fasm

FR - 2026

# flat assembler g
## Manuel de l'utilisateur

**par Tomasz Grysztar**

Date du document original : **19 juillet 2026**

*Traduction française*

By Calendos
Forum https://flatassembler.net/
Traduction du 31 juillet/2026
d'après un prélèvement effectué le 21 juillet 2026

Ce document décrit la syntaxe du langage *flat assembler g*, avec des exemples de base. Il a été rédigé dans l'optique d'une lecture séquentielle et n'utilise à chaque étape que les concepts et constructions introduits précédemment. Le lecteur peut toutefois accéder directement à la section qui l'intéresse et revenir aux parties précédentes uniquement si nécessaire pour mieux comprendre les suivantes.

# Table des matières



## 0. Éxécution de l'assembleur

Pour lancer l'assemblage en ligne de commande, il est nécessaire de fournir au moins un paramètre : le nom du fichier source, et éventuellement un second : le nom du fichier de destination. Si l'assemblage réussit, le code généré est écrit dans le fichier de destination et un bref résumé de l'opération est affiché ; sinon, les informations relatives aux erreurs sont affichées. Le nombre maximal d'erreurs affichées peut être contrôlé avec l'option supplémentaire "`-e`" (par défaut, une seule erreur est affichée). L'option "`-p`" contrôle le nombre maximal de passes que l'assembleur effectuera. Cette limite est fixée par défaut à 100. L'option "`-r`" permet de définir la limite de la pile de récursivité, c'est-à-dire la profondeur maximale autorisée pour l'exécution des macro-instructions et l'inclusion de fichiers sources supplémentaires. L'option "`-v`" permet d'afficher toutes les lignes de cette pile lors du signalement d'une erreur (par défaut, l'assembleur sélectionne uniquement les lignes les plus informatives, mais cette méthode, bien que simple, peut ne pas toujours être optimale). Si l'option "`-v`" est utilisée avec la valeur 2, elle permet en outre d'afficher en temps réel (à chaque passage) tous les messages issus des commandes du texte source. L'option "`-i`" permet d'insérer une commande quelconque au début du texte source traité.

## 1. Règles de syntaxe fondamentales

Chaque commande en langage assembleur occupe une seule ligne. Si une ligne contient un point-virgule, tout ce qui suit est considéré comme un commentaire et ignoré par l'assembleur. Le contenu d'une ligne (sans le commentaire) peut se terminer par une barre oblique inverse ; dans ce cas, la ligne suivante du texte source est ajoutée à la suite. Cela permet de répartir une commande sur plusieurs lignes, si nécessaire. Désormais, nous désignerons une ligne source comme une entité obtenue en supprimant les commentaires et en concaténant les lignes reliées par des barres obliques inverses.

Le texte d'une ligne source est divisé en unités syntaxiques appelées jetons. Certains caractères spéciaux constituent des jetons à part entière. Les caractères listés ci-dessous sont des unités syntaxiques :

```
+-/*=<>()[]{}:?!,.|&~#\
```

Toute séquence de caractères contiguë (c'est-à-dire non interrompue par un espace) autre que celles mentionnées ci-dessus forme un seul jeton, qui peut être un nom ou un nombre. L'exception à cette règle est une séquence commençant par un guillemet simple ou double. On parle alors de chaîne de caractères entre guillemets, qui peut contenir n'importe quel caractère spécial, des espaces et même des points-virgules, car elle ne se termine que lorsqu'on rencontre le même caractère que celui utilisé pour la commencer. Les guillemets qui encadrent la chaîne ne font pas partie de celle-ci. Si l'on souhaite définir une chaîne contenant le même caractère que celui qui

l'encadre, il faut le dupliquer à l'intérieur de la chaîne ; une seule copie du caractère sera intégrée à la chaîne, et la séquence se poursuivra.

Les nombres se distinguent des noms par le fait qu'ils commencent soit par un chiffre décimal, soit par le caractère "$" suivi d'un chiffre hexadécimal. Ainsi, un jeton peut être considéré comme numérique même s'il ne s'agit pas d'un nombre valide. Pour être valide, un jeton doit correspondre à l'un des éléments suivants : un nombre décimal (éventuellement suivi de la lettre "d"), un nombre binaire suivi de la lettre "b", un nombre octal suivi de la lettre "o" ou "q", ou un nombre hexadécimal précédé de "$" ou "0x", ou suivi du caractère "h". Le premier chiffre d'un nombre hexadécimal pouvant être une lettre, il peut être nécessaire de le faire précéder d'un zéro pour qu'il soit reconnu comme un nombre. Par exemple, "0Ah" est un nombre valide, tandis que "Ah" est un nom.

Outre les chiffres, les nombres peuvent également contenir des traits de soulignement ou des guillemets simples servant de séparateur ou de remplissage, afin de faciliter la lecture des nombres plus longs.

## 2. Identifiants de symboles

Tout nom peut devenir un symbole défini en se voyant attribuer une signification (une valeur). L'une des méthodes les plus simples pour créer un symbole avec une valeur donnée consiste à utiliser la commande "=" :

```
a = 1
```

La commande ":" définit une étiquette, c'est-à-dire un symbole dont la valeur correspond à l'adresse actuelle dans la sortie générée. Au début du texte source, cette adresse est toujours nulle ; par conséquent, lorsque les deux commandes suivantes sont les premières du fichier source, elles définissent des symboles ayant des valeurs identiques :

```
first:
second = 0
```

Les étiquettes définies avec la commande ":" sont des constructions spéciales en langage assembleur, car elles permettent à toute autre commande (y compris une autre définition d'étiquette) de suivre sur la même ligne. C'est le seul type de commande qui le permet.

Ce qui précède le caractère ":" ou "=" dans une telle définition est un identificateur de symbole. Il peut s'agir d'un nom simple, comme dans les exemples ci-dessus, mais il peut également contenir des modificateurs supplémentaires, décrits ci-dessous.

Lorsqu'un nom dans une définition de symbole est suivi du caractère "?" (sans espace), le symbole est insensible à la casse (sinon, il serait sensible à la casse). Cela signifie que la valeur de ce symbole peut être référencée (comme dans une expression à droite du caractère "=") par n'importe quelle variante du nom original qui ne diffère que par la casse des lettres. Seules les casses des 26 lettres de l'alphabet anglais sont autorisées.

Il est possible de définir un symbole sensible à la casse qui entre en conflit avec un symbole insensible à la casse. Dans ce cas, le symbole sensible à la casse prévaut et le symbole plus général n'est utilisé que lorsqu'aucun symbole sensible à la casse correspondant n'est défini. On peut remédier à ce problème en utilisant le modificateur "?", car il indique toujours que le nom qui le suit fait référence au symbole insensible à la casse.

```
tester? = 0
tester = 1
TESTER = 2
x = tester        ; x = 1
y = Tester        ; y = 0
z = TESTER        ; z = 2
t = tester?       ; t = 0
```

Chaque symbole possède son propre espace de noms de descendants, appelé espace de noms enfants. Lorsque deux noms sont reliés par un point (sans espace), cet identifiant fait référence à une entité nommée par le second dans l'espace de noms des descendants du symbole spécifié par le premier. Cette opération peut être répétée plusieurs fois au sein d'un même identifiant, permettant ainsi de faire référence aux descendants des descendants dans une chaîne de longueur quelconque.

```
space:
space.x = 1
space.y = 2
space.color:
space.color.r = 0
space.color.g = 0
space.color.b = 0
```

Dans une telle chaîne, n'importe quel nom peut être suivi du caractère "?" pour indiquer qu'il fait référence à un symbole insensible à la casse. Si "?" est inséré au milieu du nom (le divisant ainsi en plusieurs jetons), l'identificateur est considéré comme une erreur de syntaxe.

Lorsqu'un identificateur commence par un point (autrement dit : lorsque le nom du symbole parent est vide), il fait référence au symbole de l'espace de noms de l'étiquette régulière la plus récente définie avant la ligne courante. Cela permet de réécrire l'exemple ci-dessus comme suit :

```
space:
.x = 1
.y = 2
.color:
.color.r = 0
.color.g = 0
.color.b = 0
```

Une fois l'étiquette “space” définie, elle devient l'étiquette normale la plus récemment définie. Ainsi, “.x” fait référence au symbole “space.x” et “.color” au symbole “space.color”.

La commande “namespace”, suivie d'un identificateur de symbole, modifie l'espace de noms de base d'une section de texte source. Elle doit être associée à la commande “end namespace” plus loin dans le code source pour marquer la fin de ce bloc. On peut ainsi réécrire l'exemple ci-dessus différemment :

```
space:
namespace space
	x = 1
	y = 2
	color:
	.r = 0
	.g = 0
	.b = 0
end namespace
```

Il convient de souligner que la commande “namespace” ne crée pas un nouvel espace de noms ; elle se contente d'entrer dans un espace de noms existant associé à un symbole donné.

Lorsqu'un symbole est spécifié comme argument d'une commande (telle que “namespace”) avec un nom non précédé d'un point, et donc sans indication explicite de l'espace de noms auquel il appartient, l'assembleur recherche le symbole défini dans l'espace de noms courant. Si aucun symbole défini portant ce nom n'est trouvé, il est supposé que le nom fait référence au symbole de l'espace de noms courant (et, sauf si le caractère “?” suit ce nom, la casse est prise en compte). En revanche, une définition qui ne spécifie pas l'espace de noms où le nouveau symbole doit être créé crée toujours un nouveau symbole dans l'espace de noms de base courant.

```
global = 0
regional = 1
namespace regional
	regional = 2		; regional.regional = 2
	x = global		; regional.x = 0
	regional.x = regional	; regional.regional.x = 2
	global.x = global	; global.x = 0
end namespace
```

Les commentaires correspondant à l'exemple ci-dessus indiquent des définitions équivalentes par rapport à l'espace de noms de base d'origine. Notez que lorsqu'un nom est utilisé pour spécifier l'espace de noms, l'assembleur recherche un symbole défini portant ce nom. Cependant, lorsqu'il s'agit du nom d'un symbole à définir, celui-ci est toujours créé dans l'espace de noms de base courant. En revanche, lorsque l'argument d'une commande “namespace” est le nom d'un symbole non défini, il fait également référence à un symbole de l'espace de noms courant.

Lorsque le point final d'un identifiant n'est suivi d'aucun nom, il fait référence au symbole parent de l'espace de noms dans lequel la recherche d'un symbole serait effectuée si un nom suivait ce point. L'ajout d'un tel point à la fin d'un identificateur peut sembler redondant, mais il permet de modifier le fonctionnement de la définition d'un symbole, car il oblige l'assembleur à rechercher un symbole existant qu'il peut modifier au lieu d'en créer un nouveau dans l'espace de noms courant. Par exemple, si, à la quatrième ligne de l'exemple précédent, “regional.” remplaçait “regional”, la valeur du symbole “regional” d'origine serait réécrite au lieu de créer un nouveau symbole dans l'espace de noms enfant. De même, une définition ainsi formée peut attribuer une nouvelle valeur à un symbole, qu'il ait été précédemment défini comme insensible à la casse ou non.

Si un identificateur ne contient qu'un seul point, selon les règles ci-dessus, il fait référence à l'étiquette la plus récente ne commençant pas par un point. Ceci peut être appliqué pour réécrire l'exemple précédent d'une autre manière :

```
space:
namespace .
```

```
        x = 1
        y = 2
        color:
        namespace .
                r = 0
                g = 0
                b = 0
        end namespace
end namespace
```

Cela montre également comment les sections d'espace de noms peuvent être imbriquées les unes dans les autres.

Le caractère “#” peut être inséré n'importe où dans un identifiant sans en modifier le sens. Lorsqu'il est le seul caractère séparant deux jetons de nom, ces derniers sont interprétés comme un seul nom formé par concaténation.

```
variable = 1
varia#ble = var#iable + 2       ; variable = 3
```

Ceci s'applique également aux nombres.

À l'intérieur d'un bloc défini avec “namespace”, aucune étiquette ne sert initialement de base aux identifiants commençant par un point (même si une étiquette remplissait cette fonction en dehors du bloc, elle perd ce statut et n'est réutilisée qu'après la fermeture du bloc avec “end namespace”). Un phénomène similaire se produit au début du texte source, avant la définition de toute étiquette. Ceci est lié à des règles supplémentaires concernant les points dans les identifiants.

Lorsqu'un identifiant commence par un point, mais qu'aucune étiquette ne lui est associée, l'identifiant fait référence au descendant d'un symbole spécial sans nom, présent dans l'espace de noms courant. Si un identifiant commence par une séquence de deux points ou plus, il fait référence au descendant d'un symbole similaire sans nom, mais ce symbole est différent pour chaque nombre de points. Alors que l'espace de noms accessible avec un seul point initial change à chaque définition d'une nouvelle étiquette, l'espace de noms spécial accessible avec deux points ou plus au début d'un identifiant reste inchangé.

```
first:
        .child = 1
        ..other = 0
second:
        .child = 2
        ..another = ..other
```

Dans cet exemple, la signification de l'identifiant “.child” varie selon l'endroit, tandis que celle de l'identifiant “.other” reste la même partout.

Lorsque deux noms à l'intérieur d'un identifiant sont reliés par une séquence de deux points ou plus, l'identifiant fait référence au descendant de ce symbole spécial sans nom dans l'espace de noms spécifié par l'identifiant partiel précédant cette séquence de points. L'espace de noms de l'enfant sans nom est choisi en fonction du nombre de points, qui est ici incrémenté de un. L'exemple suivant illustre les deux méthodes d'identification d'un tel symbole :

```
namespace base
        ..other = 1
end namespace

result = base.#..other
```

Le caractère “#” a été inséré dans le dernier identifiant pour une meilleure lisibilité, mais une simple séquence de trois points aurait le même effet.

Le symbole sans nom qui héberge un espace de noms spécial est accessible lorsqu'un identifiant se termine par une séquence de deux points ou plus. Ceci est dû à la règle selon laquelle un identifiant se terminant par un point fait référence au symbole parent de l'espace de noms auquel on accéderait s'il y avait un nom après ce point. Ainsi, dans l'exemple précédent, “base...” (ou “base.#..”) ferait référence au parent sans nom de l'espace de noms où se trouve le symbole “other”, et il s'agirait du même symbole que celui identifié par “..” à l'intérieur de l'espace de noms du symbole “base”.

Tout identifiant peut être précédé du caractère “?”. Ce modificateur a un effet lorsqu'il est utilisé dans un contexte où l'identifiant pourrait avoir une signification différente d'une étiquette ou d'une variable à définir. Ce modificateur supprime alors toute autre interprétation. Par exemple, un identifiant commençant par “?” ne sera pas interprété comme une instruction, même s'il s'agit du premier symbole de la ligne. Ceci peut servir à définir une variable portant le même nom qu'une commande existante :

```
?namespace = 0
```

Si un tel identifiant modifié est utilisé à un endroit où il est évalué et non défini, il fait toujours référence au même symbole que dans une définition. Par conséquent, à moins qu'il ne contienne également un point, l'identifiant fait toujours référence à un symbole du namespace courant.

Un nombre peut être utilisé comme nom à l'intérieur d'un identifiant, mais pas en début d'identifiant, car il est alors considéré comme une valeur littérale. Cette restriction peut être contournée en faisant précéder l'identifiant de "?".

```
in.1 = 3
namespace in
        ?2 = ?1 + 1     ; in.2 = 4
end namespace
```

# 3. Définitions des symboles de base

Lorsqu'un symbole est défini comme une étiquette, il doit s'agir de sa seule définition dans l'ensemble du code source. La valeur ainsi attribuée au symbole est accessible depuis n'importe quel endroit du code source, même avant que l'étiquette ne soit définie. Lorsqu'un symbole est utilisé avant d'être défini (on parle alors de référence anticipée), l'assembleur tente de prédire correctement sa valeur en parcourant le code source à plusieurs reprises. Ce n'est que lorsque toutes les prédictions sont correctes que l'assembleur génère le résultat final.

Ce type de symbole, qui ne peut être défini qu'une seule fois et possède donc une valeur universelle toujours accessible par référence anticipée, est appelé une constante. Toutes les étiquettes sont des constantes.

Lorsqu'un symbole est défini avec la commande "=", il peut avoir plusieurs définitions de ce type. Ce symbole est alors appelé une variable et, lorsqu'il est utilisé, c'est la valeur de sa dernière définition qui est accédée. Un symbole défini avec cette commande peut également faire l'objet d'une référence anticipée, mais uniquement s'il est défini exactement une fois dans l'ensemble du code source et possède donc une valeur unique et non ambiguë.

```
a = 1       ; a = 1
a = a + 1   ; a = 2
a = b + 1   ; a = 3
b = 2
```

Un cas particulier de référence anticipée est l'autoréférence, où la valeur d'un symbole est utilisée dans sa propre définition. L'assemblage d'une telle construction ne réussit que si l'assembleur parvient à trouver une valeur stable lors de cette évaluation, résolvant ainsi une équation. Cependant, en raison de la simplicité de l'algorithme de résolution basé sur des prédictions, une solution peut ne pas être trouvée, même si elle existe.

```
x = (x-1)*(x+2)/2-2*(x+1)       ; x = 6 or x = -1
```

Le symbole ":=" définit une valeur constante. Il peut être utilisé à la place de "=" pour garantir que le symbole donné est défini une seule fois et qu'il peut être référencé ultérieurement.

Le symbole "=:" définit un symbole variable comme "=", mais il diffère dans la façon dont il traite la valeur précédente (lorsqu'elle existe). Alors que "=" ignore la valeur précédente, "=:" la conserve afin qu'elle puisse être restaurée ultérieurement avec la commande "restore".

```
a = 1
a =: 2      ; préserve a = 1
a = 3       ; rejette a = 2 et le remplace par a = 3
restore a   ; ramène a = 1
```

Une commande "restore" peut être suivie de plusieurs identifiants de symboles séparés par des virgules et ignore la dernière définition de chacun d'entre eux. Ce n'est pas considéré comme une erreur d'utiliser "restore" avec un symbole qui n'a pas de définition active (soit parce qu'il n'a jamais été défini, soit parce que toutes ses définitions ont déjà été supprimées précédemment). Si un symbole est traité avec la commande "restore", il devient une variable et ne peut jamais être référencé vers l'avant. Pour cette raison, "restore" ne peut pas être appliqué aux constantes.

Le mot-clé "label" suivi d'un identifiant de symbole est une manière alternative de définir une étiquette. Dans cette forme de base, cela équivaut à une définition faite avec ":", mais il occupe une ligne entière. Cependant avec cette commande il est possible de fournir plus de paramètres pour l'étiquette définie. L'identifiant peut être éventuellement suivi du jeton ":" puis d'une valeur supplémentaire à associer à cette étiquette (indiquant généralement la taille de l'entité étiquetée). L'assembleur dispose d'un certain nombre de constantes intégrées définissant différentes tailles à cet effet, mais cette valeur peut également être fournie sous forme de nombre simple.

```
label character:byte
label char:1
```

Le caractère “:” peut être omis et remplacé par un simple espace, mais il est recommandé pour plus de clarté. Après un identifiant et une taille facultative, le mot-clé “at” peut suivre, puis une valeur qui sera attribuée à l'étiquette à la place de l'adresse actuelle.

```
label wchar:word at char
```

Les constantes de taille intégrées sont équivalentes à l'ensemble de définitions suivant :

```
byte? = 1       ; 8 bits
word? = 2       ; 16 bits
dword? = 4      ; 32 bits
fword? = 6      ; 48 bits
pword? = 6      ; 48 bits
qword? = 8      ; 64 bits
tbyte? = 10     ; 80 bits
tword? = 10     ; 80 bits
dqword? = 16    ; 128 bits
xword? = 16     ; 128 bits
qqword? = 32    ; 256 bits
yword? = 32     ; 256 bits
dqqword? = 64   ; 512 bits
zword? = 64     ; 512 bits
```

Le mot-clé “element” suivi d'un identifiant de symbole définit une constante spéciale qui n'a pas de valeur fixe et peut être utilisée comme variable dans les polynômes linéaires. L'identifiant peut être éventuellement suivi du jeton “:” puis d'une valeur à associer à ce symbole, appelée métadonnée de l'élément.

```
element A
element B:1
```

Les métadonnées associées à un symbole peuvent être extraites à l'aide d'un opérateur spécial, défini dans la section suivante.

# 4. Valeurs d'expression

Dans toutes les constructions décrites jusqu'à présent où une valeur était fournie, par exemple après la commande “=” ou le mot-clé “at”, il pouvait s'agir d'une valeur littérale (un nombre ou une chaîne de caractères entre guillemets) ou d'un identifiant symbolique. Une valeur peut également être spécifiée par une expression contenant des opérateurs intégrés.

Les opérateurs “+”, “-” et “*” effectuent des opérations arithmétiques standard sur les entiers (les opérateurs “+” et “-” peuvent également être utilisés sous une forme unaire, avec un seul argument). Les opérateurs “/” et “mod” effectuent une division avec reste, donnant respectivement un quotient ou un reste. Parmi ces opérateurs arithmétiques, “mod” est prioritaire (il est calculé en premier), suivi de “*” et “/”, tandis que “+” et “-” sont évalués en dernier (même sous leur forme unaire). Les opérateurs de même priorité sont évalués de gauche à droite. Les parenthèses permettent d'encadrer des sous-expressions lorsqu'un ordre d'opérations différent est requis.

Les opérateurs “xor”, “and” et “or” effectuent des opérations bit à bit sur les nombres. “xor” correspond à l'addition de bits (ou exclusif), “and” à la multiplication de bits et “or” à l'opération ou inclusif (disjonction logique). Ces opérateurs sont prioritaires sur tous les opérateurs arithmétiques.

Les opérateurs “shl” et “shr” décalent les bits du premier argument du nombre de bits spécifié par le second. “shl” décale les bits vers la gauche (vers les puissances de deux supérieures), tandis que “shr” les décale vers la droite (vers zéro), en ignorant les bits fractionnaires. Ces opérateurs sont prioritaires sur toutes les autres opérations bit à bit binaires.

Les opérateurs “not”, “bsf” et “bsr” sont des opérateurs unaires encore plus prioritaires. “not” inverse tous les bits d'un nombre, tandis que “bsf” et “bsr” recherchent respectivement le bit à 1 ou le bit à 1 le plus faible et renvoient l'indice de ce bit.

Toutes les opérations sur les nombres sont effectuées comme si elles portaient sur les représentations 2-adiques infinies de ces nombres. Par exemple, l'opération “bsr” avec un nombre négatif comme argument ne donne aucun résultat valide, car un tel nombre possède une chaîne infinie de bits à 1 s'étendant vers l'infini et ne contient donc aucun bit de poids fort à 1 (ceci est signalé comme une erreur).

L'opérateur “bswap” permet de créer une chaîne d'octets représentant un nombre en ordre d'octets inversé (big-endian). Son second argument doit être la longueur, en octets, de la chaîne souhaitée. Cet opérateur a la même priorité que les opérateurs “shl” et “shr”.

Lorsqu'une chaîne de caractères est utilisée comme argument d'une opération sur un nombre, elle est traitée comme une séquence de bits et automatiquement convertie en un nombre positif (complété par des zéros à l'infini). Les caractères consécutifs d'une chaîne correspondent aux bits de poids fort du nombre.

Pour reconvertir un nombre en chaîne de caractères, on peut utiliser l'opérateur unaire “string”. Cet opérateur a la priorité la plus basse ; ainsi, s'il précède une expression, celle-ci est évaluée en entier avant la conversion. Pour effectuer la conversion inverse, l'opérateur unaire “+” suffit. La longueur d'une chaîne de caractères s'obtient grâce à l'opérateur unaire “lengthof”, qui fait partie des opérateurs prioritaires.

L'opérateur “bappend” ajoute une séquence d'octets de la chaîne fournie par le second argument à la séquence d'octets fournie par le premier. Si l'un des arguments est un nombre, il est implicitement converti en chaîne de caractères. Cet opérateur a la même priorité que les opérations binaires bit à bit.

Lorsqu'un symbole défini avec la commande “element” est utilisé dans une expression, le résultat peut être un polynôme linéaire en la variable représentée par ce symbole. Seules les opérations arithmétiques simples sont autorisées sur les termes d'un polynôme, et celui-ci doit rester linéaire ; par exemple, il est uniquement possible de multiplier un polynôme par un nombre, mais pas par un autre polynôme.

Quelques opérateurs prioritaires permettent d'extraire des informations sur les termes d'un polynôme linéaire. Le polynôme doit être passé en premier argument, et l'indice du terme en second. L'opérateur “element” extrait la variable d'un terme polynomial (dont le coefficient est égal à un), l'opérateur “scale” extrait le coefficient (le nombre par lequel la variable est multipliée) et l'opérateur “metadata” renvoie les métadonnées associées à la variable.

Si le second argument est un indice supérieur à celui du dernier terme du polynôme, les trois opérateurs renvoient zéro. Lorsque le deuxième argument est nul, “element” et “scale” donnent des informations sur le terme constant : “element” renvoie la valeur numérique 1 et “scale” renvoie la valeur du terme constant.

```
element A
linpoly = A + A + 3
vterm = linpoly scale 1 * linpoly element 1     ; vterm = 2 * A
cterm = linpoly scale 0 * linpoly element 0     ; cterm = 3 * 1
```

L'opérateur “metadata” avec un indice zéro renvoie la taille associée au premier argument. Cette valeur est définie uniquement si le premier argument est un symbole auquel est associée une taille (ou une expression arithmétique contenant un tel symbole) ; sinon, elle est nulle. Il existe un opérateur unaire supplémentaire, “sizeof”, qui renvoie la même valeur que “metadata 0”.

```
label table : 256
length = sizeof table   ; length = 256
```

Les opérateurs “elementof”, “scaleof” et “metadataof” sont des variantes des opérateurs “element”, “scale” et “metadata”, dont l'ordre des arguments est inversé. Ainsi, l'utilisation de “sizeof” dans une expression équivaut-elle à écrire “0 metadataof” en lieu et place. Ces opérateurs sont encore plus prioritaires que leurs homologues et sont associatifs à droite.

L'ordre des termes d'un polynôme linéaire dépend de la manière dont sa valeur a été construite. Toute opération arithmétique préserve l'ordre des termes du premier argument, et les termes absents du premier argument sont ajoutés à la fin dans le même ordre que celui du second argument. Cet ordre n'a d'importance que lors de l'extraction de termes à l'aide des opérateurs appropriés.

L'opérateur “elementsof” est un autre opérateur unaire de priorité maximale ; il compte le nombre de termes variables d'un polynôme linéaire.

Une expression peut également contenir une valeur littérale définissant un nombre à virgule flottante. Ce nombre doit être en notation décimale ; il peut contenir le caractère “.” comme séparateur décimal et être suivi du caractère “e”, puis de la valeur décimale de l’exposant (éventuellement précédé de “+” ou “–” pour indiquer le signe de l’exposant). Lorsqu’un “.” ou un “e” est présent, il doit être suivi d’au moins un chiffre. Le caractère “f” peut être ajouté à la fin de cette valeur littérale. Si un nombre ne contient ni “.” ni “e”, le “f” final est le seul moyen de garantir qu’il est traité comme un nombre à virgule flottante et non comme un simple entier décimal.

Les nombres à virgule flottante sont traités par l’assembleur sous forme binaire. Leur plage et leur précision sont au moins égales à celles du format à virgule flottante le plus long que l’assembleur est capable de produire en sortie.

Les opérations arithmétiques de base peuvent prendre un nombre à virgule flottante comme argument, à condition qu'aucun argument ne contienne de terme non scalaire (polynôme linéaire). Le résultat de ces opérations est toujours un nombre à virgule flottante.

L'opérateur unaire “float” permet de convertir un entier en nombre à virgule flottante. Cet opérateur est prioritaire.

L'opérateur “trunc” est un opérateur unaire de priorité maximale, applicable aux nombres à virgule flottante. Il extrait la partie entière d'un nombre (troncature vers zéro) et le résultat est toujours un entier, et non un nombre à virgule flottante. Si l'argument est déjà un entier, il demeure inchangé.

L'opérateur “bsr” s'applique aux nombres à virgule flottante et renvoie l'exposant de ce nombre, c'est-à-dire l'exposant de la plus grande puissance de deux inférieure ou égale à celle du nombre donné. Le signe du nombre à virgule flottante est sans incidence sur le résultat.

Il est également possible d'utiliser un nombre à virgule flottante comme premier argument des opérateurs “shl” et “shr”. Le nombre est alors multiplié ou divisé par la puissance de 2 spécifiée par le second argument.

## 5. Classes de symboles

Il existe trois classes distinctes de symboles, qui déterminent leur position dans la ligne source. Un symbole de la classe des instructions est reconnu uniquement lorsqu'il est le premier identifiant de la commande, tandis qu'un symbole de la classe des expressions est reconnu uniquement lorsqu'il sert à fournir la valeur d'un argument à une commande.

Tous les types de définitions décrits précédemment créent des symboles de la classe des expressions. Les symboles “label” et “restore” sont des exemples de symboles intégrés appartenant à la classe des instructions.

Dans un espace de noms donné, des symboles de classes différentes peuvent porter le même nom. Par exemple, il est possible de définir l'instruction nommée “shl”, alors qu'il existe également un opérateur portant le même nom, mais cet opérateur appartient à la classe des expressions.

Il est même possible qu'une même ligne contienne le même identifiant ayant des significations différentes selon sa position.

```
?restore = 1
restore restore    ; suppression de la valeur du symbole de classe d'expression
```

La troisième catégorie de symboles est celle des instructions étiquetées. Un symbole appartenant à cette catégorie ne peut être reconnu que si le premier identifiant de la commande n'est pas une instruction ; dans ce cas, le premier identifiant devient une étiquette de l'instruction définie par le second. Si l'on considère “=” comme un identifiant particulier, il peut servir d'exemple d'instruction étiquetée.

L'assembleur contient des symboles intégrés de toutes les classes. Leurs noms ne sont jamais sensibles à la casse et ils peuvent être redéfinis, mais il est impossible de les supprimer. Lorsque toutes les valeurs d'un tel symbole sont supprimées avec une commande comme “restore”, la valeur intégrée est conservée.

Les règles relatives aux espaces de noms s'appliquent de la même manière aux symboles de toutes les classes. Par exemple, un symbole de la classe d'instructions appartenant à l'espace de noms enfant de la dernière étiquette peut être exécuté en faisant précéder son nom d'un point. Il convient de noter, cependant, que lorsqu'un espace de noms est spécifié par son symbole parent, il s'agit toujours d'un symbole appartenant à la classe d'expressions. Il n'est pas possible de faire référence à un espace de noms enfant d'une instruction, seulement à l'espace de noms appartenant au symbole de classe d'expression portant le même nom.

```
xor?.mask? := 10101010b
a = XOR.MASK      ; symbole dans le namespace de l'opérateur "XOR" intégré
                  ; insensible à la casse

label?.test? := 0
a = LABEL.TEST    ; indéfini sauf si  “label?”  est défini
```

Ici, l'espace de noms contenant “test” appartient à un symbole de classe d'expression, et non à l'instruction existante “label”. Lorsqu'aucun symbole de classe d'expression ne correspond au spécificateur “LABEL”, l'espace de noms choisi est celui qui appartiendrait au symbole portant ce nom (sensible à la casse). “test” est donc introuvable, car il a été défini dans un autre espace de noms : celui de l'instruction insensible à la casse “label”.

## 6. Génération de données

L'instruction “db” permet de générer des octets de données et de les afficher en sortie. Elle doit être suivie d'une ou plusieurs valeurs, séparées par des virgules. Si la valeur est numérique, elle définit un octet. Si la valeur est une chaîne de caractères, elle affiche cette chaîne d'octets en sortie.

```
db 'Hello',13,10    ; génère 7 octets
```

Le mot-clé “dup” permet de dupliquer plusieurs fois la même valeur. Il doit être précédé d'une expression numérique indiquant le nombre de répétitions, puis suivi de la valeur à répéter. Une séquence de valeurs peut également être dupliquée de cette manière ; dans ce cas, “dup” doit être suivi de la séquence complète entre parenthèses (les valeurs étant séparées par des virgules).

```
db 4 dup 90h          ; génère 4 octets
db 2 dup ('abc',10)   ; génère 8 octets
```

Lorsqu'un identifiant spécial composé du seul caractère “?” est utilisé comme argument de la commande “db”, un octet est réservé. Cela incrémente l'adresse de sortie où les données suivantes seront écrites, mais les octets réservés ne sont pas générés eux-mêmes, sauf s'ils sont suivis d'autres données. Par conséquent, si les octets sont réservés à la fin de la sortie, ils n'augmentent pas la taille du fichier généré. Ces données sont dites *non initialisées*, tandis que toutes les données classiques sont dites *initialisées*.

L'instruction “rb” réserve le nombre d'octets spécifié par son argument.

```
db ?        ; réserve 1 octet
rb 7        ; réserve 7 octets
```

Chaque instruction intégrée qui génère des données (traditionnellement appelée directive de données) est associée à une instruction étiquetée du même nom. Une telle commande, en plus de générer des données, définit une étiquette à l'adresse des données générées, avec une taille associée égale à la taille de l'unité de données utilisée par cette instruction. Dans le cas de “db” et “rb”, cette taille est 1.

```
some db sizeof some    ; génération d'un octet de valeur 1
```

Les instructions “dw”, “dd”, “dp”, “dq”, “dt”, “ddq”, “dqq” et “ddqq” sont analogues à “db” et utilisent des unités de données de tailles différentes. L'ordre des octets au sein d'une unité de données est toujours petit-boutiste (little-endian). Lorsqu'une chaîne d'octets est fournie comme valeur à l'une de ces instructions, les données générées sont complétées par des octets nuls jusqu'à une longueur multiple de l'unité de données. Les instructions “rw”, “rd”, “rp”, “rq”, “rt”, “rdq”, “rqq” et “rdqq” réservent un nombre spécifié d'unités de données. Les tailles d'unités associées à ces instructions sont indiquées dans le tableau 1.

Les instructions “dw”, “dd”, “dq”, “dt” et “ddq” acceptent les nombres à virgule flottante comme unités de données. Chaque nombre est ensuite converti au format à virgule flottante approprié à la taille donnée.

La directive “emit” (avec “dbx”, pour synonyme) est une directive de données qui utilise la taille de l'unité spécifiée par son premier argument pour générer des données définies par les autres. La taille peut être séparée de l'argument suivant par deux points au lieu d'une virgule, pour une meilleure lisibilité. Lorsque la taille de l'unité est telle qu'elle possède une directive de données dédiée, la définition faite avec “emit” a le même effet que si ces valeurs étaient transmises à l'instruction adaptée à cette taille.

```
emit 2: 0,1000,2000    ; génération de 3 valeurs de 16 bits
```

L'instruction “file” lit les données d'un fichier externe et les écrit dans le flux de sortie. L'argument doit être une chaîne de caractères contenant le chemin d'accès au fichier, éventuellement suivie de “:” et d'une valeur numérique indiquant un décalage dans le fichier, puis éventuellement d'une virgule et d'une valeur numérique indiquant le nombre d'octets à copier.

```
file 'data.bin'                ; insertion de la totalité du fichier
excerpt file 'data.bin':10h,4  ; insertion des quatre octets sélectionnés
```

**Tableau 1 – Directives de données**

| Taille (octets) | Génération de données | Réservation de données |
|---|---|---|
| 1 | db<br>file | rb |
| 2 | dw | rw |
| 4 | dd | rd |
| 6 | dp | rp |
| 8 | dq | rq |
| 10 | dt | rt |
| 16 | ddq | rdq |
| 32 | dqq | rqq |
| 64 | ddqq | rdqq |
| * | emit | |

## 7. Assemblage conditionnel

L'instruction “if” permet d'assembler un bloc de texte source uniquement sous certaines conditions, spécifiées par une expression logique qui lui est passée en argument. La commande “else if” dans les lignes suivantes termine le bloc précédent, assemblé conditionnellement, et en ouvre un nouveau, assemblé uniquement si les conditions précédentes n'étaient pas remplies et que la nouvelle condition (un argument de “else if”) est vraie. La commande “else” termine le bloc précédent, assemblé conditionnellement, et commence un bloc qui ne sera assemblé que si aucune des conditions précédentes n'était vraie. La commande “end if” doit être utilisée pour terminer la construction. Il peut y avoir plusieurs commandes “else if” ou aucune, et au maximum une commande “else”.

Une expression logique est une entité syntaxique distincte des expressions de base décrites précédemment. Elle est composée de valeurs logiques reliées par des opérateurs logiques. Les opérateurs logiques sont : l'opérateur

unaire “~” pour la négation, “&” pour la conjonction et “|” pour l'alternative. La négation est évaluée en premier, tandis que “&” et “|” le sont également. Les valeurs logiques sont évaluées de gauche à droite, sans priorité les unes sur les autres.

Une valeur logique, dans sa forme la plus simple, peut être une expression de base. Elle correspond alors à la condition vraie si et seulement si sa valeur est différente de zéro. Une autre façon de créer une valeur logique consiste à comparer les valeurs de deux expressions de base à l'aide de l'un des opérateurs suivants : “=” (égal), “<” (inférieur à), “>” (supérieur à), “<=” (inférieur ou égal à), “>=” (supérieur ou égal à), “<>” (différent de).

```
count = 2
if count > 1
      db '0'
      db count-1 dup ',0'
else if count = 1
      db '0'
end if
```

Lorsqu'on compare des polynômes linéaires de cette manière, la valeur logique n'est valide que s'ils sont comparables, c'est-à-dire s'ils ne diffèrent que par leur terme constant. Autrement, la condition d'égalité n'est ni universellement vraie ni universellement fausse, car elle dépend des valeurs substituées aux variables, et l'assembleur signale cela comme une erreur.

L'opérateur “relativeto” crée une valeur logique vraie uniquement lorsque la différence des valeurs comparées ne contient aucun terme variable. Il peut donc être utilisé pour vérifier si deux polynômes linéaires sont comparables : la condition “relativeto” est vraie uniquement si les deux polynômes comparés ont les mêmes termes variables.

Comme les expressions logiques sont évaluées paresseusement, il est possible de créer une condition unique qui ne provoquera pas d'erreur lorsque les polynômes ne sont pas comparables, mais qui les comparera s'ils le sont.

```
if a relativeto b & a > b
      db a - b
end if
```

L'opérateur “eqtype” peut également être utilisé pour comparer deux expressions de base, il crée une valeur logique qui est vraie lorsque les valeurs des expressions sont du même type – soit les deux sont algébriques, soit les deux sont des chaînes, soit les deux sont des nombres à virgule flottante. Un type algébrique couvre les polynômes linéaires et inclut les valeurs entières.

L'opérateur “eq” compare deux expressions de base et crée une valeur logique qui n'est vraie que lorsque leurs valeurs sont du même type et égales. Cet opérateur peut être utilisé pour vérifier si une valeur est une certaine chaîne, un certain nombre à virgule flottante ou un certain polynôme linéaire. Il peut comparer des valeurs qui ne sont pas comparables avec l'opérateur “=”.

L'opérateur “defined” crée une valeur logique combinée à une expression de base qui la suit. Cette condition est vraie lorsque l'expression ne contient pas de symboles n'ayant pas de définition accessible. L'expression est uniquement testée pour la disponibilité de ses composants, elle n'a pas besoin d'avoir une valeur calculable. Cela peut être utilisé pour vérifier si un symbole de classe d'expression a été défini, mais comme le symbole peut être accessible via un référencement direct, cette condition peut être vraie même lorsque le symbole est défini ultérieurement dans la source. Si cela n'est pas souhaitable, l'opérateur “defined” doit être utilisé à la place, car il vérifie si tous les symboles d'une expression de base qui suit ont été définis précédemment.

L'expression de base qui suit “defined” peut également être vide ; la condition est alors trivialement satisfaite. Ceci ne s'applique pas à “definite”.

L'opérateur “used” produit une valeur logique s'il est suivi d'un seul identifiant. Cette condition est vraie lorsque la valeur du symbole spécifié a été utilisée quelque part dans le code source.

L'instruction “assert” signale une erreur lorsqu'une condition spécifiée par son argument n'est pas remplie.

```
assert a < 65536
```

# 8. Macro-instructions

La commande “macro” permet de définir une nouvelle instruction sous forme de macro-instruction. Le bloc de texte source situé entre les commandes “macro” et “end macro” constitue le texte de la macro-instruction, et cette séquence de lignes remplace la commande d'origine commençant par l'identifiant de l'instruction ainsi définie.

```
macro null
      db 0
end macro

null        ; correspond à l'assemblage de "db 0"
```

Une macro-instruction ne peut avoir d'arguments que si sa définition en contient. Après le mot “macro” et l'identifiant du symbole défini, une liste de noms simples séparés par des virgules peut être ajoutée. Ces noms définissent les paramètres de la macro-instruction. Lors de son utilisation, cette instruction peut être suivie d'au plus autant d'arguments, également séparés par des virgules, dont les valeurs sont affectées aux paramètres successifs. Avant l'interprétation de toute ligne de texte à l'intérieur de la macro-instruction, les noms correspondant aux paramètres sont remplacés par leurs valeurs.

```
macro lower name,value
      name = value and 0FFh
end macro

lower a,123h     ; a = 23h
```

La valeur d'un paramètre peut être n'importe quel texte, pas nécessairement une expression correcte. Si une ligne appelant la macro-instruction contient moins d'arguments que le nombre de paramètres définis, les paramètres excédentaires reçoivent une valeur vide.

Lorsqu'un nom de paramètre est défini, il peut être suivi du caractère “?” pour indiquer qu'il n'est pas sensible à la casse, comme pour un identifiant de symbole. Aucun espace ne doit séparer le nom et le caractère “?”. La définition d'un paramètre peut également être suivie d'un astérisque (*) pour indiquer qu'il requiert une valeur non vide, ou encore du caractère “:” suivi d'une valeur par défaut, qui est attribuée au paramètre à la place d'une valeur vide lorsqu'aucune autre valeur n'est fournie.

```
macro prepare name*,value:0
      name = value
end macro

prepare x        ; x = 0
prepare y,1      ; y = 1
```

Si un argument d'une macro-instruction doit contenir une virgule, l'argument entier doit être placé entre les caractères “<” et “>” (ils ne font pas partie de la valeur). Si un autre caractère “<” est rencontré à l'intérieur d'une telle valeur, il doit être équilibré avec le caractère “>” correspondant à l'intérieur de la même valeur.

```
macro data name,value
      name:
      .data db value
      .end:
end macro

data example, <'abc',10>
```

Le dernier paramètre défini peut être suivi du caractère “&” pour indiquer qu'il doit recevoir la valeur correspondant à la totalité du reste de la ligne, même s'il définit normalement plusieurs arguments. Ainsi, lorsqu'une macro-instruction ne comporte qu'un seul paramètre suivi de “&”, la valeur de ce paramètre correspond à l'intégralité des arguments qui suivent l'instruction.

```
macro id first,rest&
      dw first
      db rest
end macro

id 2, 7,1,8
```

Lorsque le nom d'un paramètre doit être remplacé par sa valeur et qu'il est précédé du caractère “`” (sans espace entre les deux), le texte de la valeur est intégré dans une chaîne entre guillemets et cette chaîne remplace à la fois le caractère “`” et le nom du paramètre.

```
macro text line&
      db `line
end macro

text x+1     ; db 'x+1'
```

La commande “local” ne peut être utilisée qu'à l'intérieur d'une macro-instruction. Elle doit être suivie d'un ou plusieurs noms séparés par des virgules et indique que, dans le contexte de la macro-instruction courante, les noms de cette liste doivent être interprétés comme appartenant à un espace de noms spécifique à cette macro-instruction, et non à l'espace de noms de base courant. Ceci permet de créer des symboles uniques à chaque appel de la macro-instruction. Cette déclaration définit des paramètres supplémentaires avec les noms spécifiés et n'affecte donc que les occurrences de ces noms qui suivent dans la même macro-instruction. Déclarer le même nom comme “local” plusieurs fois dans la même macro-instruction est sans effet.

```
macro measured name,string
      local top
      name db string
      top: name.length = top - name
end macro

measured hello, 'Hello!'       ; hello.length = 6
```

Un paramètre créé avec l'option “local” est remplacé par un texte portant le même nom que le paramètre, mais enrichi d'informations contextuelles permettant de l'identifier comme appartenant à l'espace de noms local unique associé à l'instance de macro-instruction. Ce type d'information contextuelle sera abordé plus en détail dans la section consacrée aux variables symboliques.

Un symbole local à une macro-instruction n'est jamais considéré comme l'étiquette de base la plus récente pour les symboles commençant par un point. De plus, son espace de noms descendant est déconnecté de l'arbre principal des symboles. Par conséquent, si la commande “namespace” est utilisée avec un symbole local comme argument, les symboles de l'arbre principal ne seront plus visibles (y compris toutes les instructions nommées de l'assembleur, même les commandes telles que “end namespace”).

De même qu'un symbole d'expression peut être redéfini et faire référence à sa valeur précédente dans la nouvelle définition, les macro-instructions peuvent également être redéfinies et utiliser la valeur précédente de ce symbole d'instruction dans leur texte :

```
macro zero
      db 0
end macro

macro zero name
      label name:byte
      zero
end macro

zero x
```

Comme tout autre symbole, une macro-instruction peut être référencée ultérieurement si elle est définie une seule fois dans le code source.

La commande “purge” supprime la définition d'un symbole, tout comme “restore”, mais pour les symboles de la classe “instruction”. Son comportement est identique à celui de “restore” pour le reste. Une macro-instruction peut supprimer sa propre définition avec “purge”.

Une macro-instruction peut utiliser sa propre valeur de manière récursive, mais pour éviter une récursion infinie involontaire, cette fonctionnalité n'est disponible que si la macro-instruction est marquée comme telle par le caractère “:” faisant suivre son identifiant.

```
macro factorial: n
      if n
            factorial n-1
            result = result * (n)
      else
            result = 1
      end if
end macro
```

Outre la possibilité d'utiliser la récursivité, une telle macro-instruction se comporte comme une constante. Elle ne peut être redéfinie et la fonction “purge” ne peut lui être appliquée.

Une macro-instruction peut à son tour définir une autre macro-instruction, ou plusieurs. Les blocs désignés par “macro” et “end macro” doivent être correctement imbriqués l'un dans l'autre pour que cette définition soit acceptée par l'assembleur.

```
macro enum enclosing
      counter = 0
      macro item name
            name := counter
            counter = counter + 1
      end macro
      macro enclosing
            purge item,enclosing
      end macro
```

```
end macro

enum x
      item a
      item b
      item c
x
```

Lorsqu'il est nécessaire qu'une macro-instruction génère une commande “macro” ou “end macro” non appariée, cela peut se faire grâce à l'instruction spéciale “esc”. Son argument fait alors partie de la macro-instruction, mais n'est pas pris en compte lors du comptage des paires “macro” et “end macro” imbriquées.

```
macro xmacro name
      esc macro name x&
end macro

xmacro text
      db ‘x
end macro
```

Si “esc” est placé à l'intérieur d'une définition imbriquée, il n'est traité que lorsque la macro-instruction la plus interne est définie. Cela permet d'insérer une définition contenant “esc” dans une autre macro-instruction sans avoir à répéter “esc” à chaque niveau d'imbrication.

Lorsqu'un identifiant de macro-instruction dans sa définition est suivi du caractère “!”, il s'agit d'une macro-instruction inconditionnelle. Ce type particulier de symbole de classe d'instruction est évalué même lorsque l'exécution de l'assembleur est suspendue, par exemple à l'intérieur d'un bloc conditionnel dont la condition est fausse ou à l'intérieur de la définition d'une autre macro-instruction. Cela permet de définir des instructions utilisables là où un “end if” ou un “end macro” explicite serait autrement requis, comme dans l'exemple suivant :

```
macro proc name
      name:
      if used name
end macro

macro endp!
      end if
      .end:
end macro

proc tester
      db ?
endp
```

Si la macro-instruction “endp” dans l'exemple ci-dessus n'était pas définie comme inconditionnelle et que le bloc commençant par “if” était ignoré, la macro-instruction ne serait pas évaluée, ce qui entraînerait une erreur car “end if” serait manquant.

Il convient de noter que la commande “end” exécute une instruction identifiée par son argument dans l'espace de noms enfant du symbole “end” insensible à la casse. Par conséquent, une commande telle que “end if” pourrait être invoquée alternativement avec un identifiant “end.if”, et il est possible de remplacer une telle instruction en redéfinissant un symbole dans le champ “end?” espace de noms. De plus, toute instruction définie dans la l'espace de noms “end?” peut ensuite être appelé avec la commande “end”. Cette variante légèrement modifiée de l’exemple ci-dessus met à profit ces faits :

```
macro proc name
      name:
      if used name
end macro

macro end?.proc!
      end if
      .end:
end macro

proc tester
      db ?
end proc
```

Une règle similaire s'applique à la commande “else” et aux instructions de l'espace de noms “else?”.

Lorsqu'un identifiant composé d'un seul caractère “?” est utilisé comme symbole d'instruction dans la définition d'une macro-instruction, il définit une instruction spéciale qui est alors appelée chaque fois qu'une ligne à assembler ne contient pas d'instruction inconditionnelle, et le texte complet de la ligne devient l'argument de cette macro-instruction. Ce symbole spécial peut également être défini comme une instruction inconditionnelle, auquel cas il est appelé pour chaque ligne suivante sans exception. Cela permet de remplacer complètement le processus d'assemblage sur des portions de texte. L'exemple suivant définit une macro-instruction qui permet de définir un bloc de commentaires en ignorant toutes les lignes de texte jusqu'à ce qu'elle rencontre une ligne dont le contenu correspond à l'argument passé à “comment”.

```
macro comment? ender
      macro ?! line&
            if ‘line = ‘ender
                  purge ?
            end if
      end macro
end macro

comment ~
      Any text may follow here.
~
```

Un identifiant composé de deux points d'interrogation permet de définir une instruction spéciale appelée en dernier recours, sur les lignes ne contenant aucune instruction reconnaissable. Ceci permet d'intercepter des lignes qui seraient autrement rejetées avec le message “instruction illégale” en raison d'une syntaxe inconnue.

L'instruction “mvmacro” prend deux arguments, identifiant chacun un symbole de classe d'instruction. La définition d'une macro-instruction spécifiée par le second argument est déplacée vers le symbole identifié par le premier. Pour le second symbole, l'effet de cette commande est identique à celui de “purge”. Ceci permet de renommer une macro-instruction, ou de la désactiver temporairement pour la réactiver ultérieurement. Les symboles affectés par cette opération deviennent des variables et ne peuvent plus être référencés.

## 9. Macro-instructions étiquetées

La commande “struc” permet de définir une instruction étiquetée, sous forme de macro-instruction. À l'exception du fait que cette définition doit se terminer par “end struc” au lieu de “end macro”, ces macro-instructions sont définies de la même manière qu'avec la commande “macro”. Une instruction étiquetée est évaluée lorsque le premier identifiant de la commande n'est pas une instruction et que le second identifiant appartient à la classe des instructions étiquetées.

```
struc some
      db 1
end struc

get some     ; correspond à l’assemblage de "db 1"
```

À l'intérieur d'une macro-instruction étiquetée, les identifiants commençant par un point ne font plus référence à l'espace de noms d'une étiquette régulière précédemment définie. Au lieu de cela, ils font référence à l'espace de noms de l'étiquette avec lequel l'instruction a été étiquetée.

```
struc POINT
      label . : qword
      .x dd ?
      .y dd ?
end struc

my POINT     ; defines my.x and my.y
```

Notez que le symbole parent, qui peut être référencé par “.” identifiant, n'est défini que si une définition appropriée est générée par la macro-instruction. De plus, ce symbole n'est pas considéré comme l'étiquette la plus récente dans l'espace de noms environnant à moins qu'il ne soit défini comme une étiquette réelle dans la macro-instruction qu'il a étiquetée.

Pour une utilisation plus facile de cette fonctionnalité, d'autres syntaxes peuvent être définies avec des macro-instructions, comme dans cet exemple :

```
macro struct? definition&
      esc struc definition
            label . : .%top - .
```

```
            namespace .
end macro

macro ends?!
                  %top:
            end namespace
      esc end struc
end macro

struct POINT vx:?,vy:?
      x dd vx
      y dd vy
ends

my POINT 10,20
```

La commande “restruc” est analogue à “purge”, mais elle opère sur des symboles de la classe des instructions étiquetées. De même, la commande “mvstruc” est la même que “mvmacro” mais pour les instructions étiquetées.

Comme pour “macro”, il est possible d'utiliser un identifiant constitué d'un seul “?” caractère avec “struc”. Il définit une macro-instruction spéciale étiquetée qui est appelée chaque fois que le premier symbole d'une ligne n'est pas reconnu comme une instruction. Tout ce qui suit ce premier identifiant devient les arguments de la macro-instruction étiquetée. L’exemple suivant utilise cette fonctionnalité pour détecter toutes les étiquettes orphelines (celles qui ne sont suivies d’aucun caractère) et les traiter comme des étiquettes normales au lieu de provoquer une erreur. Il y parvient en faisant de “: la valeur par défaut du paramètre “def” :

```
struc ? def::&
      . def
end struc

orphan
regular:
assert orphan = regular
```

À l'instar de “macro”, cette variante spéciale ne remplace pas les instructions étiquetées inconditionnelles, sauf si elle est elle-même inconditionnelle.

Bien que “.” permette d'accéder efficacement au symbole d'étiquette, il peut parfois être nécessaire de traiter le texte même de l'étiquette. Un paramètre spécifique peut être défini à cet effet ; son nom doit être inséré entre parenthèses avant le nom de la macro-instruction étiquetée.

```
struc (name) SYMBOL
      . db `name,0
end struc

test SYMBOL
```

# 10. Variables symboliques et contexte de reconnaissance

Le “equ” est une instruction étiquetée intégrée qui définit le symbole de la classe d'expression avec une valeur symbolique. Une telle valeur contient un extrait de texte source constitué d'un nombre quelconque de jetons (même zéro, autorisant une valeur vide) et lorsqu'elle est utilisée dans une expression, cela équivaut à insérer le texte de sa valeur à la place de son identifiant, avec un effet similaire à l'évaluation d'un paramètre de macro-instruction (sauf qu'un paramètre est toujours identifié par un nom unique, alors qu'une valeur symbolique peut être cachée derrière un identifiant complexe).

Cela peut conduire à un résultat inattendu par rapport à l'utilisation de variables standards définies avec “=”, comme le montre l'exemple suivant :

```
numeric = 2 + 2
symbolic equ 2 + 2
x = numeric*3      ; x = 4*3
y = symbolic*3     ; y = 2 + 2*3
```

Alors que “x” se voit attribuer la valeur 12, la valeur de “y” est 8. Ceci montre que l'utilisation de tels symboles peut entraîner des interactions inattendues et que, par conséquent, les définitions de ce type doivent être évitées sauf en cas de nécessité absolue.

La commande “equ” permet les redéfinitions et conserve la valeur précédente du symbole, de manière analogue à la commande “=:”. Ainsi, la valeur précédente peut être rétablie avec l'instruction “restore”. Pour remplacer la valeur symbolique (de la même manière que “=” écrase la valeur classique), il convient d'utiliser la commande “reequ” au lieu de “equ”.

Une valeur symbolique, en plus de conserver le texte exact avec lequel elle a été définie, préserve le contexte dans lequel les symboles contenus dans ce texte doivent être interprétés. Elle peut donc constituer un lien fiable vers la valeur d'un autre symbole, même lorsqu'elle est utilisée dans un contexte différent (y compris en cas de changement d'espace de noms de base ou de symbole référencé par un point initial).

```
first:
      .x = 1
      link equ .x
      .x = 2
second:
      .x = 3
      db link       ; db 2
```

Il convient de noter que le même processus s'applique aux arguments de toute macro-instruction lorsqu'ils deviennent des paramètres prétraités. Si, pendant l'exécution d'une macro-instruction, le contexte change, les identifiants dans le texte des paramètres font toujours référence aux mêmes symboles que dans la ligne qui a appelé l'instruction :

```
x = 1
namespace x
      x = 2
end namespace
macro prodx value
      namespace x
            db value*x
      end namespace
end macro
prodx x      ; db 1*2
```

De plus, les paramètres définis avec la commande “local” utilisent le même mécanisme pour modifier le contexte d'interprétation de leur nom, sans en altérer le texte. Cependant, ce contexte modifié est sans importance si la valeur du paramètre est insérée au milieu ou à la fin d'un identifiant complexe, car c'est la structure de l'identifiant qui détermine l'interprétation de ses parties suivantes ; seul le contexte de la partie initiale compte. Par exemple, si le nom d'un paramètre est précédé du caractère “#”, l'identifiant utilisera le contexte actuel plutôt que celui du texte du paramètre, car le contexte initial de l'identifiant est alors celui associé au caractère “#”.

Si le texte suivant “equ” contient des identifiants de variables symboliques connues, chacun d'eux est remplacé par son contenu, et c'est ce texte ainsi traité qui est affecté au symbole nouvellement défini.

L'instruction “define” est une instruction classique qui crée également une valeur symbolique, mais contrairement à “equ”, elle n'évalue pas les variables symboliques présentes dans le texte affecté. Elle doit être suivie de l'identifiant du symbole à définir, puis du texte de sa valeur.

La différence entre “equ” et “define” est souvent imperceptible, car dans une expression finale, les variables symboliques sont évaluées de manière imbriquée jusqu'à ce qu'il ne reste que les éléments utilisables des expressions. “define” peut notamment servir à créer un lien vers une autre variable symbolique, comme le montre l'exemple suivant :

```
a equ 0*
x equ -a
define y -a
a equ 1*
db x 2    ; db -0*2
db y 2    ; db -1*2
```

Les autres utilisations de “define” seront abordées dans les sections suivantes, avec l'introduction d'autres instructions agissant sur les valeurs symboliques.

L'instruction “define”, tout comme “equ”, conserve la valeur précédente du symbole. L'instruction “redefine” est une variante qui supprime la valeur précédente, de manière analogue à “reequ”.

Il est important de noter que, bien que les variables symboliques appartiennent à la classe des symboles d'expression, leur état ne peut être déterminé par des opérateurs tels que “defined”, “definite” ou “used”, car une expression logique est évaluée comme si chaque variable symbolique était remplacée par le texte de sa valeur correspondante. Par conséquent, un opérateur suivi de l'identifiant d'une variable symbolique s'appliquera au contenu de cette variable, quel qu'il soit. Par exemple, si une variable symbolique est créée et qu'elle est un lien vers

un symbole classique, alors tout opérateur tel que “defined” suivi de l'identifiant de cette variable symbolique déterminera l'état du symbole lié, et non celui de la variable de liaison.

Contrairement à la valeur d'une variable symbolique, le corps d'une macro-instruction ne contient aucun contexte (bien qu'il puisse inclure des fragments de texte provenant de paramètres remplacés et qui, de ce fait, possèdent un certain contexte). De plus, si une macro-instruction est déroulée lors de la définition d'une autre (ce qui ne peut se produire que si la macro-instruction appelée est inconditionnelle), aucune information de contexte n'est ajoutée aux arguments, afin de préserver cette absence de contexte.

Il est également possible de forcer un argument de macro à ne pas ajouter d'information de contexte à son texte. Le nom de cet argument doit être précédé du caractère “&”. Cela permet d'avoir un argument dont le texte est réinterprété dans le nouveau contexte lors de l'évaluation d'une macro.

```
char = 'A'
other.char = 'W'

macro both a, &b
      namespace other
            db a, b
      end namespace
end macro

both char+1, char+1 ; db 'B', 'X'
```

# 11. Répétition de blocs d'instructions

L'instruction “repeat” permet d'assembler un bloc d'instructions plusieurs fois, le nombre de répétitions étant spécifié par la valeur de son argument. Le bloc d'instructions doit se terminer par la commande “end repeat”. Le synonyme “rept” peut être utilisé à la place de “repeat”.

```
a = 2
repeat a + 3
      a = a + 1
end repeat
assert a = 7
```

L'instruction “while” permet d'exécuter un bloc d'instructions de manière répétée tant que la condition spécifiée par son argument est vraie. Cet argument doit être une expression logique, comme celle utilisée pour “if” ou “assert”. Le bloc doit être fermé par la commande “end while”.

```
a = 7
while a > 4
      a = a - 2
end while
assert a = 3
```

Le symbole “%” est un paramètre spécial qui est prétraité au sein du bloc d'instructions répété et remplacé par un nombre décimal correspondant au numéro de la répétition (à partir de 1). Son fonctionnement est similaire à celui d'un paramètre de macro-instruction : sa valeur est remplacée avant l'exécution de la commande proprement dite. Il peut ainsi servir à créer des identifiants de symboles contenant ce nombre dans leur nom.

```
repeat 16
      f#% = 1 shl %
end repeat
```

L'exemple ci-dessus définit les symboles “f1” à “f16” dont les valeurs sont les puissances consécutives de deux.

L'instruction “repeat” peut accepter des arguments supplémentaires, séparés par des virgules, chacun contenant le nom d'un paramètre additionnel spécifique à ce bloc. Chaque nom peut être suivi du caractère “:” et de l'expression spécifiant la valeur de base à partir de laquelle le paramètre commencera à compter les répétitions. Cela permet de modifier facilement l'exemple précédent pour définir la plage de symboles de “f0” à “f15”.

```
repeat 16, i:0
      f#i = 1 shl i
end repeat
```

Le paramètre “%%” est un autre paramètre spécial dont la valeur correspond au nombre total de répétitions prévues. Ce paramètre n'est pas défini à l'intérieur du bloc “while”. L'exemple suivant l'utilise pour créer une séquence d'octets dont les valeurs décroissent de 255 à 0 :

```
repeat 256
      db %%-%
```

```
end repeat
```

L'instruction “break” permet d'interrompre la répétition prématurément. Lorsqu'elle est rencontrée, elle provoque l'omission du reste du bloc répété et empêche toute nouvelle répétition. Elle peut être utilisée pour interrompre la répétition si une certaine condition est remplie :

```
s = x/2
repeat 100
      if x/s = s
            break
      end if
      s = (s+x/s)/2
end repeat
```

L'exemple ci-dessus tente de trouver la racine carrée de la valeur du symbole “x”, qui est supposé défini ailleurs. Il peut facilement être réécrit pour effectuer la même tâche avec “while” au lieu de “repeat” :

```
s = x/2
while x/s <> s
      s = (s+x/s)/2
      if % = 100
            break
      end if
end while
```

L'instruction “iterate” (ou “irp”) répète le bloc d'instructions en parcourant la liste de valeurs séparées par des virgules. Le premier argument de “iterate” doit être le nom du paramètre, suivi d'une virgule puis d'une liste de valeurs. À chaque itération, le paramètre reçoit une valeur de la liste.

```
iterate value, 1,2,3
      db value
end iterate
```

Comme pour un argument de macro-instruction, la valeur d'un paramètre contenant des virgules doit être encadrée par les caractères “<” et “>”. Il est également possible d'encadrer le premier argument de “iterate” par “<” et “>” afin de définir plusieurs paramètres. La liste des valeurs est alors divisée en sections contenant autant de valeurs qu'il y a de paramètres, et chaque itération opère sur une section, en attribuant à chaque paramètre une valeur correspondante.

```
iterate <name,value>, a,1, b,2, c,3
      name = value
end iterate
```

Le nom d'un paramètre peut également, comme pour les macro-instructions, être suivi d'un astérisque (*) pour exiger une valeur non vide, ou de deux guillemets (“:”) suivis d'une valeur par défaut. Si une instruction “iterate” se termine par une virgule sans autre caractère, celle-ci n'est pas interprétée comme une valeur vide supplémentaire. Pour insérer une valeur vide à la fin d'une liste, il faut utiliser un couple de chevrons vide (“<>”).

L'instruction “break”, ainsi que les paramètres “%” et “%%”, peuvent être utilisés dans le bloc “iterate” avec les mêmes effets que pour “repeat”.

L'instruction “indx” ne peut être utilisée que dans un bloc “iterate” et modifie les valeurs de tous les paramètres itérés pour leur attribuer celles correspondant à l'itération dont le numéro est spécifié par son argument (mais lors de l'itération suivante, les valeurs des paramètres sont à nouveau affectées normalement). Cela permet de traiter les valeurs itérées dans un ordre différent. Dans l'exemple suivant, les valeurs sont traitées de la dernière à la première :

```
iterate value, 1,2,3
      indx 1+%%-%
      db value
end iterate
```

Avec “indx”, il est même possible de déplacer la vue des valeurs itérées plusieurs fois au cours d'une seule répétition. Dans l'exemple suivant, l'ensemble du traitement est effectué lors de la première répétition du bloc itéré, puis l'instruction “break” est utilisée pour empêcher d'autres itérations :

```
iterate str, 'alpha','beta','gamma'
      repeat %%
            dw offset#%
      end repeat
      repeat %%
            indx %
            offset#% db str
```

```
        end repeat
        break
end iterate
```

Les paramètres définis par “iterate” n'ajoutent pas de contexte aux valeurs itérées, mais ne suppriment pas non plus le contexte d'origine s'il est déjà associé au texte des arguments. Ainsi, si les valeurs passées à “iterate” proviennent d'un autre paramètre qui a conservé le contexte d'origine des identifiants de symboles (comme le paramètre d'une macro-instruction), ce contexte est préservé. Sinon, “iterate” effectue simplement une substitution de texte brut.

Les paramètres définis par des instructions telles que “iterate” ou “repeat” sont traités dans tout le texte du bloc associé, sauf si le bloc est défini en partie par le texte d'une macro-instruction et en partie ailleurs. Dans ce cas, les paramètres ne sont accessibles que dans les parties du bloc définies au même endroit que la commande initiale.

Lorsqu'un paramètre est défini, son nom peut être suivi du caractère “?” pour indiquer qu'il n'est pas sensible à la casse. Cependant, lorsque les paramètres sont reconnus à l'intérieur d'une ligne prétraitée, la présence ou non d'un “?” à la fin est sans importance. Le seul modificateur reconnu par le préprocesseur lorsqu'il remplace le paramètre par sa valeur est le caractère “`”.

Les instructions répétitives, associées à l'instruction “if”, appartiennent au groupe des directives de contrôle. Elles déterminent le flux d'assemblage. Chacune définit un bloc d'instructions subordonnées, fermé par la commande “end” correspondante. Si ces blocs sont imbriqués, l'imbrication doit être correcte : le bloc interne doit toujours être fermé avant le bloc externe. Toutes les directives de contrôle sont donc des instructions inconditionnelles ; elles sont reconnues même à l'intérieur d'un bloc normalement ignoré.

L'instruction “postpone” est une autre directive de contrôle qui permet d'assembler un bloc d'instructions ultérieurement, une fois que tout le texte source suivant a été traité.

```
dw final_count
postpone
        final_count = counter
end postpone
counter = 0
```

L'exemple ci-dessus reporte la définition du symbole “final_count” jusqu'à ce que l'intégralité du code source ait été traitée, afin de pouvoir accéder à la valeur finale de la variable “counter”.

L'assemblage du texte source qui suit “postpone” inclut l'assemblage de tous les blocs supplémentaires déclarés avec “postpone”. Par conséquent, s'il existe plusieurs blocs de ce type, ils sont assemblés dans l'ordre inverse. Le dernier bloc déclaré est assemblé en premier lorsque la fin du texte source est atteinte.

Lorsque la directive “postpone” reçoit un argument constitué d'un seul caractère “?”, elle indique à l'assembleur que le bloc contient des opérations qui ne doivent affecter aucune des valeurs définies dans le code source principal. L'assembleur peut donc s'abstenir de les évaluer tant que toutes les autres valeurs n'ont pas été résolues avec succès. Ces blocs sont traités encore plus tard que ceux déclarés par “postpone” sans argument. Ils peuvent être utilisés pour effectuer des tâches de finalisation, comme le calcul d'une somme de contrôle du code assemblé.

“irpv” est une autre instruction de répétition et un itérateur. Elle ne prend que deux arguments : le nom du paramètre et l’identifiant de la variable. Elle parcourt toutes les valeurs empilées de la variable symbolique, en commençant par la plus ancienne (ceci s’applique uniquement aux valeurs définies précédemment dans le code source).

```
var equ 1
var equ 2
var equ 3
var reequ 4
irpv param, var
        db param
end irpv
```

Dans l'exemple ci-dessus, il y a trois itérations, avec les valeurs 1, 2 et 4.

“irpv” peut effectivement convertir une valeur de variable symbolique en paramètre, et cela peut être utile en soi, car la variable symbolique n'est évaluée que dans les expressions à l'intérieur des arguments des instructions (étiquetées ou non), tandis que les paramètres sont prétraités dans toute la ligne avant le démarrage de tout traitement de commande. Cela permet par exemple de redéfinir une valeur régulière liée par une variable symbolique :

```
x = 1
var equ x
irpv symbol, var
```

```
        indx %%
        symbol = 2
        break
end irpv
assert x = 2
```

La combinaison de “indx” et “break” a été ajoutée à l'exemple ci-dessus afin de limiter l'itération à la dernière valeur de la variable symbolique. La section suivante présentera une solution plus efficace à ce problème.

Lorsqu'une variable passée à “irpv” a une valeur non symbolique, le paramètre reçoit un texte qui produit la même valeur après calcul. Si la valeur est un nombre positif, le paramètre est remplacé par sa représentation décimale (de la même manière que le paramètre “%” est traité) ; sinon, il est remplacé par l'identifiant d'un symbole proxy contenant la valeur stockée dans la pile.

La directive “outscope” est disponible pendant l'exécution de toute macro-instruction et modifie la commande qui suit sur la même ligne. Si cette commande entraîne la définition de paramètres, ceux-ci sont créés non pas dans le contexte de la macro-instruction en cours d'exécution, mais dans celui du texte source qui l'a appelée.

```
macro irpv?! statement&
        display 'IRPV wrapper'
        esc outscope irpv statement
end macro
```

Cela permet non seulement d'encapsuler en toute sécurité certaines directives de contrôle dans des macro-instructions, mais aussi de créer des constructions de langage personnalisées supplémentaires définissant les paramètres d'un bloc de texte. Étant donné que “outscope” doit figurer dans le texte d'une macro-instruction spécifique qui le requiert, il est recommandé de l'utiliser conjointement avec “esc”, comme dans l'exemple ci-dessus. Ceci garantit un traitement identique même lorsque la définition complète est placée à l'intérieur d'une autre macro-instruction.

## 12. Correspondance de paramètres

La directive “match” contrôle l'assemblage de son bloc d'instructions uniquement lorsque le texte spécifié par son deuxième argument correspond au motif donné par le premier. Le texte est séparé du motif par une virgule et inclut tout ce qui suit ce séparateur jusqu'à la fin de la ligne.

Chaque caractère spécial (à l'exception de “, ” et “=”, qui ont une signification spécifique dans le motif) est interprété littéralement : il doit correspondre à un élément identique dans le texte. Dans l'exemple suivant, le contenu du premier bloc est assemblé, mais pas celui du second.

```
match +,+
        assert 1     ; correspondance trouvée
end match

match +,-
        assert 0     ; pas de correspondance
end match
```

Les chaînes entre guillemets sont également comparées littéralement, mais les noms dans le modèle sont traités différemment. Chaque nom fait office de caractère générique et peut correspondre à n'importe quelle séquence de jetons non vide. En cas de correspondance, les paramètres portant ces noms sont créés et chacun se voit attribuer une valeur égale au texte avec lequel le caractère générique a été trouvé.

```
match a[b], 100h[3]
        dw a+b    ; dw 100h+3
end match
```

Dans un modèle, le nom d'un paramètre peut être suivi d'un caractère “?” pour indiquer qu'il n'est pas sensible à la casse.

Le caractère “=” permet une correspondance littérale avec le jeton qui le suit. Il permet ainsi de faire correspondre des jetons de nom, mais aussi des caractères spéciaux qui auraient autrement une signification différente, tels que “,”, “=” ou “?” après un nom.

```
match =a==a, a=8
        db a     ; db 8
end match
```

Si “=” est suivi d'un jeton de nom auquel est attaché le caractère “?”, cet élément est comparé littéralement mais sans tenir compte de la casse :

```
match =a?==a, A=8
        db a     ; db 8
```

```
end match
```

Lorsqu'un motif contient de nombreux caractères génériques, chacun d'eux est associé au minimum de jetons possible, et le dernier récupère le texte restant. Si les caractères génériques se suivent sans aucun élément correspondant littéralement, le premier est associé à un seul jeton, et le second au texte restant :

```
match car cdr, 1+2+3
      db car    ; db 1
      db cdr    ; db +2+3
end match
```

Dans l'exemple ci-dessus, le texte correspondant doit contenir au moins deux jetons, car chaque caractère générique nécessite au moins un jeton pour être non vide. L'exemple suivant introduit des contraintes supplémentaires, mais les mêmes règles générales s'appliquent et le premier caractère générique consomme le moins de ressources possible :

```
match first:rest, 1+2:3+4:5+6
      db `first    ; db '1+2'
      db 13,10
      db `rest     ; db '3+4:5+6'
end match
```

Bien que tout espace à côté d'un caractère générique soit ignoré, la présence ou l'absence d'espace entre des éléments littéralement correspondants est significative. Si de tels éléments n'ont pas d'espace entre eux, leurs homologues ne doivent pas non plus contenir d'espace entre eux. Mais s'il y a un espace entre les éléments du motif, cela n'impose aucune contrainte sur l'utilisation de l'espace dans le texte correspondant – il peut être présent ou non.

```
match ++,++
      assert 1     ; correspondance trouvée
end match

match ++,+ +
      assert 0     ; pas de correspondance
end match

match + +,++
      assert 1     ; correspondance trouvée
end match

match + +,+ +
      assert 1     ; correspondance trouvée
end match
```

La présence d'espaces dans le texte devient requise lorsque le modèle contient le caractère “=” suivi d'un espace :

```
match += +, ++
      assert 0     ; pas de correspondance
end match

match += +, + +
      assert 1     ; correspondance trouvée
end match
```

La commande “match” est analogue à “if” en ce qu'elle permet d'utiliser “else” ou “`else match`” pour créer une sélection de blocs parmi lesquels un seul est exécuté :

```
macro let param
      match dest+==src, param
            dest = dest + src
      else match dest-==src, param
            dest = dest - src
      else match dest++, param
            dest = dest + 1
      else match dest--, param
            dest = dest - 1
      else match dest==src, param
            dest = src
```

```
        else
                assert 0
        end match
end macro

let x=3             ; x = 3
let x+=7            ; x = x + 7
let x++             ; x = x + 1
```

Il est même possible de combiner les conditions “if” et “match” dans une séquence de blocs “else”. L’ensemble de la construction doit se terminer par la commande “end” correspondant à la dernière instruction utilisée.

```
macro record text
        match any, text
                recorded equ ‘text
        else if RECORD_EMPTY
                recorded equ ''
        end if
end macro
```

La fonction “match” est capable de reconnaître les variables symboliques et, avant le début de la correspondance, leurs identifiants dans le texte du deuxième argument sont remplacés par les valeurs correspondantes (de la même manière qu'ils sont remplacés dans le texte qui suit la commande “equ”) :

```
var equ 2+3

match a+b, var
        db a xor b
end match
```

Cela signifie que “match” peut être utilisé à la place de “irpv” pour convertir la dernière valeur d'une variable symbolique en paramètre. L'exemple de la section précédente, où “irpv” était utilisé avec “break” pour effectuer une seule itération sur la dernière valeur, peut être réécrit en utilisant “match” à la place :

```
x = 1
var equ x
match symbol, var
        symbol = 2
end match
assert x = 2
```

La différence entre les deux réside dans le fait que “irpv” exécute son bloc même pour une valeur vide, tandis que dans le cas de “match”, le bloc “else” doit être ajouté pour gérer un texte vide.

Lorsque l'évaluation des variables symboliques dans le texte correspondant est indésirable, un symbole créé avec “define” peut servir de proxy pour préserver le texte, car le remplacement n'est pas récursif.

```
macro drop value
        local temporary
        define temporary value
        match =A, temporary
                db A
                restore A
        else
                db value
        end match
end macro

A equ 1
A equ 2

drop A
drop A
```

On pourrait craindre que “define” modifie le sens d'un texte en lui attribuant un contexte local. Cependant, lorsque la valeur de “define” provient d'un paramètre d'une macro-instruction (comme dans l'exemple ci-dessus), elle conserve son contexte d'origine et “define” ne le modifie pas.

La directive “rawmatch” (synonyme : “rmatch”) est très similaire à “match”, mais elle agit directement sur le texte brut du second argument. Non seulement elle n'évalue pas les variables symboliques, mais elle supprime également tout contexte supplémentaire que le texte aurait pu contenir.

```
struc has instruction
      rawmatch text, instruction
            namespace .
                  text
            end namespace
      end rawmatch
end struc

define x
x has a = 3
assert x.a = 3
```

Dans l'exemple ci-dessus, l'identifiant de “a” serait interprété dans le contexte effectif de la ligne appelant la macro-instruction “has” s'il n'était pas reconverti en texte brut par “rmatch”.

## 13. Zones de sortie

L'instruction “org” crée une nouvelle zone de sortie. Le contenu de cette zone est écrit dans le fichier de destination, à côté des données précédentes, mais les adresses de cette nouvelle zone dépendent de la valeur spécifiée dans l'argument de “org”. La zone est automatiquement fermée lorsque la suivante est créée ou lorsque la source s'arrête.

```
org 100h
start:      ; début à = 100h
```

Le symbole “$” est un symbole intégré de la classe d’expression qui est toujours égal à la valeur de l'adresse courante. Par conséquent, définir une constante avec la valeur spécifiée par le symbole “$” est équivalent à définir une étiquette au même endroit.

```
org 100h
start = $   ; début à = 100h
```

Le symbole “$$” est toujours égal à l'adresse de base de l'espace d'adressage courant. Ainsi, dans la zone commençant par “org”, il a la même valeur que l'adresse de base fournie en argument de “org”. La différence entre “$” et “$$” réside donc dans la position actuelle par rapport au début de la zone.

```
org 2000h
db 'Hello!'
size = $ - $$      ; taille = 6
```

Le symbole “$@” correspond à l'adresse de base du bloc de données non initialisées courant. Si aucune donnée n'était définie juste avant la position courante, cette valeur est égale à “$”. Sinon, elle est égale à “$” moins la longueur de ces données dans l'espace d'adressage courant. Notez que les données réservées ne sont plus considérées comme telles lorsqu'elles sont suivies de données initialisées.

L'instruction “section” est similaire à “org”, mais elle supprime en plus toutes les données réservées qui la précèdent, de la même manière que les données non initialisées ne sont pas écrites en sortie lorsqu'elles se trouvent en fin de fichier. L'instruction “section” peut donc être suivie de définitions de données initialisées sans que les données réservées précédentes soient initialisées à zéro et écrites en sortie. Dans cet exemple, seul le premier des trois tampons réservés est effectivement converti en données à zéro et écrit en sortie, car il est suivi de données initialisées. Le deuxième est supprimé par l'instruction “section”, et le troisième est tronqué car il se trouve en fin de fichier :

```
data1 dw 1
buffer1 rb 10h     ; mis à zéro et présent dans la sortie

org 400h
data dw 2
buffer2 rb 20h     ; pas dans la sortie

section 1000h
data3 dw 3
buffer3 rb 30h     ; pas dans la sortie
```

Le symbole intégré “$%” représente le décalage dans le fichier de sortie où les données initialisées seraient générées si elles étaient définies à cet endroit. Le symbole “$%%” représente le décalage actuel dans le fichier de sortie. Ces deux valeurs diffèrent uniquement lorsqu'elles sont utilisées après la réservation de données : “$%” est

alors supérieur à “$%%” de la longueur des données non initialisées qui seraient générées en sortie si elles étaient suivies de données initialisées.

```
db 'Hello!'
rb 4
position = $%%   ; position = 6
next = $%        ; suivant = 10
```

Les valeurs indiquées dans les commentaires de l'exemple ci-dessus supposent que le code source ne contient aucune autre instruction générant une sortie.

La commande “virtual” crée une zone de sortie spéciale qui n'est pas écrite dans le fichier de sortie principal. Cette zone doit se situer entre les commandes “virtual” et “end virtual”. Une fois fermée, le générateur de sortie reprend son exécution dans la zone précédente, à la même position et à la même adresse qu'avant l'ouverture du bloc “virtual”. Ceci permet également d'imbriquer les blocs “virtual” les uns dans les autres.

Lorsque “virtual” ne reçoit aucun argument, l'adresse de base de cette zone est identique à l'adresse courante dans la zone externe. Un argument de “virtual” peut prendre la forme du mot-clé “at” suivi d'une expression définissant l'adresse de base de la zone délimitée :

```
int dw 1234h
virtual at int
      low db ?
      high db ?
end virtual
```

Au lieu ou en plus d'un tel argument, “virtual” peut également être suivi d'un mot-clé “as” et d'une chaîne définissant une extension de fichier supplémentaire où le contenu initialisé de la zone sera stocké à la fin d'un assemblage réussi.

L'instruction “load” définit la valeur d'une variable en chargeant la chaîne d'octets à partir des données générées dans une zone de sortie. Il doit être suivi d'un identifiant du symbole à définir, puis éventuellement du caractère “:” et d'un nombre d'octets à charger, puis du mot-clé “from” et d'une adresse des données à charger. Cette adresse peut être spécifiée selon deux modes. S'il s'agit simplement d'une expression numérique, il s'agit d'une adresse dans la zone actuelle. Dans ce cas, les octets chargés doivent avoir déjà été générés, il n'est donc possible de charger qu'à partir de l'espace entre les adresses “$$” et “$”.

```
virtual at 100h
      db 'abc'
      load b:byte from 101h   ; b = 'b'
end virtual
```

Lorsque le nombre d'octets n'est pas spécifié, la longueur de la chaîne chargée est déterminée par la taille associée à l'adresse.

L'autre variante de “load” requiert un type d'étiquette particulier, créée avec “::” au lieu de “:”. Cette étiquette possède une valeur qui ne peut être utilisée directement, mais elle peut servir avec l'instruction “load” pour accéder aux données de la zone où elle est définie. L'adresse de “load” doit alors être spécifiée comme suit : l'étiquette de la zone, suivie de “:”, puis de l'adresse au sein de cette zone.

```
virtual at 0
      hex_digits::
      db '0123456789ABCDEF'
end virtual
load a:byte from hex_digits:10  ; a = 'A'
```

Cette variante de “load” permet d’accéder aux données générées ultérieurement, même dans la zone actuelle :

```
area::
db 'abc'
load sub:3 from area:$-2     ; sub = 'bcd'
db 'def'
```

L'instruction “store” permet de modifier des données déjà générées dans la zone de sortie. Elle doit être suivie d'une valeur (automatiquement convertie en chaîne d'octets), puis éventuellement du caractère “:” suivi du nombre d'octets à écrire (si ce paramètre est absent, la longueur de la chaîne est déterminée par la taille associée à l'adresse), puis du mot-clé “at” et de l'adresse des données à remplacer, selon l'un des deux modes autorisés par “load”. Cependant, “store” ne peut pas modifier les données non encore générées, et toute zone modifiée par “store” devient une zone variable, interdisant également à “load” d'y lire des données à l'avance.

L'exemple suivant utilise la combinaison de “load” et “store” pour chiffrer l'intégralité du contenu de la zone courante par une simple opération “XOR” :

```
db "Text"
```

```
key = 7Bh
repeat $-$$
      load a : byte from $$+%-1
      store a xor key : byte at $$+%-1
end repeat
```

Si les données finales d'une zone modifiée par “store” doivent être lues plus tôt dans la source, cela peut se faire en copiant ces données dans une autre zone non soumise à cette contrainte. Ceci est analogue à la définition d'une constante avec une valeur finale pour une variable :

```
load char : byte from const:0

virtual
      var::
      db 'abc'
      .length = $
end virtual

store 'A' : byte at var:0

virtual
      const::
      repeat var.length
            load a : byte from var:%-1
            db a
      end repeat
end virtual
```

L'étiquette de zone peut être référencée par l'instruction “load”, mais jamais par l'instruction “store”, même si elle fait référence à la zone de sortie courante.

L'instruction “virtual” peut prendre comme unique argument une étiquette de zone existante. Cette variante permet d'étendre un bloc précédemment défini et fermé avec des données supplémentaires. L'étiquette de zone doit faire référence à un bloc créé antérieurement dans le code source avec l'instruction “virtual”. Toute définition de données dans un bloc étendu aura le même effet que si cette définition était présente dans le bloc “virtual” d'origine.

```
virtual at 0 as 'log'
      Log::
end virtual

virtual Log
      db 'Hello!',13,10
end virtual
```

Si une étiquette de zone est utilisée dans une expression, elle constitue un terme variable d'un polynôme linéaire. Les métadonnées de ce terme correspondent à l'adresse de base de la zone. Les métadonnées de l'étiquette de zone elle-même, accessibles via l'opérateur “sizeof”, sont égales à la longueur actuelle des données dans la zone.

Il existe une variante des directives “load” et “store” permettant de lire et de modifier des données déjà générées dans le fichier de sortie, à partir d'un simple décalage dans ce fichier. Cette variante est reconnue lorsque le mot-clé “at” ou “from” est suivi du caractère “:” puis de la valeur du décalage.

```
checksum = 0
repeat $%
      load a : byte from : %-1
      checksum = checksum + a
end repeat
```

L'instruction “restartout” annule toute la sortie générée jusqu'à présent et recommence à zéro avec une zone vide. Un argument optionnel peut spécifier l'adresse de base de la nouvelle zone de sortie. Si “restartout” ne reçoit aucun argument, l'adresse courante est conservée et utilisée comme base pour la nouvelle zone.

Les instructions “org”, “section” et “restartout” ne peuvent pas être utilisées à l'intérieur d'un bloc “virtual” ; elles servent uniquement à séparer les zones qui seront écrites dans le fichier de sortie.

# 14. Contrôle de la source et de la sortie

L'instruction “include” lit le texte source d'un autre fichier et le traite avant de poursuivre l'exécution dans le code source courant. Son argument doit être une chaîne de caractères définissant le chemin d'accès au fichier (le format du chemin peut varier selon le système d'exploitation). Si un “!” est présent entre l'instruction et l'argument, l'autre fichier est lu et traité systématiquement, même s'il se trouve dans un bloc ignoré (les instructions inconditionnelles de l'autre fichier peuvent alors être reconnues).

```
include 'macro.inc'
```

Un argument supplémentaire peut être ajouté (séparé du chemin par une virgule). Il est interprété comme une commande à exécuter après la lecture et l'insertion du fichier dans le flux source, juste avant le traitement de la première ligne.

L'assembleur recherche le fichier dans le répertoire contenant le code source en cours de traitement. Si le fichier est introuvable, il recherche dans le répertoire depuis lequel le processus d'assemblage a été lancé, ainsi que dans les chemins consécutifs définis sous forme de liste séparée par des points-virgules dans la variable d'environnement “INCLUDE”.

L'instruction “eval” prend une séquence d'octets définie par ses arguments, la traite comme un texte source et l'assemble. Les arguments sont soit des chaînes de caractères, soit des valeurs numériques d'octets individuels, séparés par des virgules. Dans l'exemple suivant, “eval” est utilisé pour générer des définitions de symboles nommés d'après les lettres consécutives de l'alphabet :

```
repeat 26
      eval 'A'+%-1,'=',`%
end repeat

assert B = 2
```

L'instruction “display” provoque l'écriture d'une séquence d'octets dans la sortie standard, à côté des messages générés par l'assembleur. Il doit être suivi de chaînes ou de valeurs numériques d'octets simples, séparées par des virgules. L'exemple suivant utilise “repeat 1” pour définir un paramètre avec une représentation décimale du nombre calculé, puis l'affiche sous forme de chaîne :

```
macro show description,value
      repeat 1, d:value
            display description,`d,13,10
      end repeat
end macro

show '2^64=',1 shl 64
```

L'instruction “err” signale une erreur lors du processus d'assemblage, avec un message personnalisé spécifié par son argument. Elle accepte les mêmes types d'arguments que la directive “display”.

```
if $>10000h
      err 'segment too large'
end if
```

La directive “format” permet de spécifier des options supplémentaires concernant le format de sortie principal. Elle peut prendre la forme “format binary” ou “format executable”, un choix sans incidence sur le fichier produit, sauf que “format executable” indique que le fichier peut être marqué comme exécutable sur les systèmes où cela est nécessaire. L'instruction “format” peut également être suivie du mot-clé “as” et d'une chaîne de caractères définissant l'extension du fichier de sortie. À moins qu'un nom ne soit spécifié sur la ligne de commande, le fichier de sortie est construit à partir du chemin d'accès au fichier source principal, en supprimant l'extension d'origine et en ajoutant une nouvelle extension si celle-ci est définie.

```
format binary as 'com'
```

La directive “format”, à l'instar de “end”, utilise un identifiant qui la suit pour trouver une instruction dans l'espace de noms enfant du symbole insensible à la casse nommé “format”. Les seules instructions intégrées présentes dans cet espace de noms sont “binary” et “executable”, mais d'autres peuvent être définies sous forme de macro-instructions.

Le symbole intégré “__time__” (avec son synonyme hérité “%t”) a pour valeur constante l'horodatage marquant le moment où l'assemblage a commencé.

Le symbole intégré “__file__” a pour valeur une chaîne de caractères contenant le nom du fichier source en cours de traitement. Le symbole “__line__” associé indique le numéro de la ligne en cours de traitement dans ce fichier. Lorsque ces symboles sont utilisés dans une macro-instruction, ils conservent la même valeur que celle qu'ils avaient pour la ligne appelante. En cas de macro-instructions imbriquées, ces symboles ont la même valeur partout, correspondant à la ligne qui a appelé la macro-instruction la plus externe.

Le symbole “__source__” est un autre symbole intégré, dont la valeur est une chaîne de caractères contenant le nom du fichier source principal.

La directive “retaincomments” configure l'assembleur pour qu'il traite le point-virgule comme un caractère ordinaire et ne supprime donc pas les commentaires des lignes avant traitement. Cela permet d'utiliser des points-virgules dans des expressions régulières comme MATCH.

```
retaincomments
macro ? line&
      match instruction ; comment , line
            virtual
                  comment
            end virtual
            instruction
      else
            line
      end match
end macro

var dd ?  ; bvar db ?
```

La directive “isolatelines” empêche l'assembleur de combiner les lignes lues dans le texte source lorsque le saut de ligne est précédé d'une barre oblique inverse.

La directive “removecomments” rétablit le comportement par défaut des points-virgules, tandis que la directive “combinelines” autorise la combinaison des lignes du texte source.

# 15. Instructions CALM

La directive “calminstruction” permet de définir de nouvelles instructions sous forme de séquences compilées de commandes spécialisées. Contrairement aux macro-instructions classiques, qui fonctionnent selon un principe simple de substitution textuelle, les instructions CALM (*Compiled Assembly-Like Macro*) peuvent effectuer de nombreuses opérations sans que le texte ne subisse le prétraitement et l'assemblage standard. Ceci permet un contrôle plus précis, une meilleure gestion des erreurs et une exécution plus rapide.

Toutes les références aux symboles dans le texte définissant une instruction CALM sont fixées lors de sa définition. Par conséquent, les symboles locaux à l'instruction CALM sont partagés entre toutes ses instances exécutées (par exemple, les instances successives peuvent accéder aux valeurs des symboles locaux laissées par les précédentes). Pour faciliter la réutilisation de ces références, les commandes CALM opèrent généralement sur des variables, en réécrivant régulièrement les symboles avec de nouvelles valeurs.

Une instruction “calminstruction” suit les mêmes règles qu'une déclaration “macro”, y compris les options comme “!”. Le modificateur permet de définir une instruction inconditionnelle, l'astérisque (*) indique un argument obligatoire, la virgule (“:”) lui attribue une valeur par défaut et le symbole “&” indique que le dernier argument occupe tout le texte restant sur la ligne.

Cependant, comme l'instruction CALM fonctionne en dehors du cycle standard de prétraitement et d'assemblage, ses arguments ne deviennent pas des paramètres prétraités. Ce sont des variables symboliques locales, auxquelles sont attribuées de nouvelles valeurs à chaque appel de l'instruction.

Une instruction “calminstruction” peut également définir une instruction étiquetée ; il n'existe pas de commande spécifique à cet effet, comme “struc” pour les macro-instructions classiques. Si le nom de l'instruction définie est précédé d'un autre nom entre parenthèses, l'instruction définit une instruction étiquetée, et le nom entre parenthèses constitue l'argument qui recevra le texte de l'étiquette.

Dans la définition d'une instruction CALM, seules les instructions de son langage spécialisé sont identifiées. Le symbole initial de chaque ligne doit être un nom simple, sans modificateur. Il n'est reconnu comme instruction valide que si un symbole insensible à la casse portant ce nom est trouvé dans l'espace de noms des commandes CALM (accessible, à des fins de personnalisation, comme l'espace de noms ancré au symbole insensible à la casse “calminstruction”). Si aucune instruction nommée de ce type n'est trouvée, le nom initial peut devenir une étiquette s'il est suivi de “:”. Il est alors traité comme un symbole sensible à la casse appartenant à une classe spécialisée. Les symboles de cette classe ne sont reconnus que lorsqu'ils sont utilisés comme arguments de commandes de saut CALM (décrites plus loin).

L'instruction “end calminstruction” est nécessaire pour fermer la définition et rétablir le mode assembleur normal. Il ne s'agit pas d'une commande “end” classique, mais d'une instruction du même nom dans l'espace de noms CALM, qui n'accepte que “calminstruction” comme argument.

La commande “assemble” prend un seul argument, qui doit être l'identifiant d'une variable symbolique. Le texte de cette variable est transmis directement à l'assembleur, sans prétraitement (si le texte provenait d'un argument de l'instruction, il avait déjà été prétraité lors de la préparation de cette ligne).

```
calminstruction please? cmd&
      assemble cmd
end calminstruction

please display 'Hi!'
```

La commande “match” est à bien des égards similaire à la directive standard du même nom. Son premier argument doit être un modèle suivant les mêmes règles que celles de la directive “match”. Le deuxième argument doit être un identifiant d'une variable symbolique, dont le texte va être comparé au modèle. Les jetons de nom dans le modèle (à l'exception de ceux rendus littéraux avec le symbole “=”) sont traités comme des noms de variables où les parties de texte correspondantes doivent être placées si la correspondance est réussie. La même variable qui est une source de texte peut également être utilisée dans le modèle comme variable dans laquelle écrire. Lorsqu'il n'y a pas de correspondance, toutes les variables ne sont pas affectées.

```
calminstruction please? cmd&
      match (cmd), cmd
      assemble cmd
end calminstruction

please(display 'Hi!')
```

La réussite de la correspondance peut également être vérifiée par un saut conditionnel “jyes” ou “jno” après la commande “match”. Un saut “jyes” est effectué uniquement en cas de réussite de la correspondance.

```
calminstruction please? cmd&
      match =do? =not? cmd, cmd
      jyes done
      assemble cmd
    done:
end calminstruction

please do not display 'Bye!'
```

Pour un contrôle plus précis du flux de traitement, la commande “jump” permet un saut inconditionnel, et la commande “exit” permet d'interrompre le traitement d'une instruction CALM à tout moment (cette commande ne prend aucun argument).

Bien que les symboles utilisés comme arguments de l'instruction soient implicitement locaux, d'autres identifiants peuvent devenir des références fixes à des symboles globaux s'ils sont considérés comme accessibles lors de leur définition (car dans une instruction CALM, toutes ces références sont traitées comme des utilisations, et non comme des définitions). Une commande telle que “match” peut alors écrire dans une variable globale.

```
define comment

calminstruction please? cmd&
      match cmd //comment, cmd
      assemble cmd
end calminstruction

please display 'Hi!' // 3
db comment               ; db 3
```

Pour qu'un symbole soit traité comme local, il convient d'utiliser la commande “local”, suivie d'un ou plusieurs noms séparés par des virgules.

```
calminstruction please? cmd&
      local comment
      match cmd //comment, cmd
      assemble cmd
end calminstruction
```

Pour garantir que le symbole soit considéré comme local, une valeur définie mais inutilisable lui est initialement attribuée. Ceci se produit lorsque l'instruction “local” est rencontrée lors de la définition de l'instruction et qu'aucune commande n'est insérée dans le code de la macro compilée.

Si un motif dans une instruction CALM comporte un caractère “?” immédiatement après le nom d'un caractère générique, cela n'affecte pas l'identification du symbole (la sensibilité à la casse du symbole utilisé dépend de ce qui est présent dans la portée locale au moment de la définition de l'instruction). En revanche, modifier le nom d'un caractère générique avec “?” permet de le faire correspondre à un texte vide.

La commande “arrange” est comme l'inverse de “match”, elle permet de construire un texte contenant les valeurs d'une ou plusieurs variables symboliques. Le premier argument définit une variable où le texte construit va être stocké, tandis que le deuxième argument est un motif formé de la même manière que pour “match” (sauf qu'il n'a pas besoin de précéder une virgule de “=” pour l'inclure dans l'argument). Tous les jetons autres que “=” et les jetons précédés de “=” sont copiés littéralement dans le texte construit et ne comportent aucun contexte de reconnaissance. Les jetons de nom qui ne sont pas rendus littéraux avec “=” sont traités comme des noms de variables dont les valeurs symboliques sont mises à leur place dans le texte construit.

```
calminstruction addr? arg
	local base, index
	match base[index], arg
	local cmd
	arrange cmd, =dd base + index
	assemble cmd
end calminstruction

addr 8[5]		; dd 8 + 5
```

Avec des modèles convenablement sélectionnés, “arrange” peut être utilisé pour copier une valeur symbolique d'une variable à une autre ou pour lui attribuer une valeur fixe (même vide).

Si une variable utilisée dans le modèle s'avère avoir une valeur numérique au lieu de symbolique, tant qu'il s'agit d'un nombre non négatif sans termes supplémentaires, elle est convertie en un jeton décimal stocké dans la valeur symbolique construite (une opération qui, en dehors des instructions CALM, nécessiterait l'utilisation d'une astuce “repeat 1”) :

```
digit = 4 - 1

calminstruction demo
	local cmd
	arrange cmd, =display digit#0h
	assemble cmd
end calminstruction

demo		; display 3#0h
```

Il s'agit du seul cas où une valeur non symbolique est convertie en symboles pouvant être intégrés à un texte composé par “arrange”. Les autres types ne sont pas pris en charge.

La commande “compute” permet d'évaluer des expressions et d'affecter des résultats numériques à des variables. Le premier argument de “compute” définit la destination du résultat, tandis que le second peut être une expression numérique quelconque, précompilée lors de sa définition. Si, lors de l'évaluation de l'expression, l'un des symboles auxquels elle fait référence possède une valeur symbolique, ce texte est analysé comme une nouvelle sous-expression, et sa valeur calculée est ensuite utilisée dans le calcul de l'expression principale.

“compute” peut donc servir non seulement à évaluer une expression prédéfinie, mais aussi à analyser et calculer une expression à partir du texte d'une variable symbolique (comme un argument de l'instruction), ou une combinaison des deux.

```
a = 0

calminstruction low expr*
	compute a, expr and 0FFh
end calminstruction

low 200 + 73	; a = 11h
```

Puisqu'une variable symbolique est évaluée comme une sous-expression, son utilisation ici n'entraîne aucun effet de bord, contrairement à une simple substitution de texte.

La commande “check” est analogue à “if”. Elle évalue une condition définie par l'expression logique qui la suit et configure en conséquence le flag de résultat, lequel peut être testé avec les commandes “jyes” ou “jno”. Les valeurs des variables symboliques sont traitées comme des sous-expressions numériques (elles ne peuvent contenir aucun opérateur spécifique aux expressions logiques).

```
calminstruction u8range? value
	check value >= 0 & value < 256
	jyes ok
	local cmd
	arrange cmd, =err 'value out of range'
```

```
      assemble cmd
    ok:
end calminstruction

u8range -1
```

Toutes les commandes qui ne précisent pas explicitement que le flag vérifié par "jyes" et "jno" est défini conservent la valeur de ce flag inchangée.

La commande "publish" permet d'assigner une valeur à un symbole identifié par le texte stocké dans une variable. Cela permet de définir un symbole avec un nom construit à l'aide d'une commande telle que "arrange", ou un nom passé en argument à une instruction. Le premier argument doit être la variable symbolique contenant l'identifiant du symbole à définir, le second argument doit être la variable contenant la valeur à assigner (symbolique ou numérique). Le premier argument peut être suivi du caractère ":" pour indiquer que le symbole doit être rendu constant, ou précédé de ":" pour que la valeur soit incrémentée par rapport à la précédente (afin que cette dernière puisse être restaurée avec la directive "restore").

```
calminstruction constdefine? var
      local val
      arrange val,
      match var=  val, var
      publish var:, val
end calminstruction

constdefine plus? +
```

L'instruction ci-dessus permet de définir une constante symbolique, ce qui est impossible avec les directives standard de l'assembleur.

La commande "transform" a pour but de remplacer les identifiants de variables symboliques (ou constantes) par leurs valeurs dans un texte donné, opération identique à celle effectuée par la directive "equ" lors de la préparation de la valeur à affecter. L'argument de "transform" doit être une variable symbolique dont la valeur sera traitée puis remplacée par le texte transformé.

```
calminstruction (var) constequ? val
      transform val
      publish var:, val
end calminstruction
```

La commande "transform" met à jour le flag de résultat pour indiquer si un remplacement a été effectué.

```
calminstruction prepasm? cmd&
    loop:
      transform cmd
      jyes loop        ; Attention ! risque de blocage sur des références cycliques
      assemble cmd
end calminstruction
```

Le flag de résultat n'est modifié que par certaines commandes, comme "check", "match" ou "transform". Les autres commandes le laissent inchangé.

La commande "transform" peut prendre deux arguments, le second spécifiant un espace de noms. Les identifiants du texte fourni par le premier argument sont alors interprétés comme des symboles de cet espace de noms, quel que soit leur contexte d'origine.

La commande "stringify" convertit le texte d'une variable en une chaîne de caractères et l'écrit dans cette même variable, spécifiée par son unique argument. Cette opération est similaire à celle effectuée par l'opérateur "`" en prétraitement, mais elle produit une valeur de type chaîne de caractères, et non une chaîne entre guillemets.

```
calminstruction (var) strcalc? val
      compute val, val    ; calcul de l'expression
      arrange val, val    ; conversion du résultat en un jeton décimal
      stringify val       ; conversion d' un jeton décimal en chaîne de caractères
      publish var, val
end calminstruction

p strcalc 1 shl 1000
display p
```

Alors que la plupart des commandes disponibles pour les instructions CALM remplacent les valeurs des variables lors de leur écriture, la commande "take" permet de manipuler des piles de valeurs. Elle supprime la valeur supérieure du symbole source (spécifié par le deuxième argument) et l'affecte au symbole de destination

(le premier argument), en la plaçant au-dessus des valeurs existantes. L'argument de destination peut être vide ; dans ce cas, la valeur est entièrement supprimée et l'opération est analogue à la directive “restore”. Cette commande met à jour le flag de résultat pour indiquer s'il y avait une valeur à supprimer. Si le symbole de destination est identique au symbole source, le flag de résultat permet de vérifier la présence d'une valeur sans la modifier.

```
calminstruction reverse? cmd&
      local tmp, stack
    collect:
      match tmp=,cmd, cmd
      take stack, tmp
      jyes collect
    execute:
      assemble cmd
      take cmd, stack
      jyes execute
end calminstruction

reverse display '!', display 'i', display 'H'
```

Un symbole accédé comme destination ou source par une commande “take” ne peut jamais être référencé ultérieurement, même si cela serait possible autrement.

La définition de macro-instructions dans l'espace de noms “calminstruction” (insensible à la casse) permet d'ajouter des commandes personnalisées au langage des instructions CALM. Cependant, elles doivent être définies comme insensibles à la casse pour être reconnues comme telles.

```
macro calminstruction?.asmarranged? variable*, pattern&
      arrange variable, pattern
      assemble variable
end macro

calminstruction writeln? text&
      asmarranged text, =display text,10
end calminstruction

writeln 'Next!'
```

De telles commandes supplémentaires peuvent même être définies comme des instructions CALM elles-mêmes :

```
calminstruction calminstruction?.initsym? variable*,value&
      publish variable, value
end calminstruction

calminstruction show? text&
      local command
      initsym command, display text
      stringify text
      assemble command
end calminstruction

show :)
```

Dans cet exemple, la commande “initsym” sert à assigner du texte à la variable symbolique locale lors de la définition de l'instruction “show”. À l'instar de “local” (et contrairement à “stringify” et “assemble”), elle ne produit aucun code exécutable lors de l'appel de l'instruction “show”. Les arguments de “initsym” conservent leur contexte d'origine ; par conséquent, les symboles du texte assigné à la variable “command” sont interprétés comme appartenant à l'espace de noms local de l'instruction “show”. Ceci permet à la commande “display” d'accéder au texte, même s'il est local à l'instruction CALM et donc normalement visible uniquement dans la portée de la définition de “show”. Ce fonctionnement est similaire à l'utilisation de “define” pour créer des liens symboliques.

La commande “call” permet d'exécuter directement une autre instruction CALM. Son premier argument doit fournir l'identifiant d'un symbole de classe d'instruction, et lors de l'exécution, ce symbole doit être défini comme CALM (il est impossible d'appeler une macro-instruction ou une instruction intégrée de cette manière). L'exécution se poursuit alors directement jusqu'au point d'entrée de cette instruction, et ne se termine qu'une fois l'instruction appelée achevée.

```
define Msg display 'Hi'
```

```
calminstruction showMsg
      assemble Msg
end calminstruction

calminstruction demo
      call showMsg
      arrange Msg, =display '!'
      call showMsg
end calminstruction

demo
```

Lors de la recherche du symbole d'instruction, l'assembleur ignore l'espace de noms local de l'instruction CALM, car celui-ci ne devrait pas contenir de définitions d'instructions.

Les arguments supplémentaires de l'instruction "call" doivent être des identifiants de variables (ou de constantes) dont les valeurs seront transmises comme arguments à l'instruction appelée. Les valeurs de ces symboles sont directement affectées aux variables d'argument, sans validation supplémentaire. Ceci permet de transmettre à l'instruction CALM des valeurs qui seraient autrement impossibles à transmettre directement, comme des valeurs numériques (car lors d'un appel normal d'instructions, les arguments sont traités comme du texte et affectés comme des valeurs symboliques). Un argument peut être omis si la définition de l'instruction appelée le permet ; dans ce cas, la valeur par défaut de cet argument est utilisée.

```
calminstruction hex_nibble digit*, command: display
      compute digit, 0FFh and '0123456789ABCDEF' shr (digit*8)
      arrange command, command digit
      assemble command
end calminstruction

calminstruction display_hex_byte value: DATA
      compute value, value
      local   digit
      compute digit, (value shr 4) and 0Fh
      call    hex_nibble, digit
      compute digit, value and 0Fh
      call    hex_nibble, digit
end calminstruction

DATA = 0xedfe

calminstruction demo
      call    display_hex_byte
      compute DATA, DATA shr 8
      call    display_hex_byte
end calminstruction

demo
```

Étant donné que certaines commandes, comme "match", n'acceptent que des valeurs symboliques (contenant du texte tokenisé), il existe une variante de "take" appelée "taketext" qui n'opère que lorsque la valeur de la variable source est de ce type. L'instruction appelée peut utiliser "taketext" comme un simple test pour s'assurer que la valeur d'un argument peut être inspectée avec "match".

```
calminstruction forced_db? value*
      taketext value, value
      jno     proceed
      match   ?, value
      jno     proceed
      arrange value, =rb 1
      assemble value
      exit
    proceed:
      compute value, value and 0FFh
      arrange value, =db value
      assemble value
end calminstruction
```

```
calminstruction xoredbyte a, b
	compute a, a xor b
	call    forced_db, a
end calminstruction
```

# 16. Commandes d'assemblage dans les instructions CALM

Un ensemble de commandes supplémentaires pour les instructions CALM permet de les utiliser non seulement pour le prétraitement, mais aussi pour générer et traiter directement la sortie. Elles effectuent des opérations élémentaires, principalement sur une seule unité de données, mais peuvent également réaliser de nombreux calculs directement sur le système, car leurs arguments, à quelques exceptions près, sont des expressions précompilées, similaires au second argument de la commande "compute".

La commande "display" affiche une chaîne d'octets sous forme de message sur la sortie standard, à l'instar de la directive du même nom. Elle prend un seul argument : une expression fournissant soit une chaîne de caractères, soit la valeur numérique d'un octet.

```
calminstruction println? text
	display text bappend 13 bappend 10
end calminstruction
```

La commande "err" signale une erreur, de manière analogue à son homonyme dans le langage de base. Il faut un seul argument, spécifiant un message personnalisé à présenter. L'argument est censé être évalué en valeur de chaîne.

```
calminstruction check8bit? value*
	check   value and 0FFh = value
	jyes    ok
	stringify value
	err     value bappend ' is out of range'
    ok:
end calminstruction
```

La commande "emit" génère des données dont la longueur est spécifiée par le premier argument et la valeur par le second. Les deux arguments sont traités comme des expressions précompilées. Le second argument est facultatif ; s'il est omis, les données de longueur spécifiée sont générées sans initialisation. Si le second argument est une chaîne de caractères, celle-ci doit respecter la taille spécifiée (l'opérateur "lengthof" peut s'avérer utile dans ce cas).

```
calminstruction bigendian64? value*
	emit    8, value bswap 8
end calminstruction
```

Les commandes "load" et "store" permettent d'inspecter ou de modifier des valeurs dans la sortie déjà générée ou dans les blocs virtuels. Bien qu'elles soient similaires à leurs homologues du langage de base, leur syntaxe diffère, les deux prenant toujours trois arguments séparés par des virgules. Contrairement à ces dernières, elles n'opèrent pas sur des adresses associées à des zones de sortie, mais sur des décalages bruts. Pour pointer vers le premier octet d'une zone, le décalage indiqué doit être nul.

Les arguments de "load" sont, dans l'ordre : la variable cible, le décalage à partir duquel charger, et le nombre d'octets à charger. Le premier argument doit être l'identifiant d'un symbole ; les deux suivants sont des expressions précompilées. Le deuxième argument peut contenir l'étiquette de la zone, suivie de ":" puis de l'indice d'un octet, ou simplement une expression numérique, auquel cas il s'agit d'un décalage dans la sortie générée jusqu'à ce point. L'indice/décalage doit être une expression qui s'évalue à une valeur numérique absolue. La valeur chargée est toujours une chaîne de caractères de la longueur spécifiée.

```
virtual
	__upper::
	repeat 256, c:0
		if c >= 'a' & c <= 'z'
			db c-'a'+'A'
		else
			db c
		end if
	end repeat
end virtual

calminstruction (out) uppercase char*
```

```
      load    char, __upper:+char, 1
      publish out, char
end calminstruction
```

Les arguments de la fonction “store” sont, dans l’ordre : le décalage vers lequel stocker les données, le nombre d’octets à stocker et la valeur à stocker (numérique ou chaîne de caractères). Les deux derniers arguments sont analogues à ceux de la fonction “emit”, tandis que les deux premiers sont similaires aux deux derniers arguments de la fonction “load”. Le décalage peut être précédé du nom de la zone, le séparateur étant “:”.

```
calminstruction encrypt offset, key
      local   byte
      load    byte, offset, 1
      store   offset, 1, byte xor key
end calminstruction
```

Pour convertir les adresses utilisées par les commandes classiques “load” et “store” en décalages bruts attendus par les commandes CALM, il suffit d'ajouter ou de soustraire l'adresse de base de la zone. Si cette adresse est inconnue, elle peut être obtenue à l'aide de l'opérateur “1 metadataof” appliqué à l'étiquette de la zone.

## 17. Correspondance avancée dans les instructions CALM

La commande CALM “match” offre davantage de possibilités que son équivalent standard. Comme indiqué précédemment, l'ajout du caractère “?” à un caractère générique le rend facultatif, lui permettant ainsi de correspondre à un texte vide.

```
calminstruction show_tokens text&
    loop:
      match   token text?, text
      jno     done
      stringify token
      display token bappend 0A0Dh
      jump    loop
    done:
end calminstruction
```

Le deuxième argument de CALM “match” étant un identifiant et non un texte libre, il peut être suivi d'autres arguments. Un troisième argument optionnel peut contenir des crochets ; dans ce cas, tous les caractères génériques doivent correspondre à un texte dont les crochets sont correctement espacés.

```
calminstruction range definition
      match   from-to, definition, ()
      emit    1, from
      emit    1, to
    done:
end calminstruction

range (10-3)-10
```

Les crochets sélectionnés par le troisième argument ne doivent apparaître nulle part dans le motif.

Si le troisième argument ne contient qu'un seul caractère spécial, il choisit le symbole pouvant modifier les caractères génériques. Il est préférable de choisir un caractère qui n'apparaît pas dans le motif, mais il est toujours possible de le faire correspondre en le faisant précéder de “=”, comme d'habitude. Si le caractère choisi suit un nom de caractère générique, il doit être suivi d'un modificateur de caractère générique (qui sont des symboles intégrés de la classe d'expression dans l'espace de noms de la fonction “calminstruction”, insensible à la casse).

Le modificateur “name” permet au caractère générique de correspondre à un identifiant de symbole complet.

```
calminstruction constdefine? var
      local   val
      arrange val,
      match   var:name val, var, :
      publish var:, val
end calminstruction

constdefine plus?+
```

Le modificateur “expression” permet au caractère générique de correspondre à une expression calculable complète. Il garantit que l'expression est correctement structurée, mais n'en effectue pas le calcul. Une vérification supplémentaire “defined” peut déterminer si elle ne contient pas de valeurs indéfinies, mais même cela ne garan-

tit pas une évaluation sans erreur : les problèmes tels que les dépassements de capacité n'apparaissent que lors du calcul proprement dit.

```
calminstruction bigendian64? text
      match   text/expression, text, /
      jyes    inspect
      err     'malformed expression'
      exit
    inspect:
      check   defined text
      jyes    evaluate
      err     'expression with unknown quantities'
    evaluate:
      emit    8, text bswap 8
end calminstruction
```

Un caractère générique modifié par “number” correspond à un littéral numérique, entier ou à virgule flottante. Celui modifié avec “quoted” correspond à une chaîne entre guillemets.

```
calminstruction inspect text
      match   text-quoted, text, -
      jyes    quoted
      match   text-expression, text, -
      jno     not
      check   text eqtype ''
      jyes    computed
    not:
      display 'not a string' bappend 0A0Dh
      exit
    quoted:
      display 'string token' bappend 0A0Dh
      exit
    computed:
      display 'computed string' bappend 0A0Dh
end calminstruction

inspect 'a'
inspect 10
inspect string 10
```

Les modificateurs “equ”, “macro” et “struc” limitent l’utilisation du caractère générique à un identifiant de symbole défini : une variable symbolique, une instruction ou une instruction étiquetée, respectivement. Les modificateurs “priorequ”, “priormacro” et “priorstruc” fonctionnent de la même manière, mais garantissent en outre que le symbole a été défini précédemment et qu’il n’est pas référencé ultérieurement.

```
if_true equ if 1
if_false equ if 0

calminstruction ifdef? sym*
      match   sym.name, sym, .
      jno     error
      match   sym.equ, sym, .
      jyes    true
      check   defined sym
      jno     false
    true:
      assemble if_true
      exit
    error:
      err     'not a name'
    false:
      assemble if_false
end calminstruction
```

Un caractère générique modifié avec “area” requiert un identifiant de zone.

Les caractères génériques spécialisés ne peuvent être modifiés d'aucune autre manière : ils ne peuvent pas être rendus optionnels et il est impossible de combiner l'équilibrage des parenthèses et la modification des caractères génériques dans la même commande.

Comme la présence d'espaces est significative pour certains comparateurs spécialisés (comme ceux des identifiants de symboles), les séquences de jetons associées à ces caractères génériques peuvent contenir des espaces en fin de chaîne. Lorsque cela est indésirable, par exemple lorsqu'il est nécessaire d'ajouter un suffixe à l'identifiant correspondant, une correspondance supplémentaire avec un caractère générique simple peut être utilisée pour supprimer les espaces.

```
calminstruction ? &expr&
      local sym, value
      match sym|name == expr, expr, |
      jyes assign
      assemble expr
      exit
    assign:
      compute value, expr
      publish sym:, value
      match sym, sym    ; strip spaces
      arrange sym, sym.=expression
      stringify expr
      publish sym:, expr
end calminstruction
```