to S_1 S_2 \dots S_k$ se lit alors « pour reconnaître un S, dans une suite de terminaux donnée, il faut reconnaître un S_1 suivi d'un S_2 suivi d'un … suivi d'un S_k »

La définition d'une grammaire devrait donc commencer par l'énumération des ensembles V_T et V_N. En pratique on se limite à donner la liste des productions, avec une convention typographique pour distinguer les symboles terminaux des symboles non terminaux, et on convient que :
– V_T est l'ensemble de tous les symboles terminaux apparaissant dans les productions,
– V_N est l'ensemble de tous les symboles non terminaux apparaissant dans les productions,
– le symbole de départ est le membre gauche de la *première* production.

En outre, on allège les notations en décidant que si plusieurs productions ont le même membre gauche

[23] J. Backus a inventé le langage FORTRAN en 1955, P. Naur le langage Algol en 1963.

$S \to S_{1,1}\, S_{1,2}\, \ldots\, S_{1,k_1}$
$S \to S_{2,1}\, S_{2,2}\, \ldots\, S_{2,k_2}$
$\ldots$
$S \to S_{n,1}\, S_{n,2}\, \ldots\, S_{n,k_n}$

alors on peut les noter simplement

$$S \to S_{1,1}\, S_{1,2}\, \ldots\, S_{1,k_1} \mid S_{2,1}\, S_{2,2}\, \ldots\, S_{2,k_2} \mid \ldots \mid S_{n,1}\, S_{n,2}\, \ldots\, S_{n,k_n}$$

Dans les exemples de ce document, la convention typographique sera la suivante :
– une *chaîne_en_italiques* représente un symbole non terminal,
– une chaîne en `caractères_télétype` ou `"entre guillemets"`, représente un symbole terminal.

A titre d'exemple, voici la grammaire G_1 définissant le langage dont les chaînes sont les expressions arithmétiques formées avec des nombres, des identificateurs et les deux opérateurs `+` et `*`, comme « `60 * vitesse + 200` ». Suivant notre convention, les symboles non terminaux sont *expression*, *terme* et *facteur* ; le symbole de départ est *expression* :

expression → *expression* `"+"` *terme* | *terme*
terme → *terme* `"*"` *facteur* | *facteur* (G_1)
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

3.1.2 Dérivations et arbres de dérivation

Dérivation. Le processus par lequel une grammaire définit un langage s'appelle *dérivation*. Il peut être formalisé de la manière suivante :

Soit $G = (V_T, V_N, S_0, P)$ une grammaire non contextuelle, $A \in V_N$ un symbole non terminal et $\gamma \in (V_T \cup V_N)^*$ une suite de symboles, tels qu'il existe dans P une production $A \to \gamma$. Quelles que soient les suites de symboles α et β, on dit que $\alpha A \beta$ *se dérive en une étape* en la suite $\alpha\gamma\beta$ ce qui s'écrit

$$\alpha A \beta \Rightarrow \alpha\gamma\beta$$

Cette définition justifie la dénomination *grammaire non contextuelle* (on dit aussi *grammaire indépendante du contexte* ou *context free*). En effet, dans la suite $\alpha A \beta$ les chaînes α et β sont le contexte du symbole A. Ce que cette définition dit, c'est que le symbole A se réécrit dans la chaîne γ *quel que soit le contexte* α, β dans le lequel A apparaît.

Si $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n$ on dit que α_0 se dérive en α_n en n étapes, et on écrit

$$\alpha_0 \overset{n}{\Rightarrow} \alpha_n.$$

Enfin, si α se dérive en β en un nombre quelconque, éventuellement nul, d'étapes on dit simplement que α se dérive en β et on écrit

$$\alpha \overset{*}{\Rightarrow} \beta.$$

Soit $G = \{V_T, V_N, S_0, P\}$ une grammaire non contextuelle ; le langage engendré par G est l'ensemble des chaînes de symboles terminaux qui dérivent de S_0 :

$$L(G) = \left\{ w \in {V_T}^* \mid S_0 \overset{*}{\Rightarrow} w \right\}$$

Si $w \in L(G)$ on dit que w est une *phrase* de G. Plus généralement, si $\alpha \in (V_T \cup V_N)^*$ est tel que $S_0 \overset{*}{\Rightarrow} \alpha$ alors on dit que α est une *proto-phrase* de G. Une proto-phrase dont tous les symboles sont terminaux est une phrase.

Par exemple, soit encore la grammaire G_1 :

expression → *expression* `"+"` *terme* | *terme*
terme → *terme* `"*"` *facteur* | *facteur* (G_1)
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

et considérons la chaîne `"60 * vitesse + 200"` qui, une fois lue par l'analyseur lexical, se présente ainsi : $w =$ (`nombre "*" identificateur "+" nombre`). Nous avons *expression* $\overset{*}{\Rightarrow} w$, c'est à dire $w \in L(G_1)$; en effet, nous pouvons exhiber la suite de dérivations en une étape :

expression ⇒ *expression* `"+"` *terme*
⇒ *terme* `"+"` *terme*
⇒ *terme* `"*"` *facteur* `"+"` *terme*
⇒ *facteur* `"*"` *facteur* `"+"` *terme*
⇒ `nombre` `"*"` *facteur* `"+"` *terme*
⇒ `nombre` `"*"` `identificateur` `"+"` *terme*
⇒ `nombre` `"*"` `identificateur` `"+"` *facteur*
⇒ `nombre` `"*"` `identificateur` `"+"` `nombre`

Dérivation gauche. La dérivation précédente est appelée une *dérivation gauche* car elle est entièrement composée de dérivations en une étape dans lesquelles à chaque fois c'est le non-terminal le plus à gauche qui est réécrit. On peut définir de même une *dérivation droite*, où à chaque étape c'est le non-terminal le plus à droite qui est réécrit.

Arbre de dérivation. Soit w une chaîne de symboles terminaux du langage $L(G)$; il existe donc une dérivation telle que $S_0 \stackrel{*}{\Rightarrow} w$. Cette dérivation peut être représentée graphiquement par un arbre, appelé *arbre de dérivation*, défini de la manière suivante :

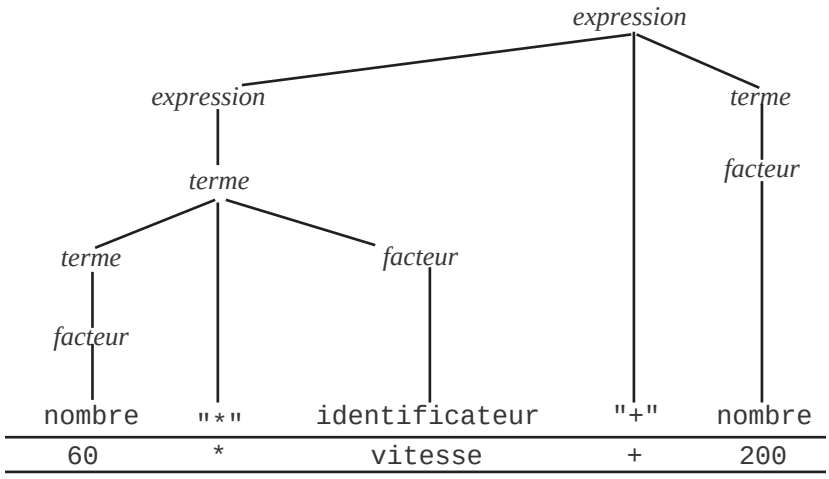


Fig. 5 – Arbre de dérivation

- la racine de l'arbre est le symbole de départ,
- les nœuds intérieurs sont étiquetés par des symboles non terminaux,
- si un nœud intérieur e est étiqueté par le symbole S et si la production $S \to S_1 S_2 \dots S_k$ a été utilisée pour dériver S alors les fils de e sont des nœuds étiquetés, de la gauche vers la droite, par S_1, S_2 … S_k,
- les feuilles sont étiquetées par des symboles terminaux et, si on allonge verticalement les branches de l'arbre (sans les croiser) de telle manière que les feuilles soient toutes à la même hauteur, alors, lues de la gauche vers la droite, elles constituent la chaîne w.

Par exemple, la figure 5 montre l'arbre de dérivation représentant la dérivation donnée en exemple ci-dessus. On notera que l'ordre des dérivations (gauche, droite) ne se voit pas sur l'arbre.

3.1.3 Qualités des grammaires en vue des analyseurs

Etant donnée une grammaire $G = \{V_T, V_N, S_0, P\}$, faire l'analyse syntaxique d'une chaîne $w \in {V_T}^*$ c'est répondre à la question « w appartient-elle au langage $L(G)$? ». Parlant strictement, un analyseur syntaxique est donc un programme qui n'extrait aucune information de la chaîne analysée, il ne fait qu'accepter (par défaut) ou rejeter (en annonçant une erreurs de syntaxe) cette chaîne.

En réalité on ne peut pas empêcher les analyseurs d'en faire un peu plus car, pour prouver que $w \in L(G)$ il faut exhiber une dérivation $S_0 \stackrel{*}{\Rightarrow} w$, c'est-à-dire construire un arbre de dérivation dont la liste des feuilles est w. Or, cet arbre de derivation est déjà une première information extraite de la chaîne source, un début de « compréhension » de ce que le texte signifie.

Nous examinons ici des qualités qu'une grammaire doit avoir et des défauts dont elle doit être exempte pour que la construction de l'arbre de dérivation de toute chaîne du langage soit possible et utile.

Grammaires ambiguës. Une grammaire est ambiguë s'il existe plusieurs dérivations gauches différentes pour une même chaîne de terminaux. Par exemple, la grammaire G_2 suivante est ambiguë :

expression → *expression* `"+"` *expression* | *expression* `"*"` *expression* | *facteur* (G_2)
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

En effet, la figure 6 montre deux arbres de dérivation distincts pour la chaîne `"2 * 3 + 10"`. Ils correspondent aux deux dérivations gauches distinctes :

expression ⇒ *expression* `"+"` *expression* ⇒ *expression* `"*"` *expression* `"+"` *expression*
⇒ *facteur* `"*"` *expression* `"+"` *expression* ⇒ `nombre "*"` *expression* `"+"` *expression*
⇒ `nombre "*"` *facteur* `"+"` *expression* ⇒ `nombre "*" nombre "+"` *expression*
⇒ `nombre "*" nombre "+"` *facteur* ⇒ `nombre "*" nombre "+" nombre`

et

expression ⇒ *expression* `"*"` *expression* ⇒ *facteur* `"*"` *expression*
⇒ `nombre "*"` *expression* ⇒ `nombre "*"` *expression* `"+"` *expression*
⇒ `nombre "*"` *facteur* `"+"` *expression* ⇒ `nombre "*" nombre "+"` *expression*
⇒ `nombre "*" nombre "+"` *facteur* ⇒ `nombre "*" nombre "+" nombre`

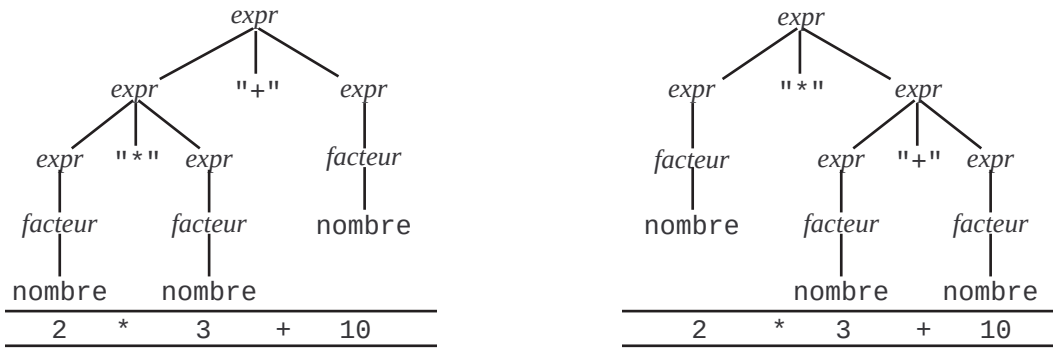


Fig. 6 – Deux arbres de dérivation pour la même chaîne

Deux grammaires sont dites *équivalentes* si elles engendrent le même langage. Il est souvent possible de remplacer une grammaire ambiguë par une grammaire non ambiguë équivalente, mais il n'y a pas une méthode générale pour cela. Par exemple, la grammaire G_1 est non ambiguë et équivalente à la grammaire G_2 ci-dessus.

Grammaires récursives à gauche. Une grammaire est récursive à gauche s'il existe un non-terminal A et une dérivation de la forme $A \stackrel{*}{\Rightarrow} A\alpha$, où α est une chaîne quelconque. Cas particulier, on dit qu'on a une récursivité à gauche *simple* si la grammaire possède une production de la forme $A \to A\alpha$.

La récursivité à gauche ne rend pas une grammaire ambiguë, mais empêche l'écriture d'analyseurs pour cette grammaire, du moins des analyseurs descendants[24].

Par exemple, la grammaire G_1 de la section 3.1.1 est récursive à gauche, et même simplement :

expression → *expression* `"+"` *terme* | *terme*
terme → *terme* `"*"` *facteur* | *facteur* (G_1)
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

Il existe une méthode pour obtenir une grammaire non récursive à gauche équivalente à une grammaire donnée. Dans le cas de la récursivité à gauche simple, cela consiste à remplacer une production telle que

[24]La question est un peu prématurée ici, mais nous verrons qu'un analyseur descendant est un programme composé de fonctions directement déduites des productions. Une production $A \to \ldots$ donne lieu à une fonction *reconnaitre_A* dont le corps est fait d'appels aux fonctions reconnaissant les symboles du membre droit de la production. Dans le cas d'une production $A \to A\alpha$ on se retrouve donc avec une fonction *reconnaitre_A* qui commence par un appel de *reconnaitre_A*. Bonjour la récursion infinie...

$A \to A\alpha \mid \beta$

par les deux productions[25]

$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

En appliquant ce procédé à la grammaire G_1 on obtient la grammaire G_3 suivante :

expression → *terme fin_expression*
fin_expression → `"+"` *terme fin_expression* | ε
terme → *facteur fin_terme* (G_3)
fin_terme → `"*"` *facteur fin_terme* | ε
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

A PROPOS DES ε-PRODUCTIONS. La transformation de grammaire montrée ci-dessus a introduit des productions avec un membre droit vide, ou ε-productions. Si on ne prend pas de disposition particulière, on aura un problème pour l'écriture d'un analyseur, puisqu'une production telle que

fin_expression → `"+"` *terme fin_expression* | ε

impliquera notamment que « une manière de reconnaître une *fin_expression* consiste à ne *rien* reconnaître », ce qui est possible quelle que soit la chaîne d'entrée ; ainsi, notre grammaire semble devenir ambiguë. On résout ce problème en imposant aux analyseurs que nous écrirons la règle de comportement suivante : *dans la dérivation d'un non-terminal, une ε-production ne peut être choisie que lorsqu'aucune autre production n'est applicable.*

Dans l'exemple précédent, cela donne : si la chaîne d'entrée commence par `+` alors on doit nécessairement choisir la première production.

FACTORISATION À GAUCHE. Nous cherchons à écrire des analyseurs *prédictifs*. Cela veut dire qu'à tout moment le choix entre productions qui ont le même membre gauche doit pouvoir se faire, sans risque d'erreur, en comparant le symbole courant de la chaîne à analyser avec les symboles susceptibles de commencer les dérivations des membres droits des productions en compétition.

Une grammaire contenant des productions comme

$A \to \alpha\beta_1 \mid \alpha\beta_2$

viole ce principe car lorsqu'il faut choisir entre les productions $A \to \alpha\beta_1$ et $A \to \alpha\beta_2$ le symbole courant est un de ceux qui peuvent commencer une dérivation de α, et on ne peut pas choisir à coup sûr entre $\alpha\beta_1$ et $\alpha\beta_2$. Une transformation simple, appelée *factorisation à gauche*, corrige ce défaut (si les symboles susceptibles de commencer une réécriture de β_1 sont distincts de ceux pouvant commencer une réécriture de β_2) :

$A \to \alpha A'$
$A' \to \beta_1 \mid \beta_2$

Exemple classique. Les grammaires de la plupart des langages de programmation définissent ainsi l'instruction conditionnelle :

instr_si → `si` *expr* `alors` *instr* | `si` *expr* `alors` *instr* `sinon` *instr*

Pour avoir un analyseur prédictif il faudra opérer une factorisation à gauche :

instr_si → `si` *expr* `alors` *instr fin_instr_si*
fin_instr_si → `sinon` *instr* | ε

Comme précédemment, l'apparition d'une ε-production semble rendre ambiguë la grammaire. Plus précisément, la question suivante se pose : n'y a-t-il pas deux arbres de dérivation possibles pour la chaîne[26] :

`si` α `alors si` β `alors` γ `sinon` δ

Nous l'avons déjà dit, on lève cette ambiguïté en imposant que la ε-production ne peut être choisie que si aucune autre production n'est applicable. Autrement dit, si, au moment où l'analyseur doit dériver le non-terminal *fin_instr_si*, la chaîne d'entrée commence par le terminal `sinon`, alors la production « *fin_instr_si* → `sinon` *instr* » *doit* être appliquée. La figure 7 montre l'arbre de dérivation obtenu pour la chaîne précédente.

[25] Pour se convaincre de l'équivalence de ces deux grammaires il suffit de s'apercevoir que, si α et β sont des symboles terminaux, alors elles engendrent toutes deux le langage $\{\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots\}$

[26] Autre formulation de la même question : « sinon » se rattache-t-il à la première ou à la deuxième instruction « si » ?

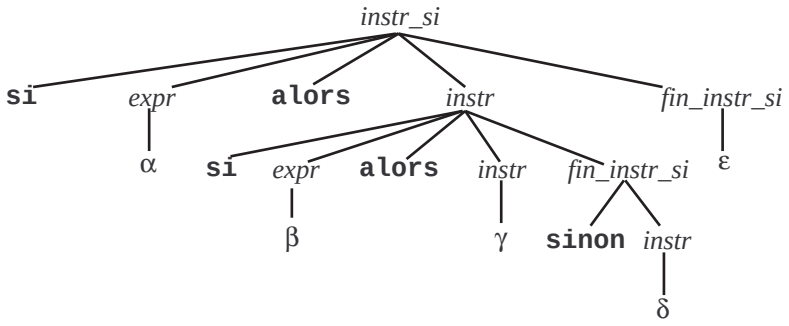


FIG. 7 – Arbre de dérivation pour la chaîne `si` α `alors si` β `alors` γ `sinon` δ

3.1.4 Ce que les grammaires non contextuelles ne savent pas faire

Les grammaires non contextuelles sont un outil puissant, en tout cas plus puissant que les expressions régulières, mais il existe des langages (pratiquement tous les langages de programmation, excusez du peu... !) qu'elles ne peuvent pas décrire complètement.

On démontre par exemple que le langage $L = \{wcw \mid w \in (a|b)^*\}$, où a, b et c sont des terminaux, ne peut pas être décrit par une grammaire non contextuelle. L est fait de phrases comportant deux chaînes de a et b identiques, séparées par un c, comme *ababcabab*. L'importance de cet exemple provient du fait que L modélise l'obligation, qu'ont la plupart des langages, de vérifier que les identificateurs apparaissant dans les instructions ont bien été préalablement déclarés (la première occurrence de w dans wcw correspond à la déclaration d'un identificateur, la deuxième occurrence de w à l'utilisation de ce dernier).

Autrement dit, l'analyse syntaxique ne permet pas de vérifier que les identificateurs utilisés dans les programmes font l'objet de déclarations préalables. Ce problème doit nécessairement être remis à une phase ultérieure d'analyse sémantique.

3.2 Analyseurs descendants

Etant donnée une grammaire $G = (V_T, V_N, S_0, P)$, analyser une chaîne de symboles terminaux $w \in {V_T}^*$ c'est construire un arbre de dérivation prouvant que $S_0 \stackrel{*}{\Rightarrow} w$.

Les grammaires des langages que nous cherchons à analyser ont un ensemble de propriétés qu'on résume en disant que ce sont des grammaires LL(1). Cela signifie qu'on peut en écrire des analyseurs :

- lisant la chaîne source de la gauche vers la droite (gauche = left, c'est le premier L),
- cherchant à construire une dérivation gauche (c'est le deuxième L),
- dans lesquels un seul symbole de la chaîne source est accessible à chaque instant et permet de choisir, lorsque c'est nécessaire, une production parmi plusieurs candidates (c'est le 1 de LL(1)).

A PROPOS DU SYMBOLE ACCESIBLE. Pour réfléchir au fonctionnement de nos analyseurs il est utile d'imaginer que la chaîne source est écrite sur un ruban défilant derrière une fenêtre, de telle manière qu'un seul symbole est visible à la fois ; voyez la figure 8. Un mécanisme permet de faire avancer –jamais reculer– le ruban, pour rendre visible le symbole suivant.

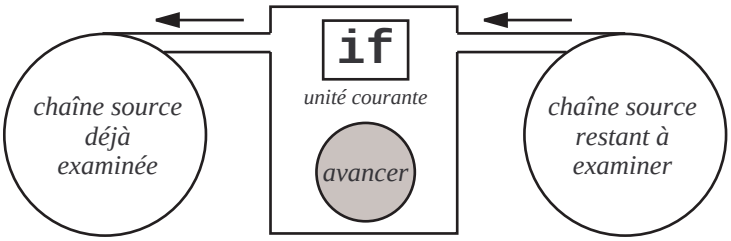


FIG. 8 – Fenêtre à symboles terminaux

Lorsque nous programmerons effectivement des analyseurs, cette « machine à symboles terminaux » ne sera rien d'autre que l'analyseur lexical préalablement écrit ; le symbole visible à la fenêtre sera représenté par une variable *uniteCourante*, et l'opération « faire avancer le ruban » se traduira par *uniteCourante = uniteSuivante()* (ou bien, si c'est *lex* qui a écrit l'analyseur lexical, *uniteCourante = yylex()*).

3.2.1 Principe

Analyseur descendant. Un analyseur *descendant* construit l'arbre de dérivation de la racine (le symbole de départ de la grammaire) vers les feuilles (la chaîne de terminaux).

Pour en décrire schématiquement le fonctionnement nous nous donnons une fenêtre à symboles terminaux comme ci-dessus et une pile de symboles, c'est-à-dire une séquence de symboles terminaux et non terminaux à laquelle on ajoute et on enlève des symboles par une même extrémité, en l'occurrence l'extrémité de gauche (c'est une pile couchée à l'horizontale, qui se remplit de la droite vers la gauche).

Initialisation. Au départ, la pile contient le symbole de départ de la grammaire et la fenêtre montre le premier symbole terminal de la chaîne d'entrée.

Itération. Tant que la pile n'est pas vide, répéter les opérations suivantes

- si le symbole au sommet de la pile (c.-à-d. le plus à gauche) est un terminal α
  - si le terminal visible à la fenêtre est le même symbole α, alors
    - dépiler le symbole au sommet de la pile et
    - faire avancer le terminal visible à la fenêtre,
  - sinon, signaler une erreur (par exemple afficher « α attendu ») ;
- si le symbole au sommet de la pile est un non terminal S
  - s'il y a une seule production $S \to S_1 S_2 \dots S_k$ ayant S pour membre gauche alors dépiler S et empiler $S_1 S_2 \dots S_k$ à la place,
  - s'il y a plusieurs productions ayant S pour membre gauche, alors d'après le terminal visible à la fenêtre, sans faire avancer ce dernier, choisir l'unique production $S \to S_1 S_2 \dots S_k$ pouvant convenir, dépiler S et empiler $S_1 S_2 \dots S_k$.

Terminaison. Lorsque la pile est vide

- si le terminal visible à la fenêtre est la marque qui indique la fin de la chaîne d'entrée alors l'analyse a réussi : la chaîne appartient au langage engendré par la grammaire,
- sinon, signaler une erreur (par exemple, afficher « caractères inattendus à la suite d'un texte correct »).

A titre d'exemple, voici la reconnaissance par un tel analyseur du texte `"60 * vitesse + 200"` avec la grammaire G_3 de la section 3.1.3 :

expression → *terme fin_expression*
fin_expression → `"+"` *terme fin_expression* | ε
terme → *facteur fin_terme* (G_3)
fin_terme → `"*"` *facteur fin_terme* | ε
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

La chaîne d'entrée est donc `(nombre "*" identif "+" nombre)`. Les états successifs de la pile et de la fenêtre sont les suivants :

fenêtre	pile
`nombre`	*expression*
`nombre`	*terme fin_expression*
`nombre`	*facteur fin_terme fin_expression*
`nombre`	`nombre` *fin_terme fin_expression*
`"*"`	*fin_terme fin_expression*
`"*"`	`"*"` *facteur fin_terme fin_expression*
`identificateur`	*facteur fin_terme fin_expression*
`identificateur`	`identificateur` *fin_terme fin_expression*
`"+"`	*fin_terme fin_expression*
`"+"`	ε *fin_expression*
`"+"`	*fin_expression*
`"+"`	`"+"` *terme fin_expression*
`nombre`	*terme fin_expression*
`nombre`	*facteur fin_expression*
`nombre`	`nombre` *fin_expression*
¶	*fin_expression*
¶	ε
¶	

Lorsque la pile est vide, la fenêtre exhibe ¶, la marque de fin de chaîne. La chaîne donnée appartient donc bien au langage considéré.

3.2.2 Analyseur descendant non récursif

Nous ne développerons pas cette voie ici, mais nous pouvons remarquer qu'on peut réaliser des programmes itératifs qui implantent l'algorithme expliqué à la section précédente.

La plus grosse difficulté est le choix d'une production chaque fois qu'il faut dériver un non terminal qui est le membre gauche de plusieurs productions de la grammaire. Comme ce choix ne dépend que du terminal visible à la fenêtre, on peut le faire, et de manière très efficace, à l'aide d'une table à double entrée, appelée *table d'analyse*, calculée à l'avance, représentant une fonction $Choix : V_N \times V_T \rightarrow P$ qui à un couple (S, α) formé d'un non terminal (le sommet de la pile) et un terminal (le symbole visible à la fenêtre) associe la production qu'il faut utiliser pour dériver S.

Pour avoir un analyseur descendant non récursif il suffit alors de se donner une fenêtre à symboles terminaux (c'est-à-dire un analyseur lexical), une pile de symboles comme expliqué à la section précédente, une table d'analyse comme expliqué ici et un petit programme qui implante l'algorithme de la section précédente, dans lequel la partie « choisir la production... » se résume à une consultation de la table $P = Choix(S, \alpha)$.

En définitive, un analyseur descendant est donc un couple formé d'une table dont les valeurs sont intimement liées à la grammaire analysée et d'un programme tout à fait indépendant de cette grammaire.

3.2.3 Analyse par descente récursive

A l'opposé du précédent, un analyseur par *descente récursive* est un type d'analyseur descendant dans lequel le programme de l'analyseur est étroitement lié à la grammaire analysée. Voici les principes de l'écriture d'un tel analyseur :

1. Chaque groupe de productions ayant le même membre gauche S donne lieu à une fonction *void reconnaitre_S(void)*, ou plus simplement *void S(void)*. Le corps de cette fonction se déduit des membres droits des productions en question, comme expliqué ci-après.
2. Lorsque plusieurs productions ont le même membre gauche, le corps de la fonction correspondante est une conditionnelle (instruction *if*) ou un aiguillage (instruction *switch*) qui, d'après le symbole terminal visible à la fenêtre, sélectionne l'exécution des actions correspondant au membre droit de la production pertinente. Dans cette sélection, le symbole visible à la fenêtre n'est pas modifié.
3. Une *séquence* de symboles $S_1 S_2 \ldots S_n$ dans le membre droit d'une production donne lieu, dans la fonction correspondante, à une *séquence* d'instructions traduisant les actions « reconnaissance de S_1 », « reconnaissance de S_2 », ...« reconnaissance de S_n ».

4. Si S est un symbole non terminal, l'action « reconnaissance de S » se réduit à l'appel de fonction *reconnaitre_S()*.
5. Si α est un symbole terminal, l'action « reconnaissance de α » consiste à considérer le symbole terminal visible à la fenêtre et
   – s'il est égal à α, faire passer la fenêtre sur le symbole suivant[27],
   – sinon, annoncer une erreur (par exemple, afficher « α attendu »).

L'ensemble des fonctions écrites selon les prescriptions précédentes forme l'analyseur du langage considéré. L'initialisation de l'analyseur consiste à positionner la fenêtre sur le premier terminal de la chaîne d'entrée. On lance l'analyse en appellant la fonction associée au symbole de départ de la grammaire. Au retour de cette fonction

– si la fenêtre à terminaux montre la marque de fin de chaîne, l'analyse a réussi,
– sinon la chaîne est erronée[28] (on peut par exemple afficher le message « caractères illégaux après une expression correcte »).

Exemple. Voici encore la grammaire G_3 de la section 3.1.3 :

expression → *terme fin_expression*
fin_expression → `"+"` *terme fin_expression* | ε
terme → *facteur fin_terme* (G_3)
fin_terme → `"*"` *facteur fin_terme* | ε
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

et voici l'analyseur par descente récursive correspondant :

```
void expression(void) {
    terme();
    fin_expression();
}
void fin_expression(void) {
    if (uniteCourante == '+') {
        terminal('+');
        terme();
        fin_expression();
    }
    else
        /* rien */;
}
void terme(void) {
    facteur();
    fin_terme();
}
void fin_terme(void) {
    if (uniteCourante == '*') {
        terminal('*');
        facteur();
        fin_terme();
    }
    else
        /* rien */;
}
void facteur(void) {
    if (uniteCourante == NOMBRE)
        terminal(NOMBRE);
```

[27] Notez que c'est ici le seul endroit, dans cette description, où il est indiqué de faire avancer la fenêtre.

[28] Une autre attitude possible -on l'a vue adoptée par certains compilateurs de Pascal- consiste à ne rien vérifier au retour de la fonction associée au symbole de départ. Cela revient à considérer que, si le texte source comporte un programme correct, peu importent les éventuels caractères qui pourraient suivre.

```
    else if (uniteCourante == IDENTIFICATEUR)
        terminal(IDENTIFICATEUR);
    else {
        terminal('(');
        expression();
        terminal(')');
    }
}
```

La reconnaissance d'un terminal revient fréquemment dans un analyseur. Nous en avons fait une fonction séparée (on suppose que *erreur* est une fonction qui affiche un message et termine le programme) :

```
void terminal(int uniteVoulue) {
    if (uniteCourante == uniteVoulue)
        lireUnite();
    else
        switch (uniteVoulue) {
            case NOMBRE:
                erreur("nombre attendu");
            case IDENTIFICATEUR:
                erreur("identificateur attendu");
            default:
                erreur("%c attendu", uniteVoulue);
        }
}
```

NOTE. Nous avons écrit le programme précédent en appliquant systématiquement les règles données plus haut, obtenant ainsi un analyseur correct dont la structure reflète la grammaire donnée. Mais il n'est pas interdit de pratiquer ensuite certaines simplifications, ne serait-ce pour rattraper certaines maladresses de notre démarche. L'appel généralisé de la fonction *terminal*, notamment, est à l'origine de certains test redondants. Par exemple, la fonction *fin_expression* commence par les deux lignes

```
    if (uniteCourante == '+') {
        terminal('+');
    ...
```

si on développe l'appel de *terminal*, la maladresse de la chose devient évidente

```
    if (uniteCourante == '+') {
        if (uniteCourante == '+')
            lireUnite();
        ...
```

Une version plus raisonnable des fonctions *fin_expression* et *facteur* serait donc :

```
void fin_expression(void) {
    if (uniteCourante == '+') {
        lireUnite();
        terme();
        fin_expression();
    }
    else
        /* rien */;
}
...
void facteur(void) {
    if (uniteCourante == NOMBRE)
        lireUnite();
    else if (uniteCourante == IDENTIFICATEUR)
        lireUnite();
    else {
```

```
        terminal('(');
        expression();
        terminal(')');
    }
}
```

Comme elle est écrite ci-dessus, la fonction *facteur* aura tendance à faire passer toutes les erreurs par le diagnostic « '(' attendu », ce qui risque de manquer d'à propos. Une version encore plus raisonnable de cette fonction serait

```
void facteur(void) {
    if (uniteCourante == NOMBRE)
        lireUnite();
    else if (uniteCourante == IDENTIFICATEUR)
        lireUnite();
    else if (uniteCourante == '(') {
        lireUnite('(');
        expression();
        terminal(')');
    }
    else
        erreur("nombre, identificateur ou '(' attendus ici");
}
```

3.3 Analyseurs ascendants

3.3.1 Principe

Comme nous l'avons dit, étant données une grammaire $G = \{V_T, V_N, S_0, P\}$ et une chaîne $w \in {V_T}^*$, le but de l'analyse syntaxique est la construction d'un arbre de dérivation qui prouve $w \in L(G)$. Les méthodes descendantes construisent cet arbre en partant du symbole de départ de la grammaire et en « allant vers » la chaîne de terminaux. Les méthodes ascendantes, au contraire, partent des terminaux qui constituent la chaîne d'entrée et « vont vers » le symbole de départ.

Le principe général des méthodes ascendantes est de maintenir une pile de symboles[29] dans laquelle sont empilés (l'empilement s'appelle ici *décalage*) les terminaux au fur et à mesure qu'ils sont lus, tant que les symboles au sommet de la pile ne sont pas le membre droit d'une production de la grammaire. Si les k symboles du sommet de la pile constituent le membre droit d'une production alors ils peuvent être dépilés et remplacés par le membre gauche de cette production (cette opération s'appelle *réduction*). Lorsque dans la pile il n'y a plus que le symbole de départ de la grammaire, si tous les symboles de la chaîne d'entrée ont été lus, l'analyse a réussi.

Le problème majeur de ces méthodes est de faire deux sortes de choix :

- si les symboles au sommet de la pile constituent le membre droit de deux productions distinctes, laquelle utiliser pour effectuer la réduction ?
- lorsque les symboles au sommet de la pile sont le membre droit d'une ou plusieurs productions, faut-il réduire tout de suite, ou bien continuer à décaler, afin de permettre ultérieurement une réduction plus juste ?

A titre d'exemple, avec la grammaire G_1 de la section 3.1.1 :

expression → *expression* `"+"` *terme* | *terme*
terme → *terme* `"*"` *facteur* | *facteur* (G_1)
facteur → `nombre` | `identificateur` | `"("` *expression* `")"`

voici la reconnaissance par un analyseur ascendant du texte d'entrée `"60 * vitesse + 200"`, c'est-à-dire la chaîne de terminaux `(nombre "*" identificateur "+" nombre)` :

[29] Comme précédemment, cette pile est un tableau couché à l'horizontale ; mais cette fois elle grandit de la gauche vers la droite, c'est-à-dire que son sommet est son extrémité droite.

fenêtre	pile	action
`nombre`		décalage
`"*"`	`nombre`	réduction
`"*"`	*facteur*	réduction
`"*"`	*terme*	décalage
`identificateur`	*terme* `"*"`	décalage
`"+"`	*terme* `"*"` `identificateur`	réduction
`"+"`	*terme* `"*"` *facteur*	réduction
`"+"`	*terme*	réduction
`"+"`	*expression*	décalage
`nombre`	*expression* `"+"`	décalage
¶	*expression* `"+"` `nombre`	réduction
¶	*expression* `"+"` *facteur*	réduction
¶	*expression* `"+"` *terme*	réduction
¶	*expression*	succès

On dit que les méthodes de ce type effectuent une *analyse par décalage-réduction*. Comme le montre le tableau ci-dessus, le point important est le choix entre réduction et décalage, chaque fois qu'une réduction est possible. Le principe est : les réductions pratiquées réalisent la *construction inverse d'une dérivation droite*.

Par exemple, le réductions faites dans l'analyse précédente construisent, à l'envers, la dérivation droite suivante :

expression ⇒ *expression* `"+"` *terme*
⇒ *expression* `"+"` *facteur*
⇒ *expression* `"+"` `nombre`
⇒ *terme* `"+"` `nombre`
⇒ *terme* `"*"` *facteur* `"+"` `nombre`
⇒ *terme* `"*"` `identificateur` `"+"` `nombre`
⇒ *facteur* `"*"` `identificateur` `"+"` `nombre`
⇒ `nombre` `"*"` `identificateur` `"+"` `nombre`

3.3.2 Analyse LR(k)

Il est possible, malgré les apparences, de construire des analyseurs ascendants plus efficaces que les analyseurs descendants, et acceptant une classe de langages plus large que la classe des langages traités par ces derniers.

Le principal inconvénient de ces analyseurs est qu'ils nécessitent des tables qu'il est extrêmement difficile de construire à la main. Heureusement, des outils existent pour les construire automatiquement, à partir de la grammaire du langage ; la section 3.4 présente *yacc*, un des plus connus de ces outils.

Les analyseurs LR(k) lisent la chaîne d'entrée de la gauche vers la droite (d'où le L), en construisant l'inverse d'une dérivation droite (d'où le R) avec une vue sur la chaîne d'entrée large de k symboles ; lorsqu'on dit simplement LR on sous-entend $k = 1$, c'est le cas le plus fréquent.

Etant donnée une grammaire $G = (V_T, V_N, S_0, P)$, un analyseur LR est constitué par la donnée d'un ensemble d'états E, d'une fenêtre à symboles terminaux (c'est-à-dire un analyseur lexical), d'une pile de doublets (s, e) où $s \in E$ et $e \in V_T$ et de deux tables *Action* et *Suivant*, qui représentent des fonctions :

$Action : E \times V_T \to (\{\, decaler \,\} \times E) \cup (\{reduire\} \times P) \cup \{\, succes,\ erreur \,\}$
$Suivant : E \times V_N \to E$

Un analyseur LR comporte enfin un programme, indépendant du langage analysé, qui exécute les opérations suivantes :

Initialisation. Placer la fenêtre sur le premier symbole de la chaîne d'entrée et vider la pile.

Itération. Tant que c'est possible, répéter :

soit s l'état au sommet de la pile et α le terminal visible à la fenêtre
si $Action(s, \alpha) = (decaler, s')$
empiler (α, s')
placer la fenêtre sur le prochain symbole de la chaîne d'entree

sinon, si $Action(s, \alpha) = (reduire, A \rightarrow \beta)$
    dépiler autant d'éléments de la pile qu'il y a de symboles dans β
    (soit (γ, s') le nouveau sommet de la pile)
    empiler $(A, Suivant(s', A))$
sinon, si $Action(s, \alpha) = succes$
    arrêt
sinon
    erreur.

NOTE. En réalité, notre pile est redondante. Les états de l'analyseur représentent les diverses configurations dans lesquelles la pile peut se trouver, il n'y a donc pas besoin d'empiler les symboles, les états suffisent. Nous avons utilisé une pile de couples $(etat, symbole)$ pour clarifier l'explication.

3.4 *Yacc*, un générateur d'analyseurs syntaxiques

Comme nous l'avons indiqué, les tables *Action* et *Suivant* d'un analyseur LR sont difficiles à construire manuellement, mais il existe des outils pour les déduire automatiquement des productions de la grammaire considérée.

Le programme *yacc*[30] est un tel générateur d'analyseurs syntaxiques. Il prend en entrée un fichier source constitué essentiellement des productions d'une grammaure non contextuelle G et sort à titre de résultat un programme C qui, une fois compilé, est un analyseur syntaxique pour le langage $L(G)$.

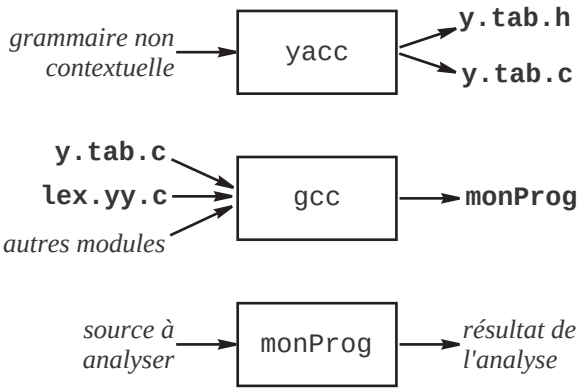


FIG. 9 – Utilisation courante de *yacc*

Dans la description de la grammaire donnée à *yacc* on peut associer des actions sémantiques aux productions ; ce sont des bouts de code source C que *yacc* place aux bons endroits de l'analyseur construit. Ce dernier peut ainsi exécuter des actions ou produire des informations déduites du texte source, c'est-à-dire devenir un compilateur.

Un analyseur syntaxique requiert pour travailler un analyseur lexical qui lui délivre le flot d'entrée sous forme d'unités lexicales successives. Par défaut, *yacc* suppose que l'analyseur lexical disponible a été fabriqué par *lex*. Autrement dit, sans qu'il faille de déclaration spéciale pour cela, le programme produit par *yacc* comporte des appels de la fonction *yylex* aux endroits où l'acquisition d'une unite lexicale est nécessaire.

3.4.1 Structure d'un fichier source pour *yacc*

Un fichier source pour *yacc* doit avoir un nom terminé par « `.y` ». Il est fait de trois sections, délimitées par deux lignes réduites au symbole `%%` :

```
%{
    déclarations pour le compilateur C
%}
```

[30]Dans le monde Linux on trouve une version améliorée de *yacc*, nommée *bison*, qui appartient à la famille GNU. Le nom de *yacc* provient de "yet another compiler compiler" (« encore un compilateur de compilateurs... »), *bison* est issu de la confusion de *yacc* avec *yak* ou *yack*, gros bovin du Tibet.

déclarations pour yacc
%%
règles (productions + actions sémantiques)
%%
fonctions C supplémentaires

Les parties « *déclarations pour le compilateur C* » et « *fonctions C supplémentaires* » sont simplement recopiées dans le fichier produit, respectivement au début et à la fin de ce fichier. Chacune de ces deux parties peut être absente.

Dans la partie « *déclarations pour* yacc » on rencontre souvent les déclarations des unités lexicales, sous une forme qui laisse *yacc* se charger d'inventer des valeurs conventionnelles :

```
%token NOMBRE IDENTIF VARIABLE TABLEAU FONCTION
%token OU ET EGAL DIFF INFEG SUPEG
%token SI ALORS SINON TANTQUE FAIRE RETOUR
```

Ces déclarations d'unités lexicales intéressent *yacc*, qui les utilise, mais aussi *lex*, qui les manipule en tant que résultats de la fonction *yylex*. Pour cette raison, *yacc* produit[31] un fichier supplémentaire, nommé `y.tab.h`[32], destiné à être inclus dans le source *lex* (au lieu du fichier que nous avons appelé `unitesLexicales.h` dans l'exemple de la section 2.3.2). Par exemple, le fichier produit pour les déclarations ci-dessus ressemble à ceci :

```
#define NOMBRE      257
#define IDENTIF     258
#define VARIABLE    259
...
#define FAIRE       272
#define RETOUR      273
```

Notez que *yacc* considère que tout caractère est susceptible de jouer le rôle d'unité lexicale (comme cela a été le cas dans notre exemple de la section 2.3.2) ; pour cette raison, ces constantes sont numérotées à partir de 256.

Spécification de V_T, V_N et S_0. Dans un fichier source *yacc* :
- les caractères simples, encadrés par des apostrophes comme dans les programmes C, et les identificateurs mentionnés dans les déclarations `%token` sont tenus pour des symboles terminaux,
- tous les autres identificateurs apparaissant dans les productions sont considères comme des symboles non terminaux,
- par défaut, le symbole de départ est le membre gauche de la *première* règle.

Règles de traduction. Une règle de traduction est un ensemble de productions ayant le même membre gauche, chacune associé à une action sémantique.

Une action sémantique (cf. section 3.4.2) est un morceau de code source C encadré par des accolades. C'est un code que l'analyseur exécutera lorsque la production correspondante aura été utilisée dans une réduction. Si on écrit un analyseur « pur », c'est-à-dire un analyseur qui ne fait qu'accepter ou rejeter la chaîne d'entrée, alors il n'y a pas d'actions sémantiques et les règles de traduction sont simplement les productions de la grammaire.

Dans les règles de traduction, le méta-symbole $\rightarrow$ est indiqué par deux points « : » et chaque règle (c'est-à-dire chaque groupe de productions avec le même membre gauche) est terminée par un point-virgule « ; ». La barre verticale « | » a la même signification que dans la notation des grammaires.

Par exemple, voici comment la grammaire G_1 de la section 3.1.1 :

expression $\rightarrow$ *expression* `"+"` *terme* | *terme*
terme $\rightarrow$ *terme* `"*"` *facteur* | *facteur* (G_1)
facteur $\rightarrow$ `nombre` | `identificateur` | `"("` *expression* `")"`

serait écrite dans un fichier source *yacc* (pour obtenir un analyseur pur, sans actions sémantiques) :

```
%token nombre identificateur
%%
expression : expression '+' terme | terme ;
terme : terme '*' facteur | facteur ;
facteur : nombre | identificateur | '(' expression ')' ;
```

[31] Du moins si on le lance avec l'option `-d`, comme dans « `yacc -d maSyntaxe.y` ».
[32] Dans le cas de *bison* les noms des fichiers « `.tab.c` » et « `.tab.h` » reflètent le nom du fichier source « `.y` ».

L'analyseur syntaxique se présente comme une fonction *int yyparse(void)*, qui rend 0 lorsque la chaîne d'entrée est acceptée, une valeur non nulle dans le cas contraire. Pour avoir un analyseur syntaxique autonome il suffit donc d'ajouter, en troisième section du fichier précédent :

```
%%
int main(void) {
    if (yyparse() == 0)
        printf("Texte correct\n");
}
```

En réalité, il faut aussi écrire la fonction appelée en cas d'erreur. C'est une fonction de prototype *void yyerror(char *message)*, elle est appelée par l'analyseur avec un message d'erreur (par défaut « *parse error* », ce n'est pas très informatif !). Par exemple :

```
void yyerror(char *message) {
    printf(" <<< %s\n", message);
}
```

N.B. L'effet précis de l'instruction ci-dessus, c'est-à-dire la présentation effective des messages d'erreur, dépend de la manière dont l'analyseur lexical écrit les unités lexicales au fur et à mesure de l'analyse.

3.4.2 Actions sémantiques et valeurs des attributs

Une *action sémantique* est une séquence d'instructions C écrite, entre accolades, à droite d'une production. Cette séquence est recopiée par *yacc* dans l'analyseur produit, de telle manière qu'elle sera exécutée, pendant l'analyse, lorsque la production correspondante aura été employée pour faire une réduction (cf. « analyse par décalage-réduction » à la section 3.3).

Voici un exemple simple, mais complet. Le programme suivant lit une expression arithmétique infixée[33] formée de nombres, d'identificateurs et des opérateurs + et ∗, et écrit la représentation en notation postfixée[34] de la même expression.

Fichier `lexique.l` :

```
%{
#include "syntaxe.tab.h"
extern char nom[];  /* chaîne de caractères partagée avec l'analyseur syntaxique */
%}
chiffre  [0-9]
lettre   [A-Za-z]

%%

[" "\t\n]                       { }
{chiffre}+                      { yylval = atoi(yytext); return nombre; }
{lettre}({lettre}|{chiffre})*   { strcpy(nom, yytext); return identificateur; }
.                               { return yytext[0]; }

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

Fichier `syntaxe.y` :

```
%{
char nom[256];  /* chaîne de caractères partagée avec l'analyseur lexical */
%}
```

[33] Dans la notation infixée, un opérateur binaire est écrit entre ses deux opérandes. C'est la notation habituelle, et elle est ambiguë ; c'est pourquoi on lui associe un système d'associativité et de priorité des opérateurs, et la possibilité d'utiliser des parenthèses.

[34] Dans la notation postifixée, appelée aussi *notation polonaise inverse*, un opérateur binaire est écrit à la suite de ses deux opérandes ; cette notation n'a besoin ni de priorités des opérateurs ni de parenthèses, et elle n'est pas ambiguë.

```
%token nombre identificateur

%%
expression  : expression '+' terme  { printf(" +"); }
            | terme
            ;
terme       : terme '*' facteur     { printf(" *"); }
            | facteur
            ;
facteur     : nombre                { printf(" %d", yylval); }
            | identificateur        { printf(" %s", nom); }
            | '(' expression ')'
            ;
%%
void yyerror(char *s) {
    printf("<<< \n%s", s);
}

main() {
    if (yyparse() == 0)
        printf(" Expression correcte\n");
}
```

Construction et essai de cet analyseur :

```
$ flex lexique.l
$ bison syntaxe.y
$ gcc lex.yy.c syntaxe.tab.c -o rpn
$ rpn
2 + A * 100
 2 A 100 * + Expression correcte
$ rpn
2 * A + 100
 2 A * 100 + Expression correcte
$ rpn
2 * (A + 100)
 2 A 100 + * Expression correcte
$
```

Attributs. Un symbole, terminal ou non terminal, peut avoir un *attribut*, dont la valeur contribue à la caractérisation du symbole. Par exemple, dans les langages qui nous intéressent, la reconnaissance du lexème `"2001"` donne lieu à l'unité lexicale `NOMBRE` avec l'attribut 2001.

Un analyseur lexical produit par *lex* transmet les attributs des unités lexicales à un analyseur syntaxique produit par *yacc* à travers une variable *yylval* qui, par défaut[35], est de type *int*. Si vous allez voir le fichier « `.tab.h` » fabriqué par *yacc* et destiné à être inclus dans l'analyseur lexical, vous y trouverez, outre les définitions des codes des unités lexicales, les déclarations :

```
#define YYSTYPE int
...
extern YYSTYPE yylval;
```

Nous avons dit que les actions sémantiques sont des bouts de code C que *yacc* se borne à recopier dans l'analyseur produit. Ce n'est pas tout à fait exact, dans les actions sémantiques on peut mettre certains symboles bizarres, que *yacc* remplace par des expressions C correctes. Ainsi, `$1`, `$2`, `$3`, etc. désignent les valeurs des attributs des symboles constituant le membre droit de la production concernée, tandis que `$$` désigne la valeur de l'attribut du symbole qui est le membre gauche de cette production.

[35] Un mécanisme puissant et relativement simple permet d'avoir des attributs polymorphes, pouvant prendre plusieurs types distincts. Nous ne l'étudierons pas dans le cadre de ce cours.

L'action sémantique { `$$ = $1 ;` } est implicite et il n'y a pas besoin de l'écrire.

Par exemple, voici notre analyseur précédent, modifié (légèrement) pour en faire un calculateur de bureau effectuant les quatre opérations élémentaires sur des nombres entiers, avec gestion des parenthèses et de la priorité des opérateurs :

Fichier `lexique.l` : le même que précédemment, à ceci près que les identificateurs ne sont plus reconnus.

Fichier `syntaxe.y` :

```
%{
void yyerror(char *);
%}
%token nombre

%%
session     : session expression '='  { printf("résultat : %d\n", $2); }
            |
            ;
expression  : expression '+' terme    { $$ = $1 + $3; }
            | expression '-' terme    { $$ = $1 - $3; }
            | terme
            ;
terme       : terme '*' facteur       { $$ = $1 * $3; }
            | terme '/' facteur       { $$ = $1 / $3; }
            | facteur
            ;
facteur     : nombre
            | '(' expression ')'      { $$ = $2; }
            ;
%%
void yyerror(char *s) {
    printf("<<< \n%s", s);
}
main() {
    yyparse();
    printf("Au revoir!\n");
}
```

Exemple d'utilisation ; ¶ représente la touche « fin de fichier », qui dépend du système utilisé (Ctrl-D, pour UNIX) :

```
$ go
2 + 3 =
résultat : 5
(2 + 3)*(1002 - 1 - 1) =
résultat : 5000
¶
Au revoir!
```

3.4.3 Conflits et ambiguïtés

Voici encore une grammaire équivalente à la précédente, mais plus compacte :

```
...
%%
session     : session expression '='     { printf("résultat: %d\n", $2); }
            |
            ;
```

```
expression  : expression '+' expression  { $$ = $1 + $3; }
            | expression '-' expression  { $$ = $1 - $3; }
            | expression '*' expression  { $$ = $1 * $3; }
            | expression '/' expression  { $$ = $1 / $3; }
            | '(' expression ')'         { $$ = $2; }
            | nombre
            ;
%%
...
```

Nous avons vu à la section 3.1.3 que cette grammaire est ambiguë ; elle provoquera donc des conflits. Lorsqu'il rencontre un conflit[36], *yacc* applique une règle de résolution par défaut et continue son travail ; à la fin de ce dernier, il indique le nombre total de conflits rencontrés et arbitrairement résolus. Il est impératif de comprendre la cause de ces conflits et il est fortement recommandé d'essayer de les supprimer (par exemple en transformant la grammaire). Les conflits possibles sont :

1. Décaler ou réduire ? (« shift/reduce conflict »). Ce conflit se produit lorsque l'algorithme de *yacc* n'arrive pas à choisir entre décaler et réduire, car les deux actions sont possibles et n'amènent pas l'analyse à une impasse. Un exemple typique de ce conflit a pour origine la grammaire usuelle de l'instruction conditionnelle

$$\textit{instr\_si} \rightarrow \texttt{si}\ \textit{expr}\ \texttt{alors}\ \textit{instr} \mid \texttt{si}\ \textit{expr}\ \texttt{alors}\ \textit{instr}\ \texttt{sinon}\ \textit{instr}$$

Le conflit apparaît pendant l'analyse d'une chaîne comme

$$\texttt{si}\ \alpha\ \texttt{alors}\ \texttt{si}\ \beta\ \texttt{alors}\ \gamma\ \texttt{sinon}\ \delta$$

lorsque le symbole courant est `sinon` : au sommet de la pile se trouve alors `si` β `alors` γ, et la question est : faut-il réduire ces symboles en *instr_si* (ce qui revient à associer la partie `sinon` δ au premier `si`) ou bien faut-il décaler (ce qui provoquera plus tard une réduction revenant à associer la partie `sinon` δ au second `si`) ?

Résolution par défaut : l'analyseur fait le décalage (c'est un comportement « glouton » : chacun cherche à manger le plus de terminaux possibles).

2. Comment réduire ? (« reduce/reduce conflict ») Ce conflit se produit lorsque l'algorithme ne peut pas choisir entre deux productions distinctes dont les membres droits permettent tous deux de réduire les symboles au sommet de la pile.

On trouve un exemple typique d'un tel conflit dans les grammaires de langages (il y en a !) où on note avec des parenthèses aussi bien les appels de fonctions que les accès aux tableaux. Sans rentrer dans les détails, il est facile d'imaginer qu'on trouvera dans de telles grammaires des productions complètement différentes avec les mêmes membres droits. Par exemple, la production définissant un appel de fonction et celle définissant un accès à un tableau pourraient ressembler à ceci :

...
appel_de_fonction → *identificateur* '(' *liste_expressions* ')'
...
acces_tableau → *identificateur* '(' *liste_expressions* ')'
...

La résolution par défaut est : dans l'ordre où les règles sont écrites dans le fichier source pour *yacc*, on préfère la *première* production. Comme l'exemple ci-dessus le montre[37], cela ne résout pas souvent bien le problème.

Grammaires d'opérateurs. La grammaire précédente, ou du moins sa partie utile, la règle *expression*, vérifie ceci :

- aucune règle ne contient d'ε-production,
- aucune règle ne contient une production ayant deux non-terminaux consécutifs.

[36] N'oubliez pas que *yacc* ne fait pas une analyse, mais un analyseur. Ce qu'il détecte en réalité n'est pas un conflit, mais la possibilité que l'analyseur produit en ait ultérieurement, du moins sur certains textes sources.

[37] Le problème consistant à choisir assez tôt entre appel de fonction et accès à un tableau, lorsque les notations sont les mêmes, est souvent résolu par des considérations sémantiques : l'identificateur qui précède la parenthèse est censé avoir été déclaré, on consulte donc la table de symboles pour savoir si c'est un nom de procédure ou bien un nom de tableau. Du point de vue de la puissance des analyseurs syntaxiques, c'est donc plutôt un aveu d'impuissance...

De telles grammaires s'appellent des *grammaires d'opérateurs*. Nous n'en ferons pas l'étude détaillée ici, mais il se trouve qu'il est très simple de les rendre non ambiguës : il suffit de préciser par ailleurs le sens de l'associativité et la priorité de chaque opérateur.

En *yacc*, cela se fait par des déclarations `%left` et `%right` qui spécifient le sens d'associativité des opérateurs, l'ordre de ces déclarations donnant leur priorité : à chaque nouvelle déclaration les opérateurs déclarés sont plus prioritaires que les précédents.

Ainsi, la grammaire précédente, présentée comme ci-après, n'est plus ambiguë. On a ajouté des déclarations indiquant que $+$, $-$, $*$ et / sont associatifs à gauche[38] et que la priorité de $*$ et / est supérieure à celle de $+$ et $-$.

```
%{
void yyerror(char *);
%}
%token nombre

%left '+' '-'
%left '*' '/'

%%
session     : session expression '='     { printf("résultat: %d\n", $2); }
            |
            ;
expression  : expression '+' expression  { $$ = $1 + $3; }
            | expression '-' expression  { $$ = $1 - $3; }
            | expression '*' expression  { $$ = $1 * $3; }
            | expression '/' expression  { $$ = $1 / $3; }
            | '(' expression ')'         { $$ = $2; }
            | nombre
            ;
%%
...
```

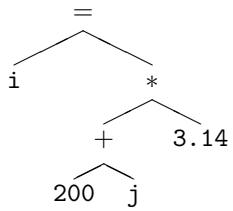
4 Analyse sémantique

Après l'analyse lexicale et l'analyse syntaxique, l'étape suivante dans la conception d'un compilateur est l'*analyse sémantique* dont la partie la plus visible est le *contrôle de type*. Des exemples de tâches liées au contrôle de type sont :

- construire et mémoriser des représentations des types définis par l'utilisateur, lorsque le langage le permet,
- traiter les déclarations de variables et fonctions et mémoriser les types qui leur sont appliqués,
- vérifier que toute variable référencée et toute fonction appelée ont bien été préalablement déclarées,
- vérifier que les paramètres des fonctions ont les types requis,
- contrôler les types des opérandes des opérations arithmétiques et en déduire le type du résultat,
- au besoin, insérer dans les expressions les conversion imposées par certaines règles de compatibilité,
- etc.

Pour fixer les idées, voici une situation typique où le contrôle de type joue. Imaginons qu'un programme, écrit en C, contient l'instruction « `i = (200 + j) * 3.14` ». L'analyseur syntaxique construit un *arbre abstrait* représentant cette expression, comme ceci (pour être tout à fait corrects, à la place de `i` et `j` nous aurions dû représenter des renvois à la table des symboles) :

[38] Dire qu'un opérateur $\otimes$ est associatif à gauche [resp. à droite] c'est dire que $a \otimes b \otimes c$ se lit $(a \otimes b) \otimes c$ [resp. $a \otimes (b \otimes c)$]. La question est importante, même pour des opérateurs simples : on tient à ce que $100 - 10 - 1$ vaille 89 et non 91 !

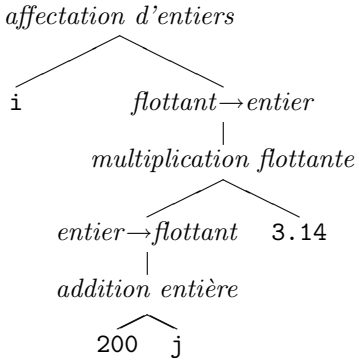


Dans de tels arbres, seules les feuilles (ici `i`, `200`, `j` et `3.14`) ont des types précis[39], tandis que les nœuds internes représentent des opérations abstraites, dont le type exact reste à préciser. Le travail sémantique à faire consiste à « remonter les types », depuis les feuilles vers la racine, rendant concrets les opérateurs et donnant un type précis aux sous-arbres.

Supposons par exemple que `i` et `j` aient été déclarées de type entier. L'analyse sémantique de l'arbre précédent permet d'en déduire, successivement :

- que le + est l'addition des entiers, puisque les deux opérandes sont entiers, et donc que le sous-arbre chapeauté par le + représente une valeur de type entier,
- que le ∗ est la multiplication des flottants, puisqu'un opérande est flottant[40], qu'il y a lieu de convertir l'autre opérande vers le type flottant, et que le sous-arbre chapeauté par ∗ représente un objet de type flottant,
- que l'affectation qui coiffe l'arbre tout entier consiste donc en l'affectation d'un flottant à un entier, et qu'il faut donc insérer une opération de troncation *flottant* → *entier* ; en C, il en découle que l'arbre tout entier représente une valeur du type entier.

En définitive, le contrôle de type transforme l'arbre précédent en quelque chose qui ressemble à ceci :



4.1 Représentation et reconnaissance des types

Une partie importante du travail sémantique qu'un compilateur fait sur un programme est

- pendant la compilation des déclarations, construire des représentations des types déclarés dans le programme,
- pendant la compilation des instructions, reconnaître les types des objets intervenant dans les expressions.

La principale difficulté de ce travail est la complexité des structures à construire et à manipuler. En effet, dans les langages modernes les types sont définis par des procédés récursifs qu'on peut composer à volonté. Par exemple, en C on peut avoir des entiers, des adresses (ou pointeurs) d'entiers, des fonctions rendant des adresses d'entiers, des adresses de fonctions rendant des adresses d'entiers, des tableaux d'adresses de fonctions rendant des adresses d'entiers, etc. Cela peut aller aussi loin qu'on veut, l'auteur d'un compilateur doit se donner le moyen de représenter ces structures de complexité quelconque.

Faute de temps, nous n'étudierons pas cet aspect des compilateurs dans le cadre de ce cours.

[39] Le type d'une constante est donné par une convention lexicale (exemple : 200 représente un entier, 3.14 un flottant), le type d'une variable ou le type rendu par une fonction est spécifié par la déclaration de la variable ou la fonction en question.

[40] Cela s'appelle la « règle du plus fort », elle est suivie par la plupart des langages : lorsque les opérandes d'une opération arithmétique ne sont pas de même type, celui dont le type est le « plus fort » (plus complexe, plus précis, plus étendu, etc.) tire à lui l'autre opérande, provoquant une conversion de type.

Ainsi, le compilateur que nous réaliserons à titre de projet ne traitera que les types *entier* et *tableau d'entiers*.

Pour les curieux, voici néanmoins une suggestion de structures de données pour la représentation des principaux types du langage C dans un compilateur qui serait lui-même[41] écrit en C :

```
typedef
    enum {
        tChar, tShort, tInt, tLong, tFloat, tDouble,
        tPointeur, tTableau, tFonction, tStructure
    } typePossible;

typedef
    struct listeDescripteursTypes {
        struct descripteurType *info;
        struct listeDescripteursTypes *suivant;
    } listeDescripteursTypes;

typedef
    struct descripteurType {
        typePossible classe;

        union {
            struct {
                struct descripteurType *typePointe;
            } casPointeur;

            struct {
                int nombreElements;
                struct descripteurType *typeElement;
            } casTableau;

            struct {
                listeDescripteursTypes *typesChamps;
            } casStructure;

            struct {
                listeDescripteursTypes *typesArguments;
                struct descripteurType *typeResultat;
            } casFonction;
        } attributs;
    } descripteurType;
```

Ainsi, un type se trouve représenté par une structure à deux champs : *classe*, dont la valeur est un code conventionnel qui indique de quelle sorte de type il s'agit, et *attributs*, qui donne les informations nécessaires pour achever de définir le type. *Attributs* est un champ polymorphe (en C, une *union*) dont la structure dépend de la valeur du champ *classe* :

– si la *classe* est celle d'un type primitif, le champ *attributs* est sans objet,

[41] Voici une question troublante qui finit par apparaître dans tous les cours de compilation : peut-on écrire le compilateur d'un langage en utilisant le langage qu'il s'agit de compiler ? Malgré l'apparent paradoxe, la chose est tout à fait possible. Il faut comprendre que la question n'est pas de savoir, par exemple, dans quel langage fut écrit le tout premier compilateur de C, si tant est qu'il y eut un jour un compilateur de C alors que la veille il n'y en avait pas – cela est le problème, peu intéressant, de l'œuf et de la poule. La question est plutôt de savoir si, de nos jours, on peut utiliser un compilateur (existant) de C pour réaliser un compilateur (nouveau) de C. Présentée comme cela, la chose est parfaitement raisonnable.

Là où elle redevient troublante : ce nouveau compilateur de C une fois écrit, on devra le valider. Pour cela, quel meilleur test que de lui donner à compiler... son propre texte source, puisqu'il est lui-même écrit en C ? Le résultat obtenu devra être un nouvel exécutable identique à celui du compilateur. On voit là un des intérêts qu'il y a à écrire le compilateur d'un langage dans le langage à compiler : on dispose *ipso facto* d'un formidable test de validation : au terme d'un certain processus de construction (on dit plutôt *bootstrap*) on doit posséder un compilateur C capable de compiler son propre texte source S et de donner un résultat $C(S)$ vérifiant $C(S) = C$.

– si le champ *classe* indique qu'il s'agit d'un type pointeur, alors le champ attributs pointe sur la description du type des objets pointés,
– si la valeur du champ *classe* est *tTableau* alors il faut deux attributs pour définir le type : *nombreElements*, le nombre de cases du tableau, et *typeElement*, qui pointe sur la description du type commun des éléments du tableau,
– s'il s'agit d'un type structure, l'attribut est l'adresse d'une liste chaînée dont chaque maillon contient un pointeur[42] sur un descripteur de type,
– enfin, si le champ *classe* indique qu'il s'agit d'un type fonction, alors le champ *attribut* se compose de l'adresse d'un descripteur de type (le type rendu par la fonction) et l'adresse d'une liste de types (les types des arguments de cette dernière).

On facilite l'allocation dynamique de telles structures en se donnant une fonction :

```
descripteurType *nouveau(typePossible classe) {
    descripteurType *res = malloc(sizeof(descripteurType));
    assert(res != NULL);
    res->classe = classe;
    return res;
}
```

Pour montrer le fonctionnement de cette structure de données, voici un exemple purement démonstratif, la construction « à la main » du descripteur correspondant au type de la variable déclarée, en C, comme suit :

```
struct {
    char *lexeme;
    int uniteLexicale;
    } motRes[N];
```

La variable est `motRes` (nous l'avons utilisée à la section 2.2.2), elle est déclarée comme un tableau de `N` éléments qui sont des structures à deux champs : un pointeur sur un caractère et un entier. Voici le code qui construit un tel descripteur (pointé, à la fin de la construction, par la variable `tmp2`) :

```
...
listeDescripteursTypes *pCour, *pSuiv;
descripteurType *pTmp1, *pTmp2;

        /* maillon de liste décrivant le type entier */
pCour = malloc(sizeof(listeDescripteursTypes));
pCour->info = nouveau(tInt);
pCour->suiv = NULL;

        /* maillon de liste décrivant le type pointeur sur caractère */
pSuiv = pCour;
pCour = malloc(sizeof(listeDescripteursTypes));
pCour->info = nouveau(tPointeur);
pCour->info->attributs.casPointeur.typePointe = nouveau(tChar);
pCour->suiv = pSuiv;

        /* pTmp1 va pointer la description de la structure */
pTmp1 = nouveau(tStructure);
pTmp1->attributs.casStructure.typesChamps = pCour;

        /* pTmp2 va pointer la description du type tableau */
pTmp2 = nouveau(tTableau);
pTmp2->attributs.casTableau.nombreElements = N;
pTmp2->attributs.casTableau.typeElement = pTmp1;
...
```

[42] Il aurait aussi été viable de faire que chaque maillon de la liste chaînée contienne un descripteur de type, au lieu d'un pointeur sur un tel descripteur. Apparemment plus lourde à gérer, la solution adoptée ici se révèle à l'usage la plus souple.

Dans le même ordre d'idées, voici la construction manuelle du descripteur du type « matrice de $NL \times NC$ flottants » ou, plus précisément, « tableau de NL éléments qui sont des tableaux de NC flottants » (en C, cela s'écrit : `float matrice[NL][NC] ;`). A la fin de la construction le descripteur cherché est pointé par `pTmp2` :

```
...
        /* description d'une ligne, tableau de NC flottants: */
pTmp1 = nouveau(tTableau);
pTmp1->attributs.casTableau.nombreElements = NC;
pTmp1->attributs.casTableau.typeElement = nouveau(tFloat);

        /* description d'une matrice, tableau de NL lignes: */
pTmp2 = nouveau(tTableau);
pTmp2->attributs.casTableau.nombreElements = NL;
pTmp2->attributs.casTableau.typeElement = pTmp1;
...
```

Enfin, pour donner encore un exemple de manipulation de ces structures de données, voici un utilitaire fondamental dans les systèmes de contrôle de type : la fonction booléenne qui fournit la réponse à la question « deux descripteurs donnés décrivent-ils des types identiques ? » :

```
int memeType(descripteurType *a, descripteurType *b) {

    if (a->classe != b->classe)
        return 0;

    switch (a->classe) {
        case tPointeur:
            return memeType(a->attributs.casPointeur.typePointe,
                    b->attributs.casPointeur.typePointe);
        case tTableau:
            return a->attributs.casTableau.nombreElements ==
                    b->attributs.casTableau.nombreElements
                && memeType(a->attributs.casTableau.typeElement,
                            b->attributs.casTableau.typeElement);
        case tStructure:
            return memeListeTypes(a->attributs.casStructure.typesChamps,
                    b->attributs.casStructure.typesChamps);
        case tFonction:
            return memeType(a->attributs.casFonction.typeResultat,
                    b->attributs.casFonction.typeResultat)
                && memeListeTypes(a->attributs.casFonction.typesArguments,
                            b->attributs.casFonction.typesArguments);
        default:
            return 1;
    }
}

int memeListeTypes(listeDescripteursTypes *a, listeDescripteursTypes *b) {
    while (a != NULL && b != NULL) {
        if ( ! memeType(a->info, b->info))
            return 0;
        a = a->suiv;
        b = b->suiv;
    }
    return a == NULL && b == NULL;
}
```

4.2 Dictionnaires (tables de symboles)

Dans les langages de programmation modernes, les variables et les fonctions doivent être déclarées avant d'être utilisées dans les instructions. Quel que soit le degré de complexité des types supportés par notre compilateur, celui-ci devra gérer une *table de symboles*, appelée aussi *dictionnaire*, dans laquelle se trouveront les identificateurs couramment déclarés, chacun associé à certains attributs, comme son *type*, son *adresse*[43] et d'autres informations, cf. figure 10.

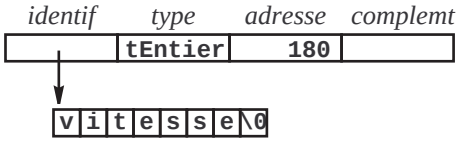


FIG. 10 – Une entrée dans le dictionnaire

Nous étudions pour commencer le cahier des charges du dictionnaire, c'est-à-dire les services que le compilateur en attend, puis, dans les sections suivantes, diverses implémentations possibles.

Grosso modo le dictionnaire fonctionne ainsi :
- quand le compilateur trouve un identificateur dans une déclaration, il le cherche dans le dictionnaire en espérant ne pas le trouver (sinon c'est l'erreur « identificateur déjà déclaré »), puis il l'ajoute au dictionnaire avec le type que la déclaration spécifie,
- quand le compilateur trouve un identificateur dans la partie exécutable[44] d'un programme, il le cherche dans le dictionnaire avec l'espoir de le trouver (sinon c'est l'erreur « identificateur non déclaré »), ensuite il utilise les informations que le dictionnaire associe à l'identificateur.

Nous allons voir que la question est un peu plus compliquée que cela.

4.2.1 Dictionnaire global & dictionnaire local

Dans les langages qui nous intéressent, un programme est essentiellement une collection de fonctions, entre lesquelles se trouvent des déclarations de variables. A l'intérieur des fonctions se trouvent également des déclarations de variables.

Les variables déclarées entre les fonctions et les fonctions elles-mêmes sont des *objets globaux*. Un objet global est visible[45] depuis sa déclaration jusqu'à la fin du texte source, sauf aux endroits où un objet local de même nom le masque, voir ci-après.

Les variables déclarées à l'intérieur des fonctions sont des *objets locaux*. Un objet local est visible dans la fonction où il est déclaré, depuis sa déclaration jusqu'à la fin de cette fonction ; il n'est pas visible depuis les autres fonctions. En tout point où il est visible, un objet local *masque*[46] tout éventuel objet global qui aurait le même nom.

En définitive, quand le compilateur se trouve dans[47] une fonction il faut posséder deux dictionnaires : un *dictionnaire global*, contenant les noms des objets globaux couramment déclarés, et un *dictionnaire local* dans lequel se trouvent les noms des objets locaux couramment déclarés (qui, parfois, masquent des objets dont les noms se trouvant dans le dictionnaire global).

Dans ces conditions, l'utilisation des dictionnaires que fait le compilateur se précise :

[43] La question des adresses des objets qui se trouvent dans les programmes sera étudiée en détail à la section 5.1.1.

[44] Dans les langages comme C, Java, etc., la *partie exécutable* des programmes est l'ensemble des corps des fonctions dont le programme se compose. En Pascal il faut ajouter à cela le corps du programme.

[45] On dit qu'un objet o ayant le nom n est *visible* en un point d'un programme si une occurrence de n en ce point est comprise comme désignant o. Cela ne préjuge en rien de la correction ou de la légalité de l'emploi de n en ce point.

[46] Par *masquage* d'un objet o par un objet o' de même nom n on veut dire que o n'est pas altéré ni détruit, mais devient inaccessible car, dans la portion de programme où le masquage a lieu, n désigne o', non o.

[47] Le compilateur lit le programme à compiler séquentiellement, du début vers la fin. A tout instant il en est à un certain endroit du texte source, correspondant à la position de l'unité lexicale courante ; quand la compilation progresse, l'unité lexicale avance. Tout cela justifie un langage imagé, que nous allons employer, avec des expressions comme « le compilateur *entre* dans la partie exécutable » ou « le compilateur *entre* (ou *sort*) d'une fonction », etc.

– Lorsque le compilateur traite la déclaration d'un identificateur i qui se trouve à l'intérieur d'une fonction, i est recherché dans le dictionnaire local *exclusivement* ; normalement, il ne s'y trouve pas (sinon, « erreur : identificateur déjà déclaré »). Suite à cette déclaration, i est ajouté au dictionnaire local.
Il n'y a strictement aucun intérêt à savoir si i figure à ce moment-là dans le dictionnaire global.
– Lorsque le compilateur traite la déclaration d'un identificateur i en dehors de toute fonction, i est recherché dans le dictionnaire global, qui est le seul dictionnaire existant en ce point ; normalement, il ne s'y trouve pas (sinon, « erreur : identificateur déjà déclaré »). Suite à cette déclaration, i est ajouté au dictionnaire global.
– Lorsque le compilateur compile une instruction exécutable, forcément à l'intérieur d'une fonction, chaque identificateur i rencontré est recherché d'abord dans le dictionnaire local ; s'il ne s'y trouve pas, il est recherché ensuite dans le dictionnaire global (si les deux recherches échouent, « erreur : identificateur non déclaré »). En procédant ainsi on assure le masquage des objets globaux par les objets locaux.
– Lorsque le compilateur quitte une fonction, le dictionnaire local en cours d'utilisation est détruit, puisque les objets locaux ne sont pas visibles à l'extérieur de la fonction. Un dictionnaire local nouveau, vide, est créé lorsque le compilateur entre dans une fonction.

Notez ceci : tout au long d'une compilation le dictionnaire global ne diminue jamais. A l'intérieur d'une fonction il n'augmente pas ; le dictionnaire global n'augmente que lorsque le dictionnaire local n'existe pas.

4.2.2 Tableau à accès séquentiel

L'implémentation la plus simple des dictionnaires consiste en un tableau dans lequel les identificateurs sont placés dans l'ordre où leurs déclarations ont été trouvées dans le texte source. Dans ce tableau, les recherches sont séquentielles. Voyez la figure 11 : lorsqu'il existe, le dictionnaire local se trouve au-dessus du dictionnaire global (en supposant que le tableau grandit du bas vers le haut).

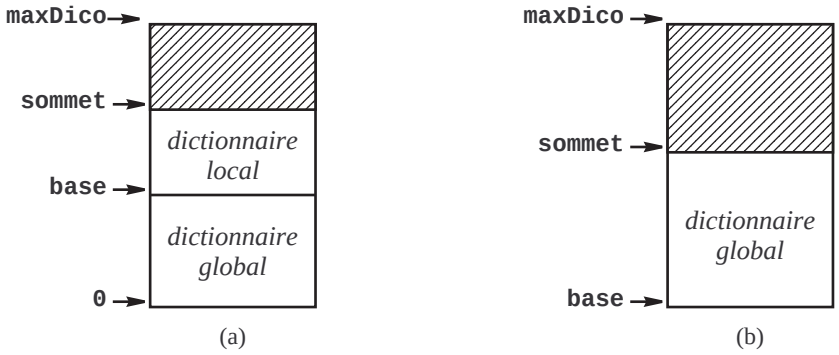


FIG. 11 – Dictionnaires, quand on est à l'intérieur (a) et à l'extérieur (b) des fonctions

Trois variables sont essentielles dans la gestion du dictionnaire :

`maxDico` est le nombre maximum d'entrées possibles (à ce propos, voir « Augmentation de la taille du dictionnaire » un peu plus loin),
`sommet` est le nombre d'entrées valides dans le dictionnaire ; on doit avoir `sommet` $\leq$ `maxDico`,
`base` est le premier élément du dictionnaire du dessus (c'est-à-dire le dictionnaire local quand il y en a deux, le dictionnaire global quand il n'y en a qu'un).

Avec tout cela, la manipulation du dictionnaire devient très simple. Les opérations nécessaires sont :

1. Recherche d'un identificateur pendant le traitement d'une déclaration (que ce soit à l'intérieur d'une fonction ou à l'extérieur de toute fonction) : rechercher l'identificateur dans la portion de tableau comprise entre les bornes $sommet - 1$ et $base$,
2. Recherche d'un identificateur pendant le traitement d'une expression exécutable : rechercher l'identificateur en parcourant *dans le sens des indices décroissants*[48] la portion de tableau comprise entre les bornes $sommet - 1$ et 0,

[48] En parcourant le tableau du haut vers le bas on assure le masquage d'un objet global par un objet local de même nom.

3. Ajout d'une entrée dans le dictionnaire (que ce soit à l'intérieur d'une fonction ou à l'extérieur de toute fonction) : après avoir vérifié que $sommet < maxDico$, placer la nouvelle entrée à la position *sommet*, puis faire $sommet \leftarrow sommet + 1$,
4. Creation d'un dictionnaire local, au moment de l'entrée dans une fonction : faire $base \leftarrow sommet$,
5. Destruction du dictionnaire local, à la sortie d'une fonction : faire $sommet \leftarrow base$ puis $base \leftarrow 0$.

Augmentation de la taille du dictionnaire. Une question technique assez agaçante qu'il faut régler lors de l'implémentation d'un dictionnaire par un tableau est le choix de la taille à donner à ce tableau, étant entendu qu'on ne connaît pas à l'avance la grosseur (en nombre de déclarations) des programmes que notre compilateur devra traiter.

La bibliothèque C offre un moyen pratique pour résoudre ce problème, la fonction *realloc* qui permet d'augmenter la taille d'un espace alloué dynamiquement tout en préservant le contenu de cet espace. Voici, à titre d'exemple, la déclaration et les fonctions de gestion d'un dictionnaire réalisé dans un tableau ayant au départ la place pour 50 éléments ; chaque fois que la place manque, le tableau est agrandi d'autant qu'il faut pour loger 25 nouveaux éléments :

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef
    struct {
        char *identif;
        int type;
        int adresse;
        int complement;
    } ENTREE_DICO;

#define TAILLE_INITIALE_DICO   50
#define INCREMENT_TAILLE_DICO  25

ENTREE_DICO *dico;
int maxDico, sommet, base;

void creerDico(void) {
    maxDico = TAILLE_INITIALE_DICO;
    dico = malloc(maxDico * sizeof(ENTREE_DICO));
    if (dico == NULL)
        erreurFatale("Erreur interne (pas assez de mémoire)");
    sommet = base = 0;
}

void agrandirDico(void) {
    maxDico = maxDico + INCREMENT_TAILLE_DICO;
    dico = realloc(dico, maxDico);
    if (dico == NULL)
        erreurFatale("Erreur interne (pas assez de mémoire)");
}

void erreurFatale(char *message) {
    fprintf(stderr, "%s\n", message);
    exit(-1);
}
```

Pour montrer une utilisation de tout cela, voici la fonction qui ajoute une entrée au dictionnaire :

```
void ajouterEntree(char *identif, int type, int adresse, int complement) {
    if (sommet >= maxDico)
```

```
        agrandirDico();

    dico[sommet].identif = malloc(strlen(identif) + 1);
    if (dico[sommet].identif == NULL)
        erreurFatale("Erreur interne (pas assez de mémoire)");
    strcpy(dico[sommet].identif, identif);
    dico[sommet].type = type;
    dico[sommet].adresse = adresse;
    dico[sommet].complement = complement;
    sommet++;
}
```

4.2.3 Tableau trié et recherche dichotomique

L'implémentation des dictionnaires expliquée à la section précédente est facile à mettre en œuvre et suffisante pour des applications simples, mais pas très efficace (la complexité des recherches est, en moyenne, de l'ordre de $\frac{n}{2}$, soit $O(n)$; les insertions se font en temps constant). Dans la pratique on recherche des implémentations plus efficaces, car un compilateur passe beaucoup de temps[49] à rechercher des identificateurs dans les dictionnaires.

Une première amélioration des dictionnaires consiste à maintenir des tableaux ordonnés, permettant des recherches par dichotomie (la complexité d'une recherche devient ainsi $O(\log_2 n)$, c'est beaucoup mieux). La figure 11 est toujours valable, mais maintenant il faut imaginer que les éléments d'indices allant de *base* à *sommet* − 1 et, lorsqu'il y a lieu, ceux d'indices allant de 0 à *base* − 1, sont placés en ordre croissant des identificateurs.

Dans un tel contexte, voici la fonction *existe*, qui effectue la recherche de l'identificateur représenté par *ident* dans la partie du tableau, supposé ordonné, comprise entre les indices *inf* et *sup*, bornes incluses. Le résultat de la fonction (1 ou 0, interprétés comme *vrai* ou *faux*) est la réponse à la question « l'élément cherché se trouve-t-il dans le tableau ? ». En outre, au retour de la fonction, la variable pointée par *ptrPosition* contient la position de l'élément recherché, c'est-à-dire :

- si l'identificateur est dans le tableau, l'indice de l'entrée correspondante,
- si l'identificateur ne se trouve pas dans le tableau, l'indice auquel il faudrait insérer, les cas échéant, une entrée concernant cet identificateur.

```
int existe(char *identif, int inf, int sup, int *ptrPosition) {
    int i, j, k;

    i = inf;
    j = sup;
    while (i <= j) {              /* invariant: i <= position <= j + 1 */
        k = (i + j) / 2;
        if (strcmp(dico[k].identif, identif) < 0)
            i = k + 1;
        else
            j = k - 1;
    }
                                  /* ici, de plus, i > j, soit i = j + 1 */
    *ptrPosition = i;
    return i <= sup && strcmp(dico[i].identif, identif) == 0;
}
```

Voici la fonction qui ajoute une entrée au dictionnaire :

```
void ajouterEntree(int position, char *identif, int type, int adresse, int complt) {
    int i;
    if (sommet >= maxDico)
        agrandirDico();
```

[49] On estime que plus de 50% du temps d'une compilation est dépensé en recherches dans les dictionnaires.

```
        for (i = sommet - 1; i >= position; i--)
            dico[i + 1] = dico[i];
        sommet++;

        dico[position].identif = malloc(strlen(identif) + 1);
        if (dico[position].identif == NULL)
            erreurFatale("Erreur interne (pas assez de mémoire)");
        strcpy(dico[position].identif, identif);
        dico[position].type = type;
        dico[position].adresse = adresse;
        dico[position].complement = complt;
    }
```

La fonction *ajouterEntree* utilise un paramètre *position* dont la valeur provient de la fonction *existe*. Pour fixer les idées, voici la fonction qui traite la déclaration d'un objet local :

```
        ...
        int placement;
        ...
        if (existe(lexeme, base, sommet - 1, &placement))
            erreurFatale("identificateur déjà déclaré");
        else {
            ici se place l'obtention des informations type, adresse, complement, etc.
            ajouterEntree(placement, lexeme, type, adresse, complement);
        }
        ...
```

Nous constatons que l'utilisation de tableaux triés permet d'optimiser la recherche, dont la complexité passe de $O(n)$ à $O(\log_2 n)$, mais pénalise les insertions, dont la complexité devient $O(n)$, puisqu'à chaque insertion il faut pousser d'un cran la moitié (en moyenne) des éléments du dictionnaire. Or, pendant la compilation d'un programme il y a beaucoup d'insertions et on ne peut pas négliger *a priori* le poids des insertions dans le calcul du coût de la gestion des identificateurs.

Il y a cependant une tâche, qui n'est pas la gestion du dictionnaire mais lui est proche, où on peut sans réserve employer un tableau ordonné, c'est la gestion d'une table de mots réservés, comme celle de la section 2.2.2. En effet, le compilateur, ou plus précisément l'analyseur lexical, fait de nombreuses recherches dans cette table qui ne subit jamais la moindre insertion.

4.2.4 Arbre binaire de recherche

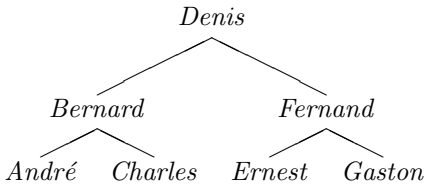
Cette section suppose la connaissance de la structure de données arbre binaire.

Les arbres binaires de recherche ont les avantages des tables ordonnées, pour ce qui est de l'efficacité de la recherche, sans leurs inconvénients puisque, étant des structures chaînées, l'insertion d'un élément n'oblige pas à pousser ceux qui d'un point de vue logique se placent après lui.

Un *arbre binaire de recherche*, ou *ABR*, est un arbre binaire étiqueté par des valeurs appartenant à un ensemble ordonné, vérifiant la propriété suivante : pour tout nœud p de l'arbre

- pour tout nœud q appartenant au sous-arbre gauche de p on a $q{\rightarrow}info \leq p{\rightarrow}info$,
- pour tout nœud r appartenant au sous-arbre droit de p on a $r{\rightarrow}info \geq p{\rightarrow}info$.

Voici, par exemple, l'*ABR* obtenu avec les « identificateurs » *Denis*, *Fernand*, *Bernard*, *André*, *Gaston*, *Ernest* et *Charles*, ajoutés à l'arbre successivement et dans cet ordre[50] :



[50]Le nombre d'éléments et, surtout, l'ordre d'insertion font que cet *ABR* est parfaitement équilibré. En pratique, les choses ne se passent pas aussi bien.

Techniquement, un tel arbre serait réalisé dans un programme comme le montre la figure 12.

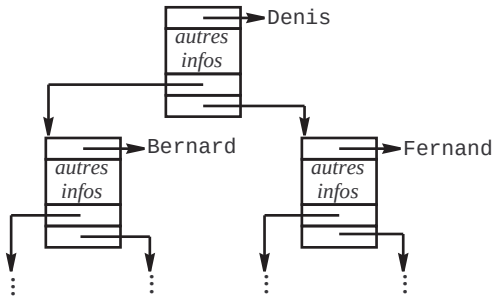


FIG. 12 – Réalisation effective des maillons d'un *ABR*

Pour réaliser les dictionnaires par des *ABR* il faudra donc se donner les déclarations :

```
typedef
    struct noeud {
        ENTREE_DICO info;
        struct noeud *gauche, *droite;
    } NOEUD;

NOEUD *dicoGlobal = NULL, *dicoLocal = NULL;
```

Voici la fonction qui recherche le nœud correspondant à un identificateur donné. Elle rend l'adresse du nœud cherché, ou NULL en cas d'échec de la recherche :

```
NOEUD *rechercher(char *identif, NOEUD *ptr) {
    while (ptr != NULL) {
        int c = strcmp(identif, ptr->info.identif);
        if (c == 0)
            return ptr;
        else
            if (c < 0)
                ptr = ptr->gauche;
            else
                ptr = ptr->droite;
    }
    return NULL;
}
```

Cette fonction est utilisée pour rechercher des identificateurs apparaissant dans les parties exécutables des fonctions. Elle sera donc appelée de la manière suivante :

```
...
p = rechercher(lexeme, dicoLocal);
if (p == NULL) {
    p = recherche(lexeme, dicoGlobal);
    if (p == NULL)
        erreurFatale("identificateur non déclaré");
    }
```

exploitation des informations du nœud pointé par `p`

```
...
```

Pendant la compilation des déclarations, les recherches se font avec la fonction suivante, qui effectue la recherche et, dans la foulée, l'ajout d'un nouveau nœud. Dans cette fonction, la rencontre d'un nœud associé à l'identificateur qu'on cherche est considérée comme une erreur grave. La fonction rend l'adresse du nœud nouvelle créé :

```
NOEUD *insertion(NOEUD **adrDico, char *identif, int type, int adresse, int complt) {
    NOEUD *ptr;
    if (*adrDico == NULL)
        return *adrDico = nouveauNoeud(identif, type, adresse, complt);

    ptr = *adrDico;
    for (;;) {
        int c = strcmp(identif, ptr->info.identif);
        if (c == 0)
            erreurFatale("identificateur deja déclaré");
        if (c < 0)
            if (ptr->gauche != NULL)
                ptr = ptr->gauche;
            else
                return ptr->gauche = nouveauNoeud(identif, type, adresse, complt);
        else
            if (ptr->droite != NULL)
                ptr = ptr->droite;
            else
                return ptr->droite = nouveauNoeud(identif, type, adresse, complt);
    }
}
```

Exemple d'appel (cas des déclarations locales) :

```
...
p = insertion( &dicoLocal, lexeme, leType, lAdresse, leComplement);
...
```

N.B. Dans la fonction *insertion*, le pointeur de la racine du dictionnaire dans lequel il faut faire l'insertion est passé *par adresse*, c'est pourquoi il y a deux ∗∗ dans la déclaration `NOEUD **adrDico`. Cela ne sert qu'à couvrir le cas de la *première* insertion, lorsque le dictionnaire est vide : le pointeur pointé par *adrDico* (en pratique il s'agit soit de *dicoLocal* soit de *dicoGlobal*) vaut NULL et doit changer de valeur. C'est beaucoup de travail pour pas grand-chose, on l'éviterait en décidant que les dictionnaires ne sont jamais vides (il suffit de leur créer d'office un « nœud bidon » sans signification).

Restitution de l'espace alloué. L'implémentation d'un dictionnaire par un *ABR* possède l'efficacité de la recherche dichotomique, car à chaque comparaison on divise par deux la taille de l'ensemble susceptible de contenir l'élément cherché, sans ses inconvénients, puisque le temps nécessaire pour faire une insertion dans un *ABR* est négligeable. Hélas, cette méthode a deux défauts :

- la recherche n'est dichotomique que si l'arbre est équilibré, ce qui ne peut être supposé que si les identificateurs sont très nombreux et uniformément répartis,
- la destruction d'un dictionnaire est une opération beaucoup plus complexe que dans les méthodes qui utilisent un tableau.

La destruction d'un dictionnaire, en l'occurrence le dictionnaire local, doit se faire chaque fois que le compilateur sort d'une fonction. Cela peut se programmer de la manière suivante :

```
void liberer(NOEUD *dico) {
    if (dico != NULL) {
        liberer(dico->gauche);
        liberer(dico->droite);
        free(dico);
    }
}
```

Comme on le voit, pour rendre l'espace occupé par un *ABR* il faut le parcourir entièrement (alors que dans le cas d'un tableau la modification d'un index suffit).

Il y a un moyen de rendre beaucoup plus simple la libération de l'espace occupé par un arbre. Cela consiste à écrire sa propre fonction d'allocation, qu'on utilise à la place *malloc*, et qui alloue un espace dont on maîtrise la remise à zéro. Par exemple :

```
#define MAX_ESPACE 1000

NOEUD espace[MAX_ESPACE];
int niveauAllocation = 0;

NOEUD *monAlloc(void) {
    if (niveauAllocation >= MAX_ESPACE)
        return NULL;
    else
        return &espace[niveauAllocation++];
}

void toutLiberer(void) {
    niveauAllocation = 0;
}
```

4.2.5 Adressage dispersé

Une dernière technique de gestion d'une table de symboles mérite d'être mentionnée ici, car elle est très utilisée dans les compilateurs réels. Cela s'appelle *adressage dispersé*, ou *hash-code*[51]. Le principe en est assez simple : au lieu de rechercher la position de l'identificateur dans la table, on obtient cette position par un calcul sur les caractères de l'identificateur ; si on s'en tient à des opérations simples, un calcul est certainement plus rapide qu'une recherche.

Soit I l'ensemble des identificateurs existant dans un programme, N la taille de la table d'identificateurs. L'idéal serait de posséder une fonction $h : I \to [0, N[$ qui serait

- rapide, facile à calculer,
- injective, c'est-à-dire qui à deux identificateurs distincts ferait correspondre deux valeurs distinctes.

On ne dispose généralement pas d'une telle fonction car l'ensemble I des identificateurs présents dans le programme n'est pas connu *a priori*. De plus, la taille de la table n'est souvent pas suffisante pour permettre l'injectivité (qui requiert $N \geq |I|$).

On se contente donc d'une fonction h prenant, sur l'ensemble des identificateurs possibles, des valeurs *uniformément réparties* sur l'intervalle $[0, N[$. C'est-à-dire que h n'est pas injective, mais

- si $N \geq |I|$, on espère que les couples d'identificateurs i_1, i_2 tels que $i_1 \neq i_2$ et $h(i_1) = h(i_2)$ (on appelle cela une *collision*) sont peu nombreux,
- si $N < |I|$, les collisions sont inévitables. Dans ce cas on souhaite qu'elles soient également réparties : pour chaque $j \in [0, N[$ le nombre de $i \in I$ tels que $h(i) = j$ est à peu près le même, c'est-à-dire $\frac{|I|}{N}$. Il est facile de voir pourquoi : h étant la fonction qui « place » les identificateurs dans la table, il s'agit d'éviter que ces derniers s'amoncellent à certains endroits de la table, tandis qu'à d'autres endroits cette dernière est peu remplie, voire présente des cases vides.

Il est difficile de dire ce qu'est une bonne fonction de hachage. La littérature spécialisée rapporte de nombreuses recettes, mais il n'y a probablement pas de solution universelle, car une fonction de hachage n'est bonne que par rapport à un ensemble d'identificateurs donné. Parmi les conseils qu'on trouve le plus souvent :

- prenez N premier (une des recettes les plus données, mais plus personne ne se donne la peine d'en rappeler la justification),
- utilisez des fonctions qui font intervenir *tous* les caractères des identificateurs ; en effet, dans les programmes on rencontre souvent des « grappes » de noms, par exemple : *poids*, *poids1*, *poidsTotal*, *poids_maxi*, etc. ; une fonction qui ne ferait intervenir que les cinq premiers caractères ne serait pas très bonne ici,
- écrivez des fonctions qui donnent comme résultat de grandes valeurs ; lorsque ces valeurs sont ramenées à l'intervalle $[0, N[$, par exemple par une opération de modulo, les éventuels défauts (dissymétries, accumulations, etc.) de la fonction initiale ont tendance à disparaître.

Une fonction assez souvent utilisée consiste à considérer les caractères d'un identificateur comme les coefficients d'un polynôme $P(X)$ dont on calcule la valeur pour un X donné (ou, ce qui revient au même, à voir un

[51] La technique expliquée ici est celle dite adressage dispersé *ouvert*. Il en existe une autre, l'adressage dispersé *fermé*, dans laquelle toute l'information se trouve dans le tableau adressé par la fonction de hachage (il n'y a pas de listes chaînées associés aux cases du tableau).

identificateur comme l'écriture d'un nombre dans une certaine base). En C, cela donne la fonction :

```
int hash(char *ident, int N) {
    const int X = 23;            /* why not? */
    int r = 0;
    while (*ident != '\0')
        r = X * r + *(ident++);
    return r % N;
}
```

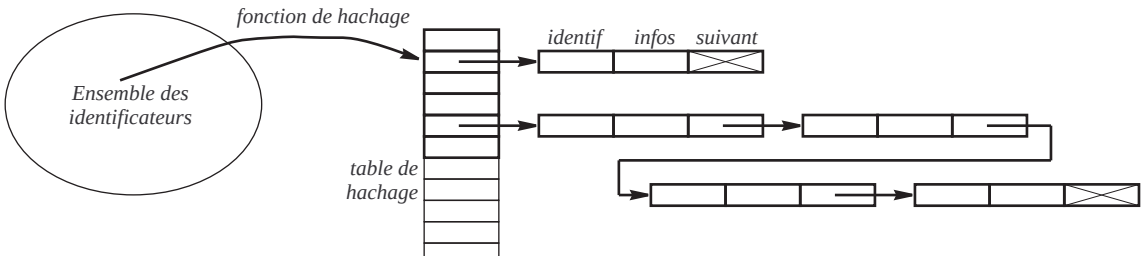


FIG. 13 – Adressage dispersé ouvert

Là où les méthodes d'adressage dispersé différent entre elles c'est dans la manière de gérer les collisions. Dans le cas de l'adressage dispersé *ouvert*, la table qu'on adresse à travers la fonction de hachage n'est pas une table d'identificateurs, mais une table de listes chaînées dont les maillons portent des identificateurs (voyez la figure 13). Si on note T cette table, alors T_j est le pointeur de tête de la liste chaînée dans laquelle sont les identificateurs i tels que $h(i) = j$.

Vu de cette manière, l'adressage dispersé ouvert apparaît comme une méthode de *partitionnement* de l'ensemble des identificateurs. Chaque liste chaînée est un compartiment de la partition. Si la fonction h est bien faite, les compartiments ont à peu près la même taille. L'efficacité de la méthode provient alors du fait qu'au lieu de faire une recherche dans une structure de taille $|I|$ on fait un calcul et une recherche dans une structure de taille $\frac{|I|}{N}$.

Voici la fonction de recherche :

```
typedef
    struct maillon {
        char *identif;
        autres informations
        struct maillon *suivant;
    } MAILLON;

#define N 101
MAILLON *table[N];

MAILLON *recherche(char *identif) {
    MAILLON *p;
    for (p = table[hash(identif, N)]; p != NULL; p = p->suivant)
        if (strcmp(identif, p->identif) == 0)
            return p;
    return NULL;
}
```

et voici la fonction qui se charge de l'insertion d'un identificateur (supposé absent) :

```
MAILLON *insertion(char *identif) {
    int h = hash(identif, N);
```

```
        return table[h] = nouveauMaillon(identif, table[h]);
    }
```

avec une fonction *nouveauMaillon* définie comme ceci :

```
MAILLON *nouveauMaillon(char *identif, MAILLON *suivant) {
    MAILLON *r = malloc(sizeof(MAILLON));
    if (r == NULL)
        erreurFatale("Erreur interne (problème d'espace)");
    r->identif = malloc(strlen(identif) + 1);
    if (r->identif == NULL)
        erreurFatale("Erreur interne (problème d'espace)");
    strcpy(r->identif, identif);
    r->suivant = suivant;
    return r;
}
```

5 Production de code

Nous nous intéressons ici à la dernière phase de notre compilateur, la production de code. Dans ce but, nous présentons certains aspects de l'architecture des machines (registres et mémoire, adresses, pile d'exécution, compilation séparée et édition de liens, etc.) et en même temps nous introduisons une machine virtuelle, la machine Mach 1, pour laquelle notre compilateur produit du code et dont nous écrirons un simulateur.

Faute de temps, nous faisons l'impasse sur certains aspects importants (et difficiles) de la génération de code, et notamment sur les algorithmes pour l'optimisation du code et pour l'allocation des registres.

5.1 Les objets et leurs adresses

5.1.1 Classes d'objets

Dans les langages qui nous intéressent les programmes manipulent trois classes d'objets :

1. Les *objets statiques* existent pendant toute la durée de l'exécution d'un programme ; on peut considérer que l'espace qu'ils occupent est alloué par le compilateur pendant la compilation[52].

   Les objets statiques sont les *fonctions*, les *constantes* et les *variables globales*. Ils sont garnis de valeurs initiales : pour une fonction, son code, pour une constante, sa valeur et pour une variable globale, une valeur initiale explicitée par l'auteur du programme (dans les langages qui le permettent) ou bien une valeur initiale implicite, souvent zéro.

   Les objets statiques peuvent être en *lecture seule* ou en *lecture-écriture*[53]. Les fonctions et les constantes sont des objets statiques en lecture seule. Les variables globales sont des objets statiques en lecture-écriture.

   On appelle *espace statique* l'espace mémoire dans lequel sont logés les objets statiques. Il est généralement constitué de deux zones : la *zone du code*, où sont les fonctions et les constantes, et l'*espace global*, où sont les variables globales.

   L'*adresse* d'un objet statique est un nombre entier qui indique la première (parfois l'unique) cellule de la mémoire occupée par l'objet. Elle est presque toujours exprimée comme un décalage[54] par rapport au début de la zone contenant l'objet en question.

2. Les *objets automatiques* sont les variables locales des fonctions, ce qui comprend :

[52] En réalité, le compilateur n'alloue pas la mémoire pendant la compilation ; au lieu de cela, il en produit une certaine représentation dans le fichier objet, et c'est un outil appelé *chargeur* qui, après traitement du fichier objet par l'*éditeur de lien*, installe les objets statiques dans la mémoire. Mais, pour ce qui nous occupe ici, cela revient au même.

[53] Lorsque le système le permet, les objets en lecture seule sont logés dans des zones de la mémoire dans lesquelles les tentatives d'écriture, c'est-à-dire de modification de valeurs, sont détectées et signalées ; les petits systèmes n'offrent pas cette sécurité (qui limite les dégâts lors des utilisations inadéquates des pointeurs et des indices des tableaux).

[54] Le mot *décalage* (les anglophones disent *offset*) fait référence à une méthode d'adressage employée très souvent : une entité est repérée par un couple (*base*,*décalage*), où *base* est une adresse connue et *décalage* un nombre entier qu'il faut ajouter à *base* pour obtenir l'adresse voulue. L'accès `t[i]` au i-ème élément d'un tableau t en est un exemple typique, dans lequel t (l'adresse de `t[0]`) est la base et i le décalage.

– les variables déclarées à l'intérieur des fonctions,
– les arguments formels de ces dernières.

Ces variables occupent un espace qui n'existe pas pendant toute la durée de l'exécution du programme, mais uniquement lorsqu'il est utile. Plus précisément, l'activation d'une fonction commence par l'allocation d'un espace, appelé *espace local de la fonction*, de taille suffisante pour contenir ses arguments et ses variables locales (plus un petit nombre d'informations techniques additionnelles expliquées en détail à la section 5.2.4).

Un espace local nouveau est alloué chaque fois qu'une fonction est appelée, même si cette fonction était déjà active et donc qu'un espace local pour elle existait déjà (c'est le cas d'une fonction qui s'appelle elle-même, directement ou indirectement). Lorsque l'activation d'une fonction se termine son espace local est détruit.

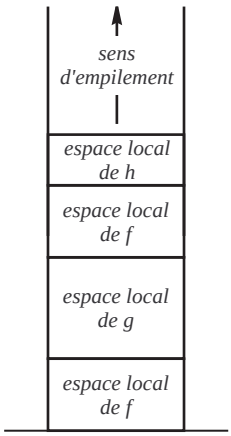


FIG. 14 – Empilement d'espaces locaux (f a appelé g qui a appelé f qui a appelé h)

La propriété suivante est importante : chaque fois qu'une fonction se termine, la fonction qui se termine est *la plus récemment activée* de celles qui ont été commencées et ne sont pas encore terminées. Il en découle que les espaces locaux des fonctions peuvent être alloués dans une mémoire gérée comme une pile (voyez la figure 14) : lorsqu'une fonction est activée, son espace local est créé au-dessus des espaces locaux des fonctions actives (i.e. commencées et non terminées) à ce moment-là. Lorsqu'une fonction se termine, son espace local est celui qui se trouve au sommet de la pile et il suffit de le dépiler pour avoir au sommet l'espace local de la fonction qui va reprendre le contrôle.

3. Les *objets dynamiques* sont alloués lorsque le programme le demande explicitement (par exemple à travers la fonction *malloc* de C ou l'opérateur *new* de Java et C++). Si leur destruction n'est pas explicitement demandée (fonction *free* de C, opérateur *delete* de C++) ces objets existent jusqu'à la terminaison du programme[55].

   Les objets dynamiques ne peuvent pas être logés dans les zones où sont les objets statiques, puisque leur existence n'est pas certaine, elle dépend des conditions d'exécution. Ils ne peuvent pas davantage être hébergés dans la pile des espaces locaux, puisque cette pile grandit et diminue en accord avec les appels et retours des fonctions, alors que les moments de la création et de la destruction des objets dynamiques ne sont pas connus *a priori*. On place donc les objets dynamiques dans un troisième espace, distinct des deux précédents, appelé le *tas* (*heap*, en anglais).

   La gestion du tas, c'est-à-dire son allocation sous forme de morceaux de tailles variables, à la demande du programme, la récupération des morceaux rendus par ce dernier, la lutte contre l'émiettement de l'espace disponible, etc. font l'objet d'algorithmes savants implémentés dans des fonctions comme *malloc* et *free* ou des opérateurs comme *new* et *delete*.

En définitive, la mémoire utilisée par un programme en cours d'exécution est divisée en quatre espaces (voyez la figure 15) :

[55] Dans certains langages, comme Java, c'est un mécanisme de récupération de la mémoire inutilisée qui se charge de détruire les objets dynamiques dont on peut prouver qu'ils n'interviendront plus dans le programme en cours.

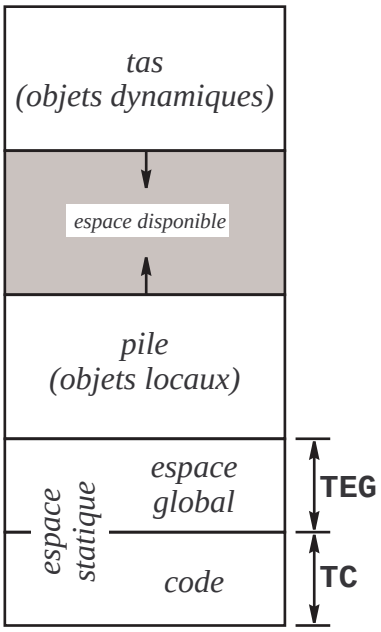


FIG. 15 – Organisation de la mémoire

- le *code* (espace statique en lecture seule), contenant le programme et les constantes,
- l'*espace global* (espace statique en lecture-écriture), contenant les variables globales,
- la *pile* (*stack*), contenant variables locales,
- le *tas* (*heap*), contenant les variables allouées dynamiquement.

Les tailles du code et de l'espace global sont connues dès la fin de la compilation et ne changent pas pendant l'exécution du programme. La pile et le tas, en revanche, évoluent au cours de l'exécution : le tas ne fait que grossir[56], la pile grossit lors des appels de fonctions et diminue lors des terminaisons des fonctions appelées. La rencontre de la pile et du tas (voyez la figure 15) est un accident mortel pour l'exécution du programme. Cela se produit lorsqu'un programme alloue trop d'objets dynamiques et/ou de taille trop importante, ou bien lorsqu'un programme fait trop d'appels de fonctions et/ou ces dernières ont des espaces locaux de taille trop importante.

5.1.2 D'où viennent les adresses des objets ?

Puisque l'adresse d'un objet est un décalage par rapport à la base d'un certain espace qui dépend de la classe de l'objet, du point de vue du compilateur « obtenir l'adresse d'un objet » c'est simplement mémoriser l'état courant d'un certain compteur dont la valeur exprime le niveau de remplissage de l'espace correspondant. Cela suppose que d'autres opérations du compilateur font par ailleurs grandir ce compteur, de sorte que si un peu plus tard on cherche à obtenir l'adresse d'un autre objet on obtiendra une valeur différente.

Plus précisément, pendant qu'il produit la traduction d'un programme, le compilateur utilise les trois variables suivantes :

TC (Taille du Code) Pendant la compilation, cette variable a constamment pour valeur le nombre d'unités (des octets, des mots, cela dépend de la machine) du programme en langage machine couramment produites.

Au début de la compilation, TC vaut 0. Ensuite, si *memoire* représente l'espace (ou le fichier) dans lequel le code machine est mémorisé, la production d'un élément de code, comme un opcode ou un opérande, se traduit par les deux affectations :

$$memoire[TC] \leftarrow element$$
$$TC \leftarrow TC + 1$$

[56] En effet, le rôle des fonctions de restitution de mémoire (*free*, *delete*) est de bien gérer les parcelles de mémoire empruntée puis rendue par un programme ; ces fonctions limitent la croissance du tas, mais elles ne sont pas censées en réduire la taille.

Par conséquent, pour mémoriser l'adresse d'une fonction *fon* il suffit de faire, au moment où commence la compilation de la fonction :

$$adresse(fon) \leftarrow TC$$

TEG (Taille de l'Espace Global) Cette variable a constamment pour valeur la somme des tailles des variables globales dont le compilateur a rencontré la déclaration.

Au début de la compilation TEG vaut 0. Par la suite, le rôle de TEG dans la déclaration d'une variable globale *varg* se résume à ceci :

$$adresse(varg) \leftarrow TEG$$
$$TEG \leftarrow TEG + taille(varg)$$

où $taille(varg)$ représente la taille de la variable en question, qui dépend de son type.

TEL (Taille de l'Espace Local) A l'intérieur d'une fonction, cette variable a pour valeur la somme des tailles des variables locales de la fonction en cours de compilation. Si le compilateur n'est pas dans une fonction TEL n'est pas défini.

A l'entrée de chaque fonction TEL est remis à zéro. Par la suite, le rôle de TEL dans la déclaration d'une variable locale *varl* se résume à ceci :

$$adresse(varl) \leftarrow TEL$$
$$TEL \leftarrow TEL + taille(varl)$$

Les arguments formels de la fonction, bien qu'étant des objets locaux, n'interviennent pas dans le calcul de TEL (la question des adresses des arguments formels sera traitée à la section 5.2.4).

A la fin de la compilation les valeurs des variables TC et TEG sont précieuses :

- TC est la taille du code donc, dans une organisation comme celle de la figure 15, elle est aussi le décalage (relatif à l'origine générale de la mémoire du programme) correspondant à la *base de l'espace global*,
- TEG est la taille de l'espace global ; par conséquent, dans une organisation comme celle de la figure 15, $TC + TEG$ est le décalage (relatif à l'origine générale de la mémoire du programme) correspondant au niveau initial de la pile.

5.1.3 Compilation séparée et édition de liens

Tout identificateur apparaissant dans une partie exécutable d'un programme doit avoir été préalablement déclaré. La déclaration d'un identificateur i, que ce soit le nom d'une variable locale, d'une variable globale ou d'une fonction, produit son introduction dans le dictionnaire adéquat, associé à une adresse, notons-la adr_i. Par la suite, le compilateur remplace chaque occurrence de i dans une expression par le nombre adr_i.

On peut donc penser que dans le code qui sort d'un compilateur les identificateurs qui se trouvaient dans le texte source ont disparu[57] et, de fait, tel peut être le cas dans les langages qui obligent à mettre tout le programme dans un seul fichier.

Mais les choses sont plus compliquées dans les langages, comme C ou Java, où le texte d'un programme peut se trouver éclaté dans plusieurs fichiers sources destinés à être compilés indépendamment les uns des autres (on appelle cela la *compilation séparée*). En effet, dans ces langages il doit être possible qu'une variable ou une fonction déclarée dans un fichier soit mentionnée dans un autre. Cela implique qu'à la fin de la compilation il y a dans le fichier produit quelque trace des noms des variables et fonctions mentionnées dans le fichier source.

Notez que cette question ne concerne que les objets globaux. Les objets locaux, qui ne sont déjà pas visibles en dehors de la fonction dans laquelle ils sont déclarés, ne risquent pas d'être visibles dans un autre fichier.

Notez également que le principal intéressé par cette affaire n'est pas le compilateur, mais un outil qui lui est associé, l'*éditeur de liens* (ou *linker*) dont le rôle est de concaténer plusieurs fichiers objets, résultats de compilations séparées, pour en faire un unique programme exécutable, en vérifiant que les objets référencés mais non définis dans certains fichiers sont bien définis dans d'autres fichiers, et en complétant de telles « références insatisfaites » par les adresses des objets correspondants.

Faute de temps, le langage dont nous écrirons le compilateur ne supportera pas la compilation séparée. Nous n'aurons donc pas besoin d'éditeur de liens dans notre système.

[57] Une conséquence tangible de la disparition des identificateurs est l'impossibilité de « décompiler » les fichiers objets, ce qui n'est pas grave, mais aussi la difficulté de les déboguer, ce qui est plus embêtant. C'est pourquoi la plupart des compilateurs ont une option de compilation qui provoque la conservation des identificateurs avec le code, afin de permettre aux débogueurs d'accéder aux variables et fonctions par leurs noms originaux.

Pour les curieux, voici néanmoins quelques explications sur l'éditeur de liens et la structure des fichiers objets dans les langages qui autorisent la compilation séparée.

Appelons *module* un fichier produit par le compilateur. Le rôle de l'éditeur de liens est de concaténer (mettre bout à bout) plusieurs modules. En général, il concatène d'un côté les zones de code (objets statiques en lecture seule) de chaque module, et d'un autre coté les espaces globaux (objets statiques en lecture écriture) de chaque module, afin d'obtenir une unique zone de code totale et un unique espace global total.

Un problème apparaît immédiatement : si on ne fait rien, les adresses des objets statiques, qui sont exprimées comme des déplacements relatifs à une base p*ropre à chaque module*, vont devenir fausses, sauf pour le module qui se retrouve en tête. C'est facile à voir : chaque module apporte une fonction d'adresse 0 et probablement une variable globale d'adresse 0. Dans l'exécutable final, une seule fonction et une seule variable globale peuvent avoir l'adresse 0.

Il faut donc que l'éditeur de liens passe en revue tout le contenu des modules qu'il concatène et en corrige toutes les références à des objets statiques, pour tenir compte du fait que le début de chaque module ne correspond plus à l'adresse 0 mais à une adresse égale à la somme des tailles des modules qui ont été mis devant lui.

Références absolues et références relatives. En pratique le travail mentionné ci-dessus se trouve allégé par le fait que les langages-machines supportent deux manières de faire référence aux objets statiques.

On dit qu'une instruction fait une *référence absolue* à un objet statique (variable ou fonction) si l'adresse de ce dernier est indiquée par un décalage relatif à la base de l'espace concerné (l'espace global ou l'espace du code). Nous venons de voir que ces références doivent être corrigées lorsque le module qui les contient est décalé et ne commence pas lui-même, dans le fichier qui sort de l'éditeur de liens, à l'adresse 0.

On dit qu'une instruction fait une *référence relative* à un objet statique lorsque ce dernier est repéré par un décalage relatif à l'adresse à laquelle se trouve la référence elle-même.

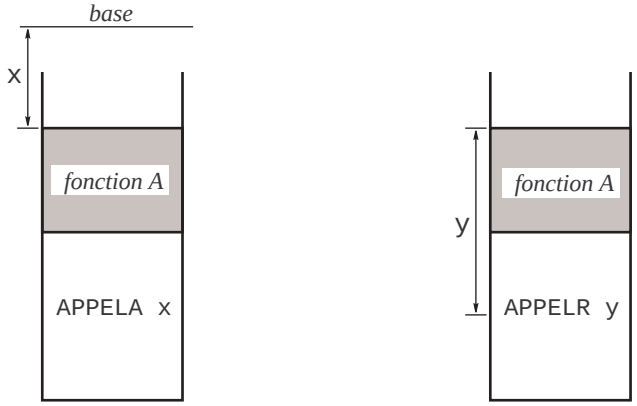


Fig. 16 – Référence absolue et référence relative à une même fonction A

Par exemple, la figure 16 montre le cas d'un langage machine dans lequel il y a deux instructions pour appeler une fonction : `APPELA`, qui prend comme opérande une référence absolue, et `APPELR` qui prend une référence relative.

L'intérêt des références relatives saute aux yeux : elles sont insensibles aux décalages du module dans lequel elles se trouvent. Lorsque l'éditeur de liens concatène des modules pour former un exécutable, les références relatives contenues dans ces modules n'ont pas à être mises à jour.

Bien entendu, cela ne concerne que l'accès aux objets qui se trouvent dans le même module que la référence ; il n'y a aucun intérêt à représenter par une référence relative un accès à un objet d'un autre module. Ainsi, une règle suivie par les compilateurs, lorsque le langage machine le permet, est : *produire des références intra-modules relatives et des références inter-modules absolues.*

Structure des modules. Il nous reste à comprendre comment l'éditeur de liens arrive à attribuer à une référence à un objet non défini dans un module (cela s'appelle une *référence insatisfaite*) l'adresse de l'objet, qui se trouve dans un autre module.

Pour cela il faut savoir que chaque fichier produit par le compilateur se compose de trois parties : une section de code et deux tables faites de couples (*identificateur*, *adresse dans la section de code*) :

- la *section de code* contient la traduction en langage machine d'un fichier source ; ce code comporte un certain nombre de valeurs incorrectes, à savoir les références à des objets externes (i.e. non définis dans ce module) dont l'adresse n'est pas connue au moment de la compilation,
- la table des *références insatisfaites* de la section de code ; dans cette table, chaque identificateur référencé mais non défini est associé à l'adresse de l'élément de code, au contenu incorrect, qu'il faudra corriger lorsque l'adresse de l'objet sera connue,
- la table des *objets publics*[58] définis dans le module ; dans cette table, chaque identificateur est le nom d'un objet que le module en question met à la disposition des autres modules, et il est associé à l'adresse de l'objet concerné dans la section de code .

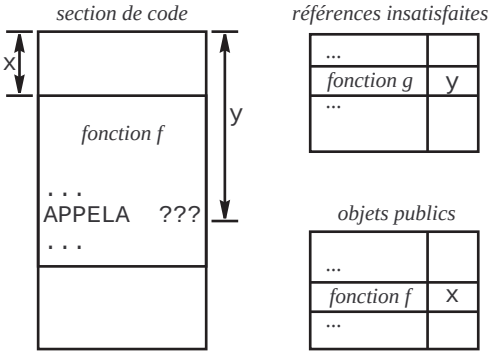


FIG. 17 – Un module objet

Par exemple, la figure 17 représente un module contenant la définition d'une fonction publique f et l'appel d'une fonction non définie dans ce module g.

En résumé, le travail de l'éditeur de liens se compose schématiquement des tâches suivantes :

- concaténation des sections de code des modules donnés à lier ensemble,
- décalage des références absolues contenues dans ces modules (dont les mémorisées dans les tables),
- réunion des tables d'objets publics et utilisation de la table obtenue pour corriger les références insatisfaites présentes dans les sections de code,
- émission du fichier exécutable final, formé du code corrigé.

5.2 La machine Mach 1

Nous continuons notre exposé sur la génération de code par la présentation de la machine Mach 1[59], la machine cible du compilateur que nous réaliserons à titre de projet.

5.2.1 Machines à registres et machines à pile

Les langages évolués permettent l'écriture d'expressions en utilisant la notation algébrique à laquelle nous sommes habitués, comme $X = Y + Z$, cette formule signifiant « ajoutez le contenu de Y à celui de Z et rangez le résultat dans X » (X, Y et Z correspondent à des emplacements dans la mémoire de l'ordinateur).

Une telle expression est trop compliquée pour le processeur, il faut la décomposer en des instructions plus simples. La nature de ces instructions plus simples dépend du type de machine dont on dispose. Relativement à la manière dont les opérations sont exprimées, il y a deux grandes familles de machines :

[58] Quels sont les objets publics, c'est-à-dire les objets définis dans un module qu'on peut référencer depuis d'autres modules ? On l'a déjà dit, un objet public est nécessairement global. Inversement, dans certains langages, tout objet global est public. Dans d'autres langages, aucun objet n'est public par défaut et il faut une déclaration explicite pour rendre publics les objets globaux que le programmeur souhaite. Enfin, dans des langages comme le C, tous les objets globaux sont publics par défaut, et une qualification spéciale (dans le cas de C c'est la qualification `static`) permet d'en rendre certains privés, c'est-à-dire non publics.

[59] Cela se prononce, au choix, « mèque ouane » ou « machun ».

1. Les *machines à registres* possèdent un certain nombre de registres, notés ici `R1`, `R2`, etc., qui sont les seuls composants susceptibles d'intervenir dans une opérations (autre qu'un transfert de mémoire à registre ou réciproquement) à titre d'opérandes ou de résultats. Inversement, n'importe quel registre peut intervenir dans une opération arithmétique ou autre ; par conséquent, les instructions qui expriment ces opérations doivent spécifier leurs opérandes. Si on vise une telle machine, l'affectation $X = Y + Z$ devra être traduite en quelque chose comme (notant `X`, `Y` et `Z` les adresses des variables X, Y et Z) :

```
MOVE  Y,R1    // déplace la valeur de Y dans R1
MOVE  Z,R2    // déplace la valeur de Z dans R2
ADD   R1,R2   // ajoute R1 à R2
MOVE  R2,X    // déplace la valeur de R2 dans X
```

2. Les *machines à pile*, au contraire, disposent d'une pile (qui peut être la pile des variables locales) au sommet de laquelle se font toutes les opérations. Plus précisément :

- les opérandes d'un opérateur binaire sont toujours les deux valeurs au sommet de la pile ; l'exécution d'une opération binaire $\otimes$ consiste toujours à dépiler deux valeurs x et y et à empiler le résultat $x \otimes y$ de l'opération,
- l'opérande d'un opérateur unaire est la valeur au sommet de la pile ; l'exécution d'une opération unaire $\triangleright$ consiste toujours à dépiler une valeur x et à empiler le résultat $\triangleright x$ de l'opération.

Pour une telle machine, le code $X = Y + Z$ se traduira donc ainsi (notant encore `X`, `Y` et `Z` les adresses des variables X, Y et Z) :

```
PUSH  Y       // met la valeur de Y au sommet de la pile
PUSH  Z       // met la valeur de Z au sommet de la pile
ADD           // remplace les deux valeurs au sommet de la pile par leur somme
POP   X       // enlève la valeur au sommet de la pile et la range dans X
```

Il est *a priori* plus facile de produire du code de bonne qualité pour une machine à pile plutôt que pour une machine à registres, mais ce défaut est largement corrigé dans les compilateurs commerciaux par l'utilisation de savants algorithmes qui optimisent l'utilisation des registres.

Car il faut savoir que les machines « physiques » existantes sont presque toujours des machines à registres, pour une raison d'efficacité facile à comprendre : les opérations qui ne concernent que des registres restent internes au microprocesseur et ne sollicitent ni le bus ni la mémoire de la machine[60]. Et, avec un compilateur optimisateur, les expressions complexes sont traduites par des suites d'instructions dont la plupart ne mentionnent que des registres.

Les algorithmes d'optimisation et d'allocation des registres sortant du cadre de ce cours, la machine pour laquelle nous générerons du code sera une machine à pile.

5.2.2 Structure générale de la machine Mach 1

La mémoire de la machine Mach 1 est faite de cellules numérotées, organisées comme le montre la figure 18 (c'est la structure de la figure 15, le tas en moins). Les registres suivants ont un rôle essentiel dans le fonctionnement de la machine :

`CO` (Compteur Ordinal) indique constamment la cellule contenant l'instruction que la machine est en train d'exécuter,

`BEG` (Base de l'Espace Global) indique la première cellule de l'espace réservé aux variables globales (autrement dit, `BEG` pointe la variable globale d'adresse 0),

`BEL` (Base de l'Espace Local) indique la cellule autour de laquelle est organisé l'espace local de la fonction en cours d'exécution ; la valeur de `BEL` change lorsque l'activation d'une fonction commence ou finit (à ce propos voir la section 5.2.4),

`SP` (Sommet de la Pile) indique constamment le sommet de la pile, ou plus exactement la première cellule libre au-dessus de la pile, c'est-à-dire le nombre total de cellules occupées dans la mémoire.

[60] Se rappeler que de nos jours, décembre 2001, le microprocesseur d'un ordinateur personnel moyennement puissant tourne à 2 GHz (c'est sa cadence interne) alors que son bus et sa mémoire ne travaillent qu'à 133 MHz.

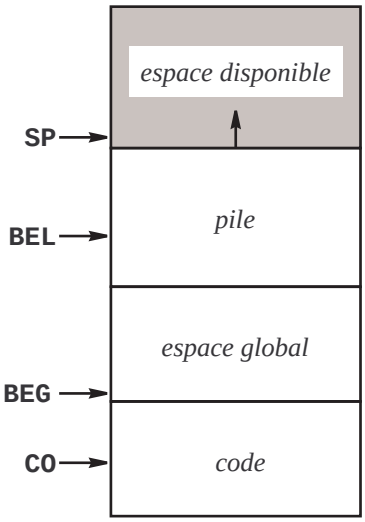


FIG. 18 – Organisation de la mémoire de la machine Mach 1

5.2.3 Jeu d'instructions

La machine Mach 1 est une machine à pile.

Chaque instruction est faite soit d'un seul opcode, elle occupe alors une cellule, soit d'un opcode et un opérande, elle occupe alors deux cellules consécutives. Les constantes entières et les « adresses » ont la même taille, qui est celle d'une cellule. La table 1, à la fin de ce polycopié, donne la liste des instructions.

5.2.4 Compléments sur l'appel des fonctions

La création et la destruction de l'espace local des fonctions a lieu à quatre moments bien déterminés : l'activation de la fonction, d'abord du point de vue de la fonction appelante (1), puis de celui de la fonction appelée (2), ensuite le retour, d'abord du point de vue de la fonction appelée (3) puis de celui de la fonction appelante (4). Voici ce qui se passe (voyez la figure 19) :

1. La fonction appelante réserve un mot vide sur la pile, où sera déposé le résultat de la fonction, puis elle empile les valeurs des arguments effectifs. Ensuite, elle exécute une instruction APPEL (qui empile l'adresse de retour).
2. La fonction appelée empile le « BEL courant » (qui est en train de devenir « BEL précédent »), prend la valeur de SP pour BEL courant, puis alloue l'espace local. Pendant la durée de l'activation de la fonction :
   - les variables locales sont atteintes à travers des déplacements (positifs ou nuls) relatifs à BEL : $0 \leftrightarrow$ première variable locale, $1 \leftrightarrow$ deuxième variable locale, etc.
   - les arguments formels sont également atteints à travers des déplacements (négatifs, cette fois) relatifs à BEL : $-3 \leftrightarrow n$-ème argument, $-4 \leftrightarrow (n-1)$-ème argument, etc.
   - la cellule où la fonction doit déposer son résultat est atteinte elle aussi à travers un déplacement relatif à BEL. Ce déplacement est $-(n+3)$, n étant le nombre d'arguments formels, et suppose donc l'accord entre la fonction appelante et la fonction appelée au sujet du nombre d'arguments de la fonction appelée.

   L'ensemble de toutes ces valeurs forme l'« espace local » de la fonction en cours. Au-delà de cet espace, la pile est utilisée pour le calcul des expressions courantes.
3. Terminaison du travail : la fonction appelée remet BEL et SP comme ils étaient lorsqu'elle a été activée, puis effectue une instruction RETOUR.
4. Reprise du calcul interrompu par l'appel. La fonction appelante libère l'espace occupé par les valeurs des arguments effectifs. Elle se retrouve alors avec une pile au sommet de laquelle est le résultat de la fonction qui vient d'être appelée. Globalement, la situation est la même qu'après une opération arithmétique : les opérandes ont été dépilés et remplacés par le résultat.

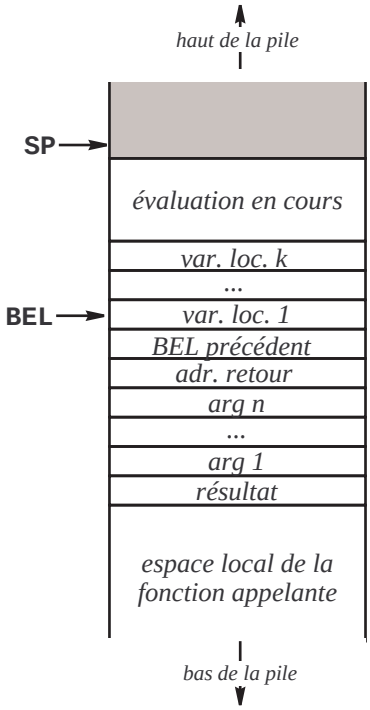


FIG. 19 – Espace local d'une fonction

QUI CRÉE L'ESPACE LOCAL D'UNE FONCTION ? Les arguments d'une fonction et les variables déclarées à l'intérieur de cette dernière sont des objets locaux et ont donc des adresses qui sont des déplacements relatifs à `BEL` (positifs dans le cas des variables locales, négatifs dans le cas des arguments, voir la figure 19). L'explication précédente montre qu'il y a une différence importante entre ces deux sortes d'objets locaux :

- les arguments sont installés dans la pile par la fonction appelante, car ce sont les valeurs d'expressions qui doivent être évaluées dans le contexte de la fonction appelante. Il est donc naturel de donner à cette dernière la responsabilité de les enlever de la pile, au retour de la fonction appelée,
- les variables locales sont allouées par la fonction appelée, qui est la seule à en connaître le nombre ; c'est donc cette fonction qui doit les enlever de la pile, lorsqu'elle se termine (l'instruction `SORTIE` fait ce travail).

LES CONVENTIONS D'APPEL. Il y a donc un ensemble de conventions, dites *conventions d'appel*, qui précisent à quel endroit la fonction appelante doit déposer les valeurs des arguments et où trouvera-t-elle le résultat ou, ce qui revient au même, à quel endroit la fonction appelée trouvera ses arguments et où doit-elle déposer son résultat.

En général les conventions d'appel ne sont pas laissées à l'appréciation des auteurs de compilateurs. Édictées par les concepteurs du système d'exploitation, elles sont suivies par tous les compilateurs « homologués », ce qui a une conséquence très importante : puisque tous les compilateurs produisent des codes dans lesquels les paramètres et le résultat des fonctions sont passés de la même manière, une fonction écrite dans un langage L_1, et compilée donc par le compilateur de L_1, pourra appeler une fonction écrite dans un autre langage L_2 et compilée par un autre compilateur, celui de L_2.

Nous avons choisi ici des conventions d'appel simples et pratiques (beaucoup de systèmes font ainsi) puisque :

- les arguments sont empilés dans l'ordre dans lequel ils se présentent dans l'expression d'appel,
- le résultat est déposé là où il faut pour que, après nettoyage des arguments, il soit au sommet de la pile.

Mais il y a un petit inconvénient[61]. Dans notre système, les argument arg_1, arg_2, ... arg_n sont atteints, à l'intérieur de la fonction, par les adresses locales respectives $-(n+3)+1$, $-(n+3)+2$, ... $-(n+3)+n=-3$, et le résultat de la fonction lui-même possède l'adresse locale $-(n+3)$. Cela oblige la fonction appelée à connaître

[61] Et encore, ce n'est pas sur que ce soit un inconvénient, tellement il semble naturel d'exiger, comme en Pascal ou Java, que la fonction appelante et la fonction appelée soient d'accord sur le nombre d'arguments de cette dernière.

le nombre n d'arguments effectifs, et interdit l'écriture de fonctions admettant un nombre variable d'arguments (de telles fonctions sont possibles en C, songez à `printf` et `scanf`).

5.3 Exemples de production de code

Quand on n'est pas connaisseur de la question on peut croire que la génération de code machine est la partie la plus importante d'un compilateur, et donc que c'est elle qui détermine la structure générale de ce dernier. On est alors surpris, voire frustré, en constatant la place considérable que les ouvrages spécialisés consacrent à l'analyse (lexicale, syntaxique, sémantique, etc.). C'est que[62] :

> La bonne manière d'écrire un compilateur du langage L pour la machine M consiste à écrire un analyseur du langage L auquel, dans un deuxième temps, on ajoute –sans rien lui enlever– les opérations qui produisent du code pour la machine M.

Dans cette section nous expliquons, à travers des exemples, comment ajouter à notre analyseur les opérations de génération de code qui en font un compilateur. Il faudra se souvenir de ceci : quand on réfléchit à la génération de code on est concerné par deux exécutions différentes : d'une part l'exécution du compilateur que nous réalisons, qui produit comme résultat un programme P en langage machine, d'autre part l'exécution du programme P ; *a priori* cette exécution a lieu ultérieurement, mais nous devons l'imaginer en même temps que nous écrivons le compilateur, pour comprendre pourquoi nous faisons que ce dernier produise ce qu'il produit.

5.3.1 Expressions arithmétiques

Puisque la machine Mach 1 est une machine à pile, la traduction des expressions arithmétiques, aussi complexes soient-elles, est extrêmement simple et élégante. A titre d'exemple, imaginons que notre langage source est une sorte de « C francisé », et considérons la situation suivante (on suppose qu'il n'y a pas d'autres déclarations de variables que celles qu'on voit ici) :

```
entier x, y, z;
...
entier uneFonction() {
    entier a, b, c;
    ...
    y = 123 * x + c;
}
```

Pour commencer, intéressons-nous à l'expression `123 * x + c`. Son analyse syntaxique aura été faite par des fonctions *expArith* (expression arithmétique) et *finExpArith* ressemblant à ceci :

```
void expArith(void) {
    terme();
    finExpArith();
}

void finExpArith(void) {
    if (uniteCourante == '+' || uniteCourante == '-') {
        uniteCourante = uniteSuivante();
        terme();
        finExpArith();
    }
}
```

(les fonctions *terme* et *finTerme* sont analogues aux deux précédentes, avec '$*$' et '/' dans les rôles de '+' et '$-$', et *facteur* dans le rôle de *terme*). Enfin, une version simplifiée[63] de la fonction *facteur* pourrait commencer comme ceci :

[62] Plus généralement : la bonne façon d'obtenir l'information portée par un texte soumis à une syntaxe consiste à écrire l'analyseur syntaxique correspondant et, dans un deuxième temps, à lui ajouter les opérations qui construisent les informations en question.
Ce principe est tout à fait fondamental. Si vous ne deviez retenir qu'une seule chose de ce cours, que ce soit cela.

[63] Attention, nous envisageons momentanément un langage ultra-simple, sans tableaux ni appels de fonctions, dans lequel une occurrence d'un identificateur dans une expression indique toujours à une variable simple.

```
void facteur(void) {
    if (uniteCourante == NOMBRE)
        uniteCourante = uniteSuivante();
    else
        if (uniteCourante == IDENTIFICATEUR)
            uniteCourante = uniteSuivante();
        else
            ...
}
```

Le principe de fonctionnement d'une machine à pile, expliqué à la section 5.2.1, a la conséquence fondamentale suivante :

1. La compilation d'une *expression* produit une suite d'instructions du langage machine (plus ou moins longue, selon la complexité de l'expression) dont l'exécution a pour effet global d'ajouter *une* valeur au sommet de la pile, à savoir le résultat de l'évaluation de l'expression.
2. La compilation d'une *instruction* du langage source produit une suite d'instructions du langage machine (plus ou moins longue, selon la complexité de l'expression) dont l'exécution laisse la pile dans l'état où elle se trouvait avant de commencer l'instruction en question.

Une manière de retrouver quel doit être le code produit pour la compilation de l'expression `123 * x + c` consiste à se dire que `123` est déjà une expression correcte ; l'effet d'une telle expression doit être de mettre au sommet de la pile la valeur `123`. Par conséquent, la compilation de l'expression `123` doit donner le code

```
| EMPC 123 |
```

De même, la compilation de l'expression `x` doit donner

```
| EMPG   0 |
```

(car `x` est, dans notre exemple, la première variable globale) et celle de l'expression `c` doit donner

```
| EMPL   2 |
```

(car `c` est la troisième variable locale). Pour obtenir ces codes il suffit de transformer la fonction facteur comme ceci :

```
void facteur(void) {
    if (uniteCourante == NOMBRE) {
        genCode(EMPC);
        genCode(atoi(lexeme));
        uniteCourante = uniteSuivante();
    }
    else
        if (uniteCourante == IDENTIFICATEUR) {
            ENTREE_DICO *pDesc = rechercher(lexeme);
            genCode(pDesc->classe == VARIABLE_GLOBALE ? EMPG : EMPL);
            genCode(pDesc->adresse);
            uniteCourante = uniteSuivante();
        }
        else
            ...
}
```

La fonction *genCode* se charge de placer un élément (un opcode ou un opérande) dans le code. Si nous supposons que ce dernier est rangé dans la mémoire, ce pourrait être tout simplement :

```
void genCode(int element) {
    mem[TC++] = element;
}
```

Pour la production de code, les fonctions *expArith* et *terme* n'ont pas besoin d'être modifiées. Seules les fonctions dans lesquelles des opérateurs apparaissent explicitement doivent l'être :

```
void finExpArith(void) {
    int uc = uniteCourante;
    if (uniteCourante == '+' || uniteCourante == '-') {
        uniteCourante = uniteSuivante();
        terme();
        genCode(uc == '+' ? ADD : SOUS);
        finExpArith();
    }
}

void finterme(void) {
    int uc = uniteCourante;
    if (uniteCourante == '*' || uniteCourante == '/') {
        uniteCourante = uniteSuivante();
        facteur();
        genCode(uc == '*' ? MUL : DIV);
        finTerme();
    }
}
```

Au final, le code produit à l'occasion de la compilation de l'expression `123 * x + c` sera donc :

```
EMPC 123
EMPG   0
MUL
EMPL   2
ADD
```

et, en imaginant l'exécution de la séquence ci-dessus, on constate que son effet global aura bien été d'ajouter une valeur au sommet de la pile.

Considérons maintenant l'instruction complète `y = 123 * x + c`. Dans un compilateur ultra-simplifié qui ignorerait les appels de fonction et les tableaux, on peut imaginer qu'elle aura été analysée par une fonction *instruction* commençant comme ceci :

```
void instruction(void) {
    if (uniteCourante == IDENTIFICATEUR) {    /* instruction d'affectation */
        uniteCourante = uniteSuivante();
        terminal('=');
        expArith();
        terminal(';');
    }
    else
        ...
}
```

pour qu'il produise du code il faut transformer cet analyseur comme ceci :

```
void instruction(void) {
    if (uniteCourante == IDENTIFICATEUR) {    /* instruction d'affectation */
        ENTREE_DICO *pDesc = rechercher(lexeme);
        uniteCourante = uniteSuivante();
        terminal('=');
        expArith();
        genCode(pDesc->classe == VARIABLE_GLOBALE ? DEPG : DEPL);
        genCode(pDesc->adresse);
        terminal(';');
    }
    else
        ...
}
```

on voit alors que le code finalement produit par la compilation de `y = 123 * x + c` aura été :

```
EMPC 123
EMPG   0
MUL
EMPL   2
ADD
DEPG   1
```

(`y` est la deuxième variable globale). Comme prévu, l'exécution du code précédent laisse la pile comme elle était en commençant.

5.3.2 Instruction conditionnelle et boucles

La plupart des langages modernes possèdent des instructions conditionnelles et des « boucles tant que » définies par des productions analogues à la suivante :

instruction → ... |
`tantque` *expression* `faire` *instruction* |
`si` *expression* `alors` *instruction* |
`si` *expression* `alors` *instruction* `sinon` *instruction* |
...

La partie de l'analyseur syntaxique correspondant à la règle précédente ressemblerait à ceci[64] :

```
void instruction(void) {
    ...
    else if (uniteCourante == TANTQUE) {    /* boucle "tant que" */
        uniteCourante = uniteSuivante();
        expression();
        terminal(FAIRE);
        instruction();
    }
    else if (uniteCourante == SI) {         /* instruction conditionnelle */
        uniteCourante = uniteSuivante();
        expression();
        terminal(ALORS);
        instruction();
        if (uniteCourante == SINON) {
            uniteCourante = uniteSuivante();
            instruction();
        }
    }
    else
    ...
}
```

Commençons par le code à générer à l'occasion d'une boucle « tant que ». Dans ce code on trouvera les suites d'instructions générées pour l'*expression* et pour l'*instruction* que la syntaxe prévoit (ces deux suites peuvent être très longues, cela dépend de la complexité de l'expression et de l'instruction en question).

L'effet de cette instruction est connu : l'*expression* est évaluée ; si elle est fausse (c.-à-d. nulle) l'exécution continue après le corps de la boucle ; si l'expression est vraie (c.-à-d. non nulle) le corps de la boucle est exécuté, puis on revient à l'évaluation de l'expression et on recommence tout.

Notons $expr_1$, $expr_2$, ... $expr_n$ les codes-machine produits par la compilation de l'*expression* et $instr_1$, $instr_2$, ... $instr_k$ ceux produits par la compilation de l'*instruction*. Pour l'instruction « tant que » toute entière on aura donc un code comme ceci (les repères dans la colonne de gauche, comme α et β, représentent les adresses des instructions) :

[64] Notez que, dans le cas de l'instruction conditionnelle, l'utilisation d'un analyseur récursif descendant nous a permis de faire une factorisation à gauche originale du début commun des deux formes de l'instruction si (à propos de factorisation à gauche voyez éventuellement la section 3.1.3).

α	$expr_1$	
	$expr_2$	
	...	
	$expr_n$	
	SIFAUX	β
	$instr_1$	
	$instr_2$	
	...	
	$instr_k$	
	SAUT	α
β		

Le point le plus difficile, ici, est de réaliser qu'au moment où le compilateur doit produire l'instruction `SIFAUX` β, l'adresse β n'est pas connue. Il faut donc produire quelque chose comme `SIFAUX 0` et noter que la case dans laquelle est inscrit le `0` est à corriger ultérieurement (quand la valeur de β sera connue). Ce qui donne le programme :

```
void instruction(void) {
    int alpha, aCompleter;
    ...
    else if (uniteCourante == TANTQUE) {    /* boucle "tant que" */
        uniteCourante = uniteSuivante();
        alpha = TC;
        expression();
        genCode(SIFAUX);
        aCompleter = TC;
        genCode(0);
        terminal(FAIRE);
        instruction();
        genCode(SAUT);
        genCode(alpha);
        repCode(aCompleter, TC);    /* ici, beta = TC */
    }
    else
    ...
}
```

où *repCode* (pour « réparer le code ») est une fonction aussi simple que *genCode* (cette fonction est très simple parce que nous supposons que notre compilateur produit du code dans la mémoire ; elle serait considérablement plus complexe si le code était produit dans un fichier) :

```
void repCode(int place, int valeur) {
    mem[place] = valeur;
}
```

Instruction conditionnelle. Dans le même ordre d'idées, voici le code à produire dans le cas de l'instruction conditionnelle à une branche ; il n'est pas difficile d'imaginer ce qu'il faut ajouter, et où, dans l'analyseur montré plus haut pour lui faire produire le code suivant :

	...	
	$expr_1$	*début de l'instruction si...alors...*
	$expr_2$	
	...	
	$expr_n$	
	SIFAUX β	
	$instr_1$	
	$instr_2$	
	...	
	$instr_k$	
β	...	*la suite (après l'instruction si...alors...)*

Et voici le code à produire pour l'instruction conditionnelle à deux branches. On note $alors_1$, $alors_2$, ... $alors_k$ les codes-machine produits par la compilation de la première branche et $sinon_1$, $sinon_2$, ... $sinon_m$ ceux produits par la compilation de la deuxième branche :

	$expr_1$	*début de l'instruction si...alors...sinon...*
	$expr_2$	
	...	
	$expr_n$	
	`SIFAUX` β	
	$alors_1$	*début de la première branche*
	$alors_2$	
	...	
	$alors_k$	
	`SAUT` γ	
β	$sinon_1$	*début de la deuxième branche*
	$sinon_2$	
	...	
	$sinon_m$	
γ	...	*la suite (après l'instruction si...alors...sinon...)*

5.3.3 Appel de fonction

Pour finir cette galerie d'exemples, examinons quel doit être le code produit à l'occasion de l'appel d'une fonction, aussi bien du côté de la fonction appelante que de celui de la fonction appelée.

Supposons que l'on a d'une part compilé la fonction suivante :

```
entier distance(entier a, entier b) {
    entier x;
    x = a - b;
    si x < 0 alors
        x = - x;
    retour x;
}
```

et supposons que l'on a d'autre part le code (`u`, `v`, `w` sont les seules variables globales de ce programme) :

```
entier u, v, w;
...
entier principale() {
    ...
    u := 200 - distance(v, w / 25 - 100) * 2;
    ...
}
```

Voici le segment du code de la fonction principale qui est la traduction de l'affectation précédente :

```
...
EMPC 200
PILE   1    emplacement pour le résultat de la fonction
EMPG   1    premier argument
EMPG   2
EMPC  25
DIV
EMPC 100
SUB         ceci achève le calcul du second argument
APPEL  α
PILE  -2    on vire les arguments (ensuite le résultat de la fonction est au sommet)
EMPC   2
MUL
SUB         ceci achève le calcul du membre droit de l'affectation
DEPG   0
...
```

et voici le code de la fonction *distance* (voyez la section 5.2.4) :

```
...
α  ENTREE
   PILE   1    allocation de l'espace local (une variable)
   EMPL  -4
   EMPL  -3
   SUB
   DEPL   0
   EMPL   0    debut de l'instruction si
   EMPC   0
   INF
   SIFAUX β
   EMPC   0    ]
   EMPL   0    ] simulation du – unaire
   SUB         ]
   DEPL   0
β  EMPL   0
   DEPL  -5    le résultat de la fonction
   SORTIE
   RETOUR
...
```

Références

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs. Principes, techniques et outils.* Dunod, Paris, 1991.

[2] Alfred Aho and Jeffrey Ullman. *The Theory of Parsing, Translating and Compiling.* Prentice Hall Inc, 1972.

[3] Andrew Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[4] John Levine, Tony Mason, and Doug Brown. *Lex & yacc.* O'Reilly & Associates, Inc., 1990.

TAB. 1 – Les instructions de la machine *Mach 1*

opcode	*opérande*	*explication*
EMPC	valeur	*EMPiler Constante*. Empile la valeur indiquée.
EMPL	adresse	*EMPiler la valeur d'une variable Locale*. Empile la valeur de la variable déterminée par le déplacement relatif à BEL donné par adresse (entier relatif).
DEPL	adresse	*DEPiler dans une variable Locale*. Dépile la valeur qui est au sommet et la range dans la variable déterminée par le déplacement relatif à BEL donné par adresse (entier relatif).
EMPG	adresse	*EMPiler la valeur d'une variable Globale*. Empile la valeur de la variable déterminée par le déplacement (relatif à BEG) donné par adresse.
DEPG	adresse	*DEPiler dans une variable Globale*. Dépile la valeur qui est au sommet et la range dans la variable déterminée par le déplacement (relatif à BEG) donné par adresse.
EMPT	adresse	*EMPiler la valeur d'un élément de Tableau*. Dépile la valeur qui est au sommet de la pile, soit i cette valeur. Empile la valeur de la cellule qui se trouve i cases au-delà de la variable déterminée par le déplacement (relatif à BEG) indiqué par adresse.
DEPT	adresse	*DEPiler dans un élément de Tableau*. Dépile une valeur v, puis une valeur i. Ensuite range v dans la cellule qui se trouve i cases au-delà de la variable déterminée par le déplacement (relatif à BEG) indiqué par adresse.
ADD		*ADDition*. Dépile deux valeurs et empile le résultat de leur addition.
SOUS		*SOUStraction*. Dépile deux valeurs et empile le résultat de leur soustraction.
MUL		*MULtiplication*. Dépile deux valeurs et empile le résultat de leur multiplication.
DIV		*DIVision*. Dépile deux valeurs et empile le quotient de leur division euclidienne.
MOD		*MODulo*. Dépile deux valeurs et empile le reste de leur division euclidienne.
EGAL		Dépile deux valeurs et empile 1 si elles sont égales, 0 sinon.
INF		*INFerieur*. Dépile deux valeurs et empile 1 si la première est inférieure à la seconde, 0 sinon.
INFEG		*INFerieur ou EGal*. Dépile deux valeurs et empile 1 si la première est inférieure ou égale à la seconde, 0 sinon.
NON		Dépile une valeur et empile 1 si elle est nulle, 0 sinon.
LIRE		Obtient de l'utilisateur un nombre et l'empile
ECRIV		*ECRIre Valeur*. Extrait la valeur qui est au sommet de la pile et l'affiche
SAUT	adresse	*Saut inconditionnel*. L'exécution continue par l'instruction ayant l'adresse indiquée.
SIVRAI	adresse	*Saut conditionnel*. Dépile une valeur et si elle est non nulle, l'exécution continue par l'instruction ayant l'adresse indiquée. Si la valeur dépilée est nulle, l'exécution continue normalement.
SIFAUX	adresse	Comme ci-dessus, en permutant nul et non nul.
APPEL	adresse	*Appel de sous-programme*. Empile l'adresse de l'instruction suivante, puis fait la même chose que SAUT.
RETOUR		*Retour de sous-programme*. Dépile une valeur et continue l'exécution par l'instruction dont c'est l'adresse.
ENTREE		*Entrée dans un sous-programme*. Empile la valeur courante de BEL, puis copie la valeur de SP dans BEL.
SORTIE		*Sortie d'un sous-programme*. Copie la valeur de BEL dans SP, puis dépile une valeur et la range dans BEL.
PILE	nbreMots	*Allocation et restitution d'espace dans la pile*. Ajoute nbreMots, qui est un entier positif ou négatif, à SP
STOP		La machine s'arrête.