fasm

flat assembler g
Manuel de l'Utilisateur

par Tomasz Grysztar

Dernière mise à jour du document original : 23 Février 2024

Traduction française

By Calendos
Forum https://flatassembler.net/
Prélevé le 11 mai 2024
Trad. v 2.2 (22/05/2024 à 10:54)

Ce document décrit la syntaxe du langage *flat assembler g*, avec des exemples de base. Il a été écrit avec l'hypothèse qu'il serait lu séquentiellement et à tout moment il n'utilise que les concepts et les constructions qui ont été introduits précédemment. Cependant, il est possible de passer directement à la section qui intéresse le lecteur, puis de ne revenir aux parties précédentes que lorsque cela est nécessaire afin de mieux comprendre les dernières.

Table des matières

0. Éxécution de l'assembleur

Pour démarrer l'assemblage à partir de la ligne de commande, il est nécessaire de fournir au moins un paramètre : le nom d'un fichier source et, éventuellement, celui du fichier destination. Si l'assemblage réussit, la sortie générée est écrite dans la destination et un bref résumé des tâches effectuées est affiché, sinon une information sur les erreurs est proposée. Le nombre maximum d'erreurs présentées peut être contrôlé avec un commutateur supplémentaire `"-e"` (par défaut, pas plus d'une erreur n'est présentée). Le commutateur `"-p"` contrôle le nombre maximal de passes que l'assembleur va tenter. Cette limite est définie par défaut sur 100. Le commutateur `"-r"` permet de définir la limite de la pile de récursivité, c'est-à-dire la profondeur maximale autorisée pour entrer des macro-instructions et inclure des fichiers source supplémentaires. Le commutateur `"-v"` peut activer l'affichage de toutes les lignes de cette pile lors du signalement d'une erreur (par défaut, l'assembleur essaie de sélectionner uniquement les lignes qui sont probablement les plus informatives, mais cette heuristique simple peut ne pas toujours être correcte). Si le commutateur `"-v"` est utilisé avec la valeur 2, il permet en outre d'afficher tous les messages affichés par les commandes du texte source en temps réel (à chaque passe consécutive). Le commutateur `"-i"` permet d'insérer n'importe quelle commande au début de la source traitée.

1. Règles de syntaxe fondamentales

Chaque commande du langage d'assemblage occupe une seule ligne de texte. Si une ligne contient le caractère point-virgule, tout ce qui va au-delà de ce caractère jusqu'à la fin de la ligne est traité comme un commentaire et ignoré par l'assembleur. La partie principale d'une ligne (c'est-à-dire excluant le commentaire) peut se terminer par une barre oblique inverse, auquel cas la ligne suivante du texte source lui sera ajoutée. Cela permet de diviser n'importe quelle commande sur plusieurs lignes, si nécessaire. Désormais, nous désignerons une ligne source comme une entité obtenue en supprimant les commentaires et en joignant les lignes de texte reliées éventuellement par des barres obliques inverses.

Le texte de la ligne source est divisé en unités syntaxiques appelées jetons. Il existe un certain nombre de caractères spéciaux qui deviennent des jetons séparés par eux-mêmes. Tout caractère de la liste ci-dessous constitue une telle unité syntaxique :

```
+-/*=<>()[]{}:?!,.|&~#‘\
```

Toute séquence contiguë (c'est-à-dire non interrompue par des espaces) de caractères autres que ceux ci-dessus devient un seul jeton, qui peut être un nom ou un nombre. L'exception à cette règle intervient lorsqu'une séquence commence par le caractère guillemet simple ou double. Cela définit une chaîne entre guillemets qui peut contenir n'importe lequel des caractères spéciaux énumérés précédemment, des espaces blancs et même des points-virgules. On retiendra qu'elle se termine uniquement lorsque le même caractère qui a été utilisé pour la

démarrer est rencontré. Les guillemets utilisés pour encadrer une chaîne ne font pas eux-mêmes partie de cette chaîne. S'il est nécessaire de définir une chaîne contenant le même caractère que celui utilisé pour l'enfermer, le caractère doit être doublé à l'intérieur de la chaîne. Une seule copie du caractère fera partie de la chaîne et la séquence continuera.

Les nombres se distinguent des noms par le fait qu'ils commencent soit par un chiffre décimal, soit par le caractère "$" suivi de tout chiffre hexadécimal. Cela signifie qu'un jeton peut être considéré comme numérique même s'il ne s'agit pas d'un nombre valide. Pour être correct, il doit être l'un des suivants : un nombre décimal (éventuellement avec la lettre "d" attachée à la fin), un nombre binaire suivi de la lettre "b", un nombre octal suivi des lettres "o" ou "q", ou un nombre hexadécimal précédé de "$" ou "0x", ou suivi du caractère "h". Étant donné que le premier chiffre d'un nombre hexadécimal peut être une lettre, il peut être nécessaire de la faire précéder du chiffre zéro afin de rendre l'expression reconnaissable en tant que nombre. Par exemple, "0Ah" est un nombre valide, tandis que "Ah" est juste un nom.

2. Identifiants de symboles

Tout nom peut devenir un symbole défini en lui attribuant une signification (c'est-à-dire une valeur). La méthode la plus simple 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 avec une valeur égale à l'adresse actuelle dans la sortie générée. Au début du texte source, cette adresse est toujours nulle. Il en résulte que, lorsque les deux commandes suivantes sont les premières du fichier source, elles définissent des symboles de valeur identique :

```
first:
second = 0
```

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

Ce qui précède les caractères ":" ou "=" dans une telle définition est un identifiant de symbole. Il peut s'agir d'un simple nom, 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 immédiatement par le caractère "?" (sans espace séparateur), ce symbole est alors réputé insensible à la casse (sinon il sera défini comme étant sensible à la casse). Cela signifie que la valeur d'un tel symbole peut être désignée (comme dans une expression à droite du caractère "=") par un nom étant une variante du nom original qui ne diffère que par la casse des lettres. Cependant, seule la casse des 26 lettres de l'alphabet anglais peut différer.

Il est possible de définir un symbole sensible à la casse qui entre en conflit avec un symbole insensible à la casse. Alors, le symbole sensible à la casse prend la priorité et le plus général est utilisé uniquement lorsque le symbole sensible à la casse correspondant n'est pas défini. Cela peut être résolu en utilisant le modificateur "?", car cela signifie toujours que le nom qui le précède fait référence au symbole d'insensibilité 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 a son propre espace de noms de descendants, appelé espace de noms enfant. Lorsque deux noms sont connectés par un point (sans espace entre les deux), un tel identificateur 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 de se référer aux descendants de descendants dans une chaîne de n'importe quelle longueur.

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

N'importe lequel des noms de cette chaîne peut éventuellement être suivi du caractère "?" pour marquer qu'il fait référence à un symbole insensible à la casse. Si "?" est inséré au milieu du nom (en le divisant effectivement en

jetons séparés), cet identifiant est considéré comme une erreur de syntaxe.

Lorsqu'un identifiant commence par un point (en d'autres termes: lorsque le nom du symbole parent est vide), il fait référence au symbole dans l'espace de nom de l'étiquette régulière la plus récente définie avant la ligne courante. Cela permet de réécrire l'exemple ci-dessus comme ceci :

```
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, de sorte que le ".x" suivant fait référence au symbole "space.x" et ensuite le ".color" fait référence au "space.color".

La commande "namespace" suivie d'un identifiant de symbole modifie l'espace de nom de base d'une section de texte source. Elle doit être associée à la commande "end namespace" plus loin dans la source pour marquer la fin d'un tel bloc. Cela peut être utilisé pour réécrire à nouveau l'exemple ci-dessus d'une manière différente :

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

Lorsqu'un nom n'est pas précédé d'un point et que, en tant que tel, il n'a pas explicitement spécifié dans quel espace de noms réside le symbole, l'assembleur recherche le symbole défini dans l'espace de noms actuel, et si aucun n'est trouvé, dans les espaces de noms consécutifs des symboles parents, à partir de l'espace de noms contenant le symbole parent de l'espace de noms actuel. Si aucun symbole défini avec un tel nom n'est trouvé, on suppose que le nom fait référence au symbole dans l'espace de noms courant (et à moins qu'il n'y ait le caractère "?" après un nom, il est supposé que le symbole est sensible à la casse). Une définition qui ne spécifie pas l'espace de noms dans lequel le nouveau symbole doit être créé, crée toujours un nouveau symbole dans l'espace de noms de base actuel.

```
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 de l'exemple ci-dessus affichent 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 avec un tel nom à rechercher dans son espace de noms, mais lorsqu'il s'agit du nom d'un symbole à définir, il est toujours créé dans l'espace de noms de base 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 qui serait recherché pour un symbole s'il y avait un nom après ce point. L'ajout d'un tel point à la fin d'un identifiant peut sembler redondant, mais il peut être utilisé pour modifier le fonctionnement de la définition d'un symbole, car il oblige l'assembleur à rechercher un symbole déjà existant qu'il peut modifier au lieu d'en créer carrément un nouveau dans l'espace de noms actuel. Par exemple, si dans la quatrième ligne de l'exemple précédent "regional." avait été mis à la place de "regional", il réécrirait une valeur du symbole "regional" d'origine au lieu de créer un nouveau symbole dans l'espace de noms enfant. De même, une définition formée de cette manière peut attribuer une nouvelle valeur à un symbole, qu'il ait été précédemment défini comme insensible à la casse ou non.

Si un identifiant est juste un point unique, par les règles ci-dessus, il fait référence à l'étiquette la plus récente qui n'a pas commencé par un point. Cela 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 "#" peut être inséré n'importe où à l'intérieur d'un identifiant sans en changer la signification. Lorsque "#" est le seul caractère séparant deux jetons de nom, ils sont interprétés comme un nom unique formé par la concaténation des jetons.

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

Cela peut également être appliqué aux nombres.

A l'intérieur d'un bloc défini avec "namespace", il n'y a initialement aucune étiquette qui serait considérée comme la base des identificateurs commençant par un point (cependant, l'étiquette qui servait cet objectif plus tôt est remise en service après "end namespace"). Une chose similaire se produit également au début du texte source, avant qu'une étiquette n'ait été définie. Ceci est lié à quelques règles supplémentaires concernant l'utilisation de points dans les identifiants.

Lorsqu'un identifiant commence par un point, mais qu'il n'y a pas d'étiquette qui serait un parent pour lui, l'identifiant fait référence au descendant d'un symbole spécial qui réside dans l'espace de noms courant mais n'a pas de nom. Si un identifiant commence par une séquence de deux points ou plus, l'identifiant fait référence au descendant d'un symbole similaire sans nom, mais il est distinct pour un nombre donné de points. Alors que l'espace de noms accessible avec un seul point de départ change chaque fois qu'une nouvelle étiquette régulière est définie, l'espace de noms spécial auquel on accède avec deux points ou plus au début d'un identificateur reste le même:

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

Dans cet exemple, la signification de l'identifiant ".child" change d'un endroit à l'autre, mais l'identifiant "..other" signifie la même chose partout.

Lorsque deux noms à l'intérieur d'un identifiant sont connectés avec 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 avant cette séquence de points. L'espace de noms enfant sans nom est choisi en fonction d'un nombre de points et dans ce cas, le nombre de points requis est augmenté d'une unité. L'exemple suivant montre les deux méthodes d'identification de ce symbole :

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

result = base.#..other
```

Le caractère "#" a été inséré dans le dernier identifiant pour une meilleure lisibilité, mais la séquence simple de trois points ferait de même.

Le symbole sans nom qui héberge un espace de nom spécial est lui-même accessible lorsqu'un identifiant se termine par une séquence de deux points ou plus – grâce à la règle selon laquelle un identifiant qui se termine par un point fait référence au symbole parent de l'espace de noms auquel on accède si il y avait un nom après ce point. Donc, dans le contexte de l'exemple précédent, le "base..." (ou "base.#..") ferait référence au parent sans nom de l'espace de noms où réside le symbole "other", et ce serait le même symbole qui serait identifié par un simple ".." dans l'espace de nom du symbole "base".

Tout identifiant peut être précédé d'un caractère "?" et un tel modificateur a un effet lorsqu'il est utilisé dans un contexte où l'identifiant pourrait signifier quelque chose de différent d'une étiquette ou d'une variable à définir. Ce modificateur supprime alors toute autre interprétation. Par exemple, identifiant commençant par "?" ne sera pas traité comme une instruction, même s'il s'agit du premier symbole de la ligne. Cela peut être utilisé pour définir une variable qui partage un nom avec une commande existante :

```
?namespace = 0
```

Si un tel identifiant modifié est utilisé à un endroit où il est évalué et non défini, il se réfère toujours au même

symbole auquel il ferait référence dans une définition. Par conséquent, à moins que l'identifiant n'utilise également un point, il fait toujours référence à un symbole dans l'espace de noms actuel.

Un nombre peut être utilisé dans le rôle d'un nom à l'intérieur d'un identifiant, mais pas lorsqu'il est placé au début, car il est alors considéré comme une valeur littérale. Cette restriction peut également être contournée en ajoutant un identifiant avec "?".

3. Définitions des symboles de base

Lorsqu'un symbole est défini comme une étiquette, il doit être la seule définition de ce symbole dans toute la source. Une valeur qui est affectée au symbole de cette manière peut être accédée à partir de n'importe quel endroit de la source, même avant que l'étiquette ne soit réellement définie. Lorsqu'un symbole est utilisé avant sa définition (cela s'appelle souvent le référencement direct), l'assembleur essaie de prédire correctement la valeur du symbole en effectuant plusieurs passages sur le texte source. Ce n'est que lorsque toutes les prédictions s'avèrent correctes que l'assembleur génère la sortie finale.

Ce type de symbole, qui ne peut être défini qu'une seule fois et donc avoir une valeur universelle qui peut toujours être référencée en avant, est appelé une constante. Toutes les étiquettes sont des constantes.

Lorsqu'un symbole est défini avec une commande "=", il peut avoir plusieurs définitions de ce type. Ce symbole est appelé variable et lorsqu'il est utilisé, la valeur de sa dernière définition est accessible. Un symbole défini avec une telle commande peut également être référencé en avant, mais seulement lorsqu'il est défini exactement une fois dans la source entière et en tant que tel a une seule valeur non ambiguë.

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

Un cas particulier de référencement direct est l'auto-référencement, c'est-à-dire lorsque la valeur d'un symbole est utilisée dans sa propre définition. L'assemblage d'une telle construction ne réussit que lorsque l'assembleur est capable de trouver une valeur stable sous une telle évaluation, résolvant efficacement une équation. Mais en raison de la simplicité de l'algorithme de résolution basé sur des prédictions, une solution peut ne pas être trouvée même lorsqu'elle existe.

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

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

Le "=:" définit un symbole de variable comme "=", mais il diffère dans la façon dont il traite la valeur précédente (quand elle existe). Tandis que "=" supprime la valeur précédente, "=:" la préserve afin qu'elle puisse être rétablie plus tard 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 restauration ("restore") peut être suivie de plusieurs identifiants de symboles séparés par des virgules, et elle rejette la dernière définition de chacun d'entre eux. Il n'est pas considéré comme une erreur d'utiliser "restaurer" avec un symbole qui n'a pas de définition active (soit parce qu'il n'a jamais été défini ou parce que toutes ses définitions ont déjà été ignorées). Si un symbole est traité avec la commande "restore", il devient une variable et ne peut jamais être référencé en avant. Pour cette raison, "restore" ne peut pas être appliqué aux constantes.

Le mot-clé "label" suivi d'un identifiant de symbole constitue une autre manière 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 a 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 au profit d'un espace blanc ordinaire, mais il est recommandé pour plus de clarté. Après un identifiant et une taille optionnelle, le mot-clé "at" peut suivre et ensuite une valeur qui doit être attribuée à l'étiquette au lieu de l'adresse courante.

```
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 métadonnées de l'élément.

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

Les métadonnées affectées à un symbole peuvent être extraites avec un opérateur spécial, défini dans la section suivante.

4. Valeurs d'expression

Dans toutes les constructions décrites jusqu'ici où une valeur d'un type quelconque était fournie, comme après la commande "=" ou après le mot-clé "at", cela pouvait être une valeur littérale (un nombre ou une chaîne entre guillemets) ou un identificateur de symbole. Une valeur peut également être spécifiée via une expression contenant des opérateurs intégrés.

Les opérateurs "+", "-" et "*" effectuent des opérations arithmétiques standard sur les entiers ("+" et "-" peuvent également être utilisés sous forme unaire – avec un seul argument). Les commandes "/" et "mod" effectuent une division avec reste, donnant respectivement un quotient ou un reste. De ces opérateurs arithmétiques, "mod" a la priorité la plus élevée (il est calculé en premier), "*" et "/" viennent ensuite, tandis que "+" et "-" sont évalués en dernier (même dans leurs variantes unaires). Les opérateurs ayant la même priorité sont évalués de gauche à droite. Les parenthèses peuvent être utilisées pour entourer des sous-expressions lorsqu'un ordre différent d'opérations est requis.

Les opérateurs "xor", "and" et "or" effectuent des opérations au niveau du bit sur les nombres. L'opérateur "xor" correspond à une addition de bits (ou exclusif), "and" à une multiplication de bits, et "or" est inclusif ou (disjonction logique). Ces opérateurs ont une priorité plus élevée que tous les opérateurs arithmétiques.

Les opérateurs "shl" et "shr" effectuent un décalage de bits du premier argument de la quantité de bits spécifiée par le second. "shl" décale les bits vers la gauche (vers les puissances supérieures de deux), tandis que "shr" décale les bits vers la droite (vers zéro), abandonnant les bits qui tombent dans la plage fractionnaire. Ces opérateurs ont une priorité plus élevée que les autres opérations binaires bit-à-bit.

Les opérateurs "not", "bsf" et "bsr" sont des opérateurs unaires avec une priorité encore plus élevée. "not" inverse tous les bits d'un nombre, tandis que "bsf" et "bsr" recherchent respectivement le bit à 1 le plus bas ou le plus élevé et donnent l'indice de ce bit en conséquence.

Toutes les opérations sur les nombres sont effectuées comme si elles étaient faites sur les représentations binaires infinies de ces nombres. Par exemple, le "bsr" avec un nombre négatif comme argument ne donne aucun résultat valide, car ce nombre a une chaîne infinie de bits à 1 s'étendant vers l'infini et en tant que tel ne contient pas de bit à 1 le plus élevé (ceci est signalé comme une erreur).

L'opérateur "bswap" permet de créer une chaîne d'octets contenant la représentation d'un nombre dans un ordre d'octet inversé (*big endian*). Le deuxième argument de cet opérateur doit être la longueur en octets de la chaîne requise. Cet opérateur a la même priorité que les opérateurs "shl" et "shr".

Lorsqu'une valeur de chaîne est utilisée comme argument pour l'une des opérations sur les nombres, elle est traitée comme une séquence de bits et automatiquement convertie en un nombre positif (étendu avec une infinité de bits à zéro). Les caractères consécutifs d'une chaîne correspondent aux bits supérieurs et inférieurs d'un nombre.

Pour reconvertir un nombre en chaîne, l'opérateur unaire "string" peut être utilisé. Cet opérateur a la priorité la plus basse possible. Donc, quand il précède une expression, tout est évalué avant la conversion. Lorsqu'une conversion dans la direction opposée est nécessaire, un simple "+" unaire suffit pour qu'une chaîne devienne un nombre.

La longueur d'une chaîne peut être obtenue avec l'opérateur unaire "lengthof". Cet opérateur ne peut être appliqué qu'à une chaîne et c'est l'un des opérateurs ayant la priorité la plus élevée.

L'opérateur "bappend" ajoute une séquence d'octets d'une chaîne donnée par le deuxième argument à la séquence d'octets du premier. Si l'un des arguments est un nombre, il est implicitement converti en chaîne. 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 dans une variable représentée par le symbole. Seules les opérations arithmétiques simples sont autorisées sur les termes d'un polynôme, et il doit rester linéaire. Ainsi, par exemple, il est seulement permis de multiplier un polynôme par un nombre, mais pas par un autre polynôme.

Il existe quelques opérateurs à priorité élevée qui permettent d'extraire les informations sur les termes du polynôme linéaire. Le polynôme devrait être le premier argument et l'index du terme, le second. L'opérateur "element" extrait la variable d'un terme polynomial (avec le coefficient 1), l'opérateur "scale" extrait le coefficient (un nombre par lequel la variable est multipliée) et l'opérateur "metadata" renvoie les métadonnées associées à la variable.

Lorsque le deuxième argument est un index supérieur à l'indice du dernier terme du polynôme, les trois opérateurs renvoient zéro. Lorsque le deuxième argument est zéro, "element" et "scale" donnent des informations sur le terme constant – "element" renvoie le 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 index de zéro renvoie la taille associée au premier argument. Cette valeur n'est définie que lorsque le premier argument est un symbole auquel est associée une taille (ou une expression arithmétique contenant un tel symbole), sinon elle vaut zéro. Il existe un opérateur unaire supplémentaire "sizeof", qui donne 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" avec l'ordre inverse des arguments. Par conséquent, lorsque "sizeof" est utilisé dans une expression, cela équivaut à écrire "0 metadataof" à sa place. Ces opérateurs ont une prédominance encore plus élevée que leurs homologues et sont associatifs à droite.

L'ordre des termes du polynôme linéaire dépend de la manière dont la valeur a été construite. Chaque opération arithmétique préserve l'ordre des termes dans le premier argument, et les termes qui n'étaient pas présents dans le premier argument sont attachés à la fin dans le même ordre dans lequel ils sont apparus dans le second argument. Cet ordre n'a d'importance que lors de l'extraction des termes avec les opérateurs appropriés.

L'opérateur "elementsof" est un autre opérateur unaire de la plus haute priorité. Il compte le nombre de termes variables d'un polynôme linéaire.

Une expression peut également contenir une valeur littérale qui définit un nombre à virgule flottante. Ce nombre doit être en notation décimale, il peut contenir le caractère "." comme une marque décimale et peut être suivi du caractère "e" puis d'une valeur décimale de l'exposant (éventuellement précédé de "+" ou "-" pour marquer son signe). Quand "." ou "e" sont présents, ils doivent être suivis d'au moins un chiffre. Le caractère "f" peut être ajouté à la fin d'une telle valeur littérale. Si un nombre ne contient ni "." ni "e", le "f" final est le seul moyen de s'assurer qu'il est traité comme une virgule flottante et non comme un simple entier décimal.

Les nombres à virgule flottante sont gérés par l'assembleur sous forme binaire. Leur portée et leur précision sont au moins aussi élevées qu'elles le sont dans le format à virgule flottante le plus long que l'assembleur est capable de produire dans la sortie.

Les opérations arithmétiques de base sont autorisées à avoir un nombre à virgule flottante comme n'importe quel argument, mais aucun des arguments ne peut alors contenir de termes non scalaires (polynômes linéaires). Le résultat d'une telle opération est toujours un nombre à virgule flottante.

L'opérateur unaire "float" peut être utilisé pour convertir une valeur entière en virgule flottante. Cet opérateur a la priorité la plus élevée.

L'opérateur "trunc" est un autre unaire avec la priorité la plus élevée qui peut être appliqué aux nombres à virgule flottante. Il extrait la partie entière d'un nombre (c'est une troncature vers zéro) et le résultat est toujours un entier simple, pas un nombre à virgule flottante. Si l'argument était déjà un entier simple, cette opération le laisse inchangé.

L'opérateur "bsr" peut être appliqué aux nombres à virgule flottante. Il renvoie l'exposant de ce nombre, qui est l'exposant de la plus grande puissance de deux qui n'est pas supérieure au nombre donné. Le signe de la valeur à virgule flottante n'affecte pas le résultat de cette opération.

Il est également permis d'utiliser un nombre à virgule flottante comme premier argument des opérateurs "shl" et "shr". Le nombre est ensuite multiplié ou divisé par la puissance de deux spécifiée par le deuxième argument.

5. Classes de symboles

Il existe trois classes distinctes de symboles, déterminant la position sur la ligne source à laquelle le symbole peut être reconnu. Un symbole appartenant à la classe d'instructions n'est reconnu que lorsqu'il est le premier identifiant de la commande, tandis qu'un symbole de la classe d'expression n'est reconnu que lorsqu'il est utilisé pour fournir une valeur d'arguments à une commande.

Tous les types de définitions décrits dans les sections précédentes créent les symboles de classe d'expression. Le "label" et "restore" sont des exemples de symboles intégrés appartenant à la classe d'instructions.

Dans n'importe quel espace de noms, il est permis aux symboles de différentes classes de partager le même nom. Par exemple il est possible de définir l'instruction nommée "shl", alors qu'il y a aussi un opérateur avec le même nom – mais un opérateur appartient à la classe d'expression.

Il est même possible qu'une seule ligne contienne le même identifiant signifiant des choses différentes selon sa position :

```
?restore = 1
restore restore ; remove the value of the expression-class symbol
```

La troisième classe de symboles est constituée par les instructions étiquetées. Un symbole appartenant à cette classe ne peut être reconnu que lorsque 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 la seconde. Si nous traitons "=" comme un type spécial d'identifiant, il peut servir d'exemple d'instruction étiquetée.

L'assembleur contient des symboles intégrés de toutes les classes. Leurs noms sont toujours insensibles à la casse et ils peuvent être redéfinis, mais il n'est pas possible de les supprimer. Lorsque toutes les valeurs d'un tel symbole sont supprimées avec une commande telle que "restore", la valeur intégrée persiste.

Les règles concernant l'espace de noms s'appliquent également aux symboles de toutes les classes. Par exemple, le symbole de 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 toutefois de noter que lorsqu'un espace de noms est spécifié via son symbole parent, il s'agit toujours d'un symbole appartenant à la classe d'expression. Il n'est pas possible de faire référence à un espace de noms enfant d'une instruction, uniquement à l'espace de noms appartenant au symbole de classe d'expression avec le même nom.

```
xor?.mask? := 10101010b
a = XOR.MASK    ; symbol in the namespace of built-in case-insensitive "XOR"

label?.test? := 0
a = LABEL.TEST  ; undefined unless "label?" is defined
```

Ici, l'espace de noms contenant "test" appartient à un symbole de classe d'expression, pas à l'instruction existante "label". Lorsqu'il n'y a pas de symbole de classe d'expression qui correspondrait au spécificateur "LABEL", l'espace de noms choisi est celui qui appartiendrait au symbole sensible à la casse d'un tel nom. Le "test" n'est donc pas trouvé, car il a été défini dans un autre espace de noms – celui de "label" insensible à la casse.

6. Génération de données

L'instruction "db" permet de générer des octets de données et de les mettre en sortie. Elle doit être suivie d'une ou plusieurs valeurs, séparées par des virgules. Lorsque la valeur est numérique, elle définit un seul octet. Lorsque la valeur est une chaîne, elle met la chaîne d'octets en sortie.

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

Le mot-clé "dup" peut être utilisé pour générer la même valeur plusieurs fois. Le "dup" doit être précédé d'une expression numérique définissant le nombre de répétitions et la valeur à répéter doit suivre. Une séquence de valeurs peut également être dupliquée de cette manière, dans ce cas, "dup" doit être suivi de la séquence entière entre parenthèses (avec des valeurs séparées par des virgules).

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

Lorsqu'un identifiant spécial composé d'un seul caractère "?" est utilisé comme valeur dans les arguments de "db", il réserve un seul octet. Cela fait avancer l'adresse dans la sortie où les données suivantes vont être placées, mais les octets réservés ne sont pas générés eux-mêmes à moins qu'ils ne soient suivis par d'autres données. Par conséquent, si les octets sont réservés en fin de sortie, ils n'augmentent pas la taille du fichier généré. Ce type de données est qualifié de *non-initialisée*, tandis que toutes les données régulières sont dites *initialisées*.

L'instruction "rb" réserve un 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 de 1.

```
some db sizeof some      ; génère un octet avec la valeur 1
```

Les instructions "dw", "dd", "dp", "dq", "dt", "ddq", "dqq" et "ddqq" sont analogues à "db" avec une unité de données de taille différente. L'ordre des octets dans une unité donnée générée est toujours le *little-endian*. Lorsqu'une chaîne d'octets est fournie comme valeur à l'une de ces instructions, les données générées sont étendues avec zéro octet jusqu'à la longueur qui est le 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'unité associées à toutes ces instructions sont répertoriées dans le tableau 1.

Les instructions "dw", "dd", "dq", "dt" et "ddq" autorisent les nombres à virgule flottante comme unités de données. Un tel nombre est ensuite converti au format à virgule flottante approprié pour une taille donnée.

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

```
emit 2: 0,1000,2000      ; génère trois valeurs de 16 bits
```

L'instruction "file" lit les données d'un fichier externe et les écrit dans la sortie. L'argument doit être une chaîne contenant le chemin d'accès au fichier, il peut éventuellement être suivi de ":" et la valeur numérique spécifiant un décalage dans le fichier, ensuite il peut être suivi d'une virgule et de la valeur numérique spécifiant le nombre d'octets à copier.

```
file 'data.bin'                ; insère la totalité du fichier
excerpt file 'data.bin':10h,4  ; insère 4 octets sélectionnés
```

Tableau 1 Directives de données

Taille (octets)	Génération de données	Réservation de données
1	db 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" entraîne l'assemblage d'un bloc de texte source uniquement sous certaines conditions, déterminées par une expression logique qui est un argument de cette instruction. La commande "else if" dans les lignes suivantes termine le bloc assemblé conditionnellement précédent et en ouvre un nouveau, assemblé uniquement lorsque les conditions précédentes ne sont pas remplies et que la nouvelle condition (un argument pour "else if") est vraie. La commande "else" termine le bloc assemblé conditionnellement précédent et commence un bloc qui est assemblé uniquement lorsqu'aucune des conditions précédentes n'était vraie. La commande "end if" doit être utilisée pour terminer la construction entière. Il peut y avoir plusieurs ou aucune commande "else if" à l'intérieur et pas plus d'une commande "else".

Une expression logique est une entité syntaxique distincte des expressions de base décrites précédemment. Une expression logique se compose de valeurs logiques connectées à des opérateurs logiques. Les opérateurs logiques sont: unaire "~" pour la négation, "&" pour la conjonction et "|" pour l'alternative. La négation est évaluée en premier, tandis que "&" et "|" sont simplement évalués de gauche à droite, sans priorité les uns sur les autres.

Une valeur logique dans sa forme la plus simple peut être une expression de base. Elle correspond alors à une condition vraie si et seulement si sa valeur n'est pas une constante nulle. Une autre façon de créer une valeur

logique consiste à comparer les valeurs de deux expressions de base avec l'un des opérateurs suivants: "=" (égal), "<" (inférieur à), ">" (supérieur à), "<=" (inférieur ou égal), ">=" (supérieur ou égal), "<>" (différent).

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

Lorsque les polynômes linéaires sont comparés de cette manière, la valeur logique n'est valide que lorsqu'ils sont comparables, c'est-à-dire qu'ils ne diffèrent qu'en termes constants. Sinon, la condition telle que l'égalité n'est ni universellement vraie ni universellement fausse, car elle dépend des valeurs substituées aux variables, et l'assembleur le signale cela comme une erreur.

L'opérateur "relativeto" crée une valeur logique qui n'est vraie que lorsque la différence des valeurs comparées ne contient aucun terme variable. Par conséquent, il peut être utilisé pour vérifier si deux polynômes linéaires sont comparables. La condition "relativeto" n'est vraie que lorsque les deux polynômes comparés ont les mêmes termes variables.

Étant donné que les expressions logiques sont évaluées paresseusement, il est possible de créer une seule condition qui ne provoquera pas d'erreur lorsque les polynômes ne sont pas comparables, mais 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 ce 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 avec une expression de base qui la suit. Cette condition est vraie lorsque l'expression ne contient pas de symboles qui n'ont pas de définition accessible. L'expression n'est testée que pour la disponibilité de ses composants et 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 "definite" doit être utilisé à la place, car il vérifie si tous les symboles de l'expression de base qui suit ont été définis plus tôt.

L'expression de base qui suit "defined" est également autorisée à être vide et la condition est alors trivialement satisfaite. Cela ne s'applique pas à "definite".

L'opérateur "used" forme une valeur logique s'il est suivi d'un identifiant unique. Cette condition est vraie lorsque la valeur du symbole spécifié a été utilisée n'importe où dans la 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 entre les commandes "macro" et "end macro" devient le texte de la macro-instruction et cette séquence de lignes est assemblée à la place de la commande d'origine qui commence par l'identifiant de l'instruction ainsi défini.

```
macro null
	db 0
end macro

null		; "db 0" est assemblé ici
```

La macro-instruction est autorisée à avoir des arguments uniquement lorsque la définition les contient. Après la "macro" et l'identifiant du symbole défini peut éventuellement suivre une liste de noms simples séparés par des virgules. Ces noms définissent les paramètres de macro-instruction. Lorsque cette instruction est alors utilisée, elle peut être suivie d'au plus le même nombre d'arguments séparés par des virgules, et leurs valeurs sont affec-

tées aux paramètres consécutifs. Avant qu'une ligne de texte à l'intérieur de la macro-instruction ne soit interprétée, les jetons de nom qui correspondent à l'un des paramètres sont remplacés par leurs valeurs affectées.

```
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 et pas nécessairement une expression correcte. Si une ligne appelant la macro-instruction contient moins d'arguments que le nombre de paramètres définis, les paramètres en excès reçoivent des valeurs vides.

Lorsqu'un nom de paramètre est défini, il peut être suivi du caractère "?" pour indiquer qu'il est insensible à la casse, de manière analogue à un nom dans un identifiant de symbole. Il ne doit y avoir aucun espace entre le nom et "?". La définition d'un paramètre peut également être suivie de "*" pour indiquer qu'elle nécessite une valeur qui n'est pas vide, ou alternativement par le caractère ":" suivi d'une valeur par défaut, qui est affectée au paramètre au lieu 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 de macro-instruction doit contenir une virgule, l'argument entier doit être placé entre les crochets angulaires "<" et ">" (qui ne font pas partie de la valeur). Si un autre crochet "<" est rencontré à l'intérieur de cette valeur, il doit être équilibré avec le crochet ">" complémentaire 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 que ce paramètre doit se voir attribuer une valeur contenant toute la partie restante de la ligne, même s'il définirait normalement plusieurs arguments. Par conséquent, lorsque la macro-instruction n'a qu'un seul paramètre suivi de "&", la valeur de ce paramètre est le texte entier des arguments suivant 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 aucun espace entre les deux), le texte de la valeur est incorporé 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 elle déclare que les noms de cette liste doivent dans le contexte de la macro-instruction actuelle être interprétés comme appartenant à un espace de noms spécial associé à cette macro-instruction au lieu de l'espace de noms de base actuel. Cela permet de créer des symboles uniques à chaque fois que la macro-instruction est appelée. Une telle déclaration définit des paramètres supplémentaires avec les noms spécifiés et n'affecte donc que les utilisations des noms qui suivent dans la même macro-instruction. Déclarer le même nom comme local plusieurs fois dans la même macro-instruction ne donne aucun effet supplémentaire.

```
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 "local" est remplacé par un texte qui contient le même nom que le nom du paramètre, mais a ajouté des informations de contexte qui le font être identifié comme appartenant à l'espace de noms local unique associé à l'instance de macro-instruction. Ce type d'informations de contexte sera étudié plus en détail dans la section sur les variables symboliques.

Un symbole qui est local par rapport à une macro-instruction n'est jamais considéré comme l'étiquette la plus récente qui est à la base des symboles commençant par un point. De plus, son espace de noms descendant est déconnecté de l'arborescence principale des symboles. Donc si la commande "namespace" était utilisée avec un symbole local comme argument, les symboles de l'arborescence principale ne seraient plus visibles (y compris toutes les instructions nommées de l'assembleur, même des commandes comme "end namespace").

Tout comme un symbole d'expression peut être redéfini et faire référence à sa valeur précédente dans la définition du nouveau, les macro-instructions peuvent également être redéfinies et utiliser la valeur précédente de ce symbole d'instruction dans son texte :

```
macro zero
        db 0
end macro

macro zero name
        label name:byte
        zero
end macro

zero x
```

Et tout comme les autres symboles, une macro-instruction peut être référencée en avant lorsqu'elle est définie exactement une fois dans toute la source.

La commande "purge" rejette la définition d'un symbole tout comme "restore", mais elle le fait pour le symbole de la classe d'instructions. Elle se comporte de la même manière que "restore" dans tous les autres aspects. Une macro-instruction peut supprimer sa propre définition avec "purge".

Il est possible pour une macro-instruction d'utiliser sa propre valeur de manière récursive, mais pour éviter une récursion infinie par inadvertance, cette fonctionnalité n'est disponible que lorsque la macro-instruction est marquée comme telle en suivant son identifiant avec le caractère ":".

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

En plus de permettre la récursivité, une telle macro-instruction se comporte comme une constante. Elle ne peut pas être redéfinie et "purge" ne peut pas lui être appliqué.

Une macro-instruction peut à son tour définir une autre macro-instruction ou un certain nombre d'entre elles. Les blocs désignés par "macro" et "end macro" doivent être correctement imbriqués l'un dans l'autre pour qu'une telle 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 que la macro-instruction génère une commande "macro" ou "end macro" non appairée, cela peut être fait avec une instruction spéciale "esc". Son argument devient une partie de la macro-instruction, mais n'est pas pris en compte lors du comptage des paires imbriquées "macro" et "end macro".

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

xmacro text
        db 'x
end macro
```

Si l'instruction "esc" est placée dans une définition imbriquée, elle n'est pas traitée jusqu'à ce que la macro-instruction la plus interne soit définie. Cela permet à une définition contenant "esc" d'être placée dans une autre macro-instruction sans avoir à répéter "esc" pour chaque niveau d'imbrication.

Lorsqu'un identifiant de macro-instruction dans sa définition est suivi du caractère "!", cela définit une macro-instruction inconditionnelle. Il s'agit d'un type spécial de symbole de classe d'instructions, qui est évalué même dans les endroits où l'assemblage est suspendu – comme à l'intérieur d'un bloc conditionnel dont la condition est fausse, ou à l'intérieur d'une définition d'une autre macro-instruction. Cela permet de définir des instructions qui peuvent être utilisées là où autrement une "end if" ou une "end macro" déclarée directement serait requise, 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" aurait disparu.

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 comme "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 l'espace de noms "end?". De plus, toute instruction définie dans l'espace de noms "end?" peut alors être appelée avec la commande "end". Cette variante légèrement modifiée de l'exemple ci-dessus met ces faits à profit :

```
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 dans l'espace de noms "else?".

Lorsqu'un identifiant constitué d'un seul caractère "?" est utilisé comme symbole d'instruction dans la définition d'une macro-instruction, il définit une instruction spéciale qui est ensuite appelée chaque fois qu'une ligne à assembler ne contient pas d'instruction inconditionnelle, et le texte complet de la ligne devient les arguments de cette macro-instruction. Ce symbole spécial peut également être défini comme une instruction inconditionnelle, puis il est appelé pour chaque ligne suivante sans exception. Cela permet de remplacer complètement le processus d'assemblage sur des parties du texte. L'exemple suivant définit une macro-instruction qui permet de définir un bloc de commentaires en sautant toutes les lignes de texte jusqu'à ce qu'il rencontre une ligne de contenu égal à l'argument donné 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 peut être utilisé pour définir une instruction spéciale qui n'est appelée qu'en dernier recours, sur des lignes ne contenant aucune instruction reconnaissable. Cela permet d'intercepter des lignes qui autrement seraient rejetées avec le message "*illegal instruction*" en raison d'une syntaxe inconnue.

L'instruction "mvmacro" prend deux arguments, toutes deux identifiant des symboles de classe d'instructions. La définition d'une macro-instruction spécifiée par le deuxième argument est déplacée vers le symbole identifié par le premier. Pour le deuxième symbole, l'effet de cette commande est le même que celui de "purge". Cela permet de renommer efficacement une macro-instruction, ou de la désactiver temporairement uniquement pour la ramener plus tard. Les symboles affectés par cette opération deviennent des variables et ne peuvent pas être référencés en avant.

9. Macro-instructions étiquetées

La commande "struc" permet de définir une instruction étiquetée sous la forme d'une macro-instruction. À l'exception du fait qu'une telle définition doit être fermée par "end struc" au lieu de "end macro", ces macroinstructions sont définies de la même manière qu'avec la commande "macro". Une instruction étiquetée est évaluée lorsque le premier identifiant d'une commande n'est pas une instruction et que le second identifiant est de la classe d'instructions étiquetée :

```
struc some
      db 1
end struc

get some        ; "db 1" est assemblé ici
```

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 se réfèrent à l'espace de nom de l'étiquette avec lequel l'instruction a été étiquetée.

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

my POINT        ; définit my.x et my.y
```

Notez que le symbole parent, qui peut être référencé par l'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 l'exemple qui suit :

```
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 d'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 étiquetée spéciale 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 intercepter 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
```

Comme pour "macro", cette variante spéciale ne remplace pas les instructions étiquetées inconditionnelles à moins qu'elle ne soit elle-même inconditionnelle.

Tandis que "." fournit une méthode efficace pour accéder au symbole d'étiquette, parfois il peut être nécessaire pour traiter le texte réel de l'étiquette. Un paramètre spécial peut être défini à cet effet et 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 peut contenir n'importe quel texte (même vide) et lorsqu'elle est utilisée dans une expression, elle 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.

Cela peut conduire à des résultats différents de ceux lorsqu'une variable standard définie avec "=" est utilisée, 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" reçoit la valeur 12, la valeur de "y" est 8. Cela montre que l'utilisation de tels symboles peut conduire à des interactions involontaires et par conséquent, les définitions de ce type doivent être évitées à moins qu'elles ne soient vraiment nécessaires.

Le "equ" permet les redéfinitions, et il préserve la valeur précédente du symbole de manière analogue à la commande "=:", de sorte que la valeur précédente peut être ramenée avec l'instruction "restore". Pour remplacer la valeur symbolique (de la même manière que "=" écrase la valeur normale), la commande "reequ" doit être utili-

sée à la place 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. Par conséquent, elle peut effectivement devenir un lien fiable vers la valeur d'un autre symbole, même si elle est utilisée dans un contexte différent (cela inclut le changement de l'espace de noms de base ou d'un symbole référencé par un point de départ) :

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

Il convient de noter que le même processus est appliqué aux arguments de toute macro-instruction lorsqu'ils deviennent des paramètres prétraités. Si lors de 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 dans lequel le nom donné est interprété, sans modifier le texte du nom. Cependant, un tel contexte modifié n'est pas pertinent si la valeur de paramètre est insérée au milieu ou à la fin d'un identifiant complexe, car c'est la structure d'un identifiant qui dicte la façon dont ses parties ultérieures sont interprétées et seul le contexte d'une première partie compte. Par exemple, ajouter au début le nom d'un paramètre avec le caractère "#" va amener l'identifiant à utiliser le contexte courant au lieu du contexte porté par le texte de ce paramètre, car le contexte initial de l'identifiant est alors le contexte associé au texte "#".

Si le texte suivant "equ" contient des identifiants de variables symboliques connues, chacune d'elles est remplacée par son contenu et c'est ce texte traité qui est affecté au symbole nouvellement défini.

Le "define" est une instruction régulière qui crée également une valeur symbolique, mais contrairement à "equ", elle n'évalue pas les variables symboliques dans le texte affecté. Elle doit être suivie d'un identifiant de symbole à définir puis du texte de la valeur.

La différence entre "equ" et "define" n'est souvent pas perceptible, car lorsqu'elles sont utilisées dans l'expression finale, les variables symboliques sont évaluées de manière imbriquée jusqu'à ce que seuls les constituants utilisables des expressions soient laissés. Une utilisation possible de "define" est de 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" se présenteront dans les sections suivantes, avec l'introduction d'autres instructions qui opèrent sur des valeurs symboliques.

Le "define", comme "equ", préserve la valeur précédente du symbole. Le "redefine" est une variante de cette instruction qui rejette la valeur antérieure, de manière analogue à "reequ".

Notez que si les variables symboliques appartiennent à la classe d'expression des symboles. Leur état ne peut pas être déterminé avec 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 valeur correspondante. Par conséquent, l'opérateur suivi d'un identifiant de variable symbolique va être appliqué au contenu de cette variable, quelle qu'elle soit. Par exemple, si une variable symbolique est créée qui est un lien vers un symbole régulier, alors tout opérateur comme "defined" suivi de l'identifiant de ladite variable symbolique va déterminer le statut d'un symbole lié, pas une variable de lien.

Contrairement à la valeur d'une variable symbolique, le corps d'une macro-instruction en tant que tel ne contient aucun contexte (bien qu'il puisse contenir des extraits de texte provenant de paramètres remplacés et, de ce fait, être associés à un certain contexte). De plus, si une macro-instruction est déroulée au moment où une autre est définie (cela ne peut se produire que lorsque la macro-instruction appelée est inconditionnelle), aucune information contextuelle n'est ajoutée aux arguments, pour aider à préserver cette absence de contexte.

Il est également possible de forcer un argument de macro à n'ajouter aucune information contextuelle à son texte. Le nom d'un tel 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, avec le nombre de répétitions 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" provoque l'assemblage du bloc d'instructions à plusieurs reprises tant que la condition spécifiée par son argument est vraie. Son argument doit être une expression logique, comme un argument pour "if" ou "assert". Le bloc doit être fermé avec la commande "end while".

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

Le "%" est un paramètre spécial qui est prétraité à l'intérieur du bloc d'instructions répété et est remplacé par un nombre décimal correspondant au nombre de répétitions en cours (commençant par 1). Il fonctionne de la même manière qu'un paramètre de macro-instruction. Il est donc remplacé par sa valeur avant que la commande réelle ne soit traitée et peut donc être utilisé pour créer des identifiants de symbole contenant le nombre comme partie du nom :

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

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

L'instruction "repeat" peut avoir des arguments supplémentaires, séparés par des virgules, contenant chacun un nom de paramètres supplémentaires propres à ce bloc. Chacun des noms peut être suivi du caractère ":" et de l'expression spécifiant la valeur de base à partir de laquelle le paramètre va commencer à compter les répétitions. Cela permet de modifier facilement l'échantillon précédent pour définir la plage de symboles de "f0" à "f15" :

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

Le "%%" est un autre paramètre spécial qui a une valeur égale au nombre total de répétitions prévues. Ce paramètre n'est pas défini dans le bloc "while". L'exemple qui suit l'utilise pour créer la séquence d'octets avec des valeurs descendant de 255 à 0 :

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

L'instruction "break" permet d'arrêter la répétition prématurément. Lorsqu'elle est rencontrée, elle fait sauter le reste du bloc répété et aucune autre répétition n'est exécutée. Elle peut être utilisée pour arrêter 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 se propose de trouver la racine carrée de la valeur du symbole "x", qui est supposée définie 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" (avec un synonyme "irp") répète le bloc d'instructions tout en parcourant la liste de valeurs séparées par des virgules. Le premier argument pour "itérate" doit être un nom de paramètre, suivi d'une virgule puis d'une liste de valeurs. Lors de chaque itération, le paramètre reçoit une des valeurs de la liste :

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

Comme c'est le cas dans le cas d'un argument de macro-instruction, la valeur du paramètre qui contient des virgules doit être encadrée de caractères "<" et ">". Il est également possible de placer le premier argument à itérer ("iterate") avec "<" et ">", afin de définir plusieurs paramètres. La liste de valeurs est ensuite divisée en section contenant autant de valeurs qu'il y a de paramètres, et chaque itération opère sur une de ces sections, 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 dans le cas de macro-instructions, être suivi de "*" pour exiger que le paramètre ait une valeur qui n'est pas vide, ou ":" et une valeur par défaut. Si une instruction "iterate" se termine par une virgule qui n'est suivie par rien d'autre, elle n'est pas interprétée comme une valeur vide supplémentaire. Pour mettre une valeur vide à la fin de la liste, un "<>" englobant vide doit être utilisé.

L'instruction "break" ainsi que les paramètres "%" et "%%" peuvent être utilisés à l'intérieur du bloc "iterate" avec les mêmes effets que dans le cas de "repeat".

Le "indx" est une instruction qui ne peut être utilisée qu'à l'intérieur d'un bloc itéré et il change les valeurs de tous les paramètres itérés en celles correspondant à l'itération avec le nombre spécifié par l'argument à "indx" (mais quand l'itération suivante est démarré, les valeurs des paramètres sont à nouveau affectées de la manière normale). 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, tout le 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 "itérate" n'attachent pas le contexte aux valeurs itérées, mais ne suppriment pas non plus le contexte d'origine si celui-ci est déjà attaché au texte des arguments. Donc, si les valeurs données à "iterate" ont elles-mêmes été créées à partir d'un autre paramètre qui a conservé le contexte d'origine pour les identifiants de symboles (comme le paramètre de macro-instruction), alors ce contexte est conservé, mais sinon, "iterate" définit juste une substitution de texte brut.

Les paramètres définis par des instructions comme "iterate" ou "repeat" sont traités partout dans le texte du bloc associé, mais avec quelques limitations si le bloc est défini en partie par le texte de macro-instruction et en partie à d'autres endroits. Dans ce cas, les paramètres ne sont accessibles que dans les parties du bloc définies au même endroit que la commande initiale.

Chaque fois qu'un paramètre est défini, son nom peut avoir le caractère "?" qui lui est attaché pour préciser qu'il est insensible à la casse. Cependant, lorsque des paramètres sont reconnus à l'intérieur de la ligne prétraitée, peu importe qu'ils soient suivis de "?". Le seul modificateur reconnu par le préprocesseur lorsqu'il remplace le paramètre par sa valeur est le caractère "‘".

Les instructions répétitives avec "if" appartiennent à un groupe appelé directives de contrôle. Ce sont les instructions qui contrôlent le flux de l'assemblage. Chacun d'eux définit son propre bloc d'instructions subordonnées, fermé avec la commande "end" correspondante, et si ces blocs sont imbriqués les uns dans les autres, il doit toujours s'agir d'une imbrication appropriée dans le sens où le bloc interne doit toujours être fermé avant le bloc externe. Toutes les directives de contrôle sont donc les instructions inconditionnelles – elles sont reconnues même lorsqu'elles sont à l'intérieur d'un bloc autrement ignoré.

Le «report» est une autre directive de contrôle, qui entraîne l'assemblage ultérieur d'un bloc d'instructions, lorsque tout le texte source suivant a déjà 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 la source entière ait été traitée, afin qu'elle puisse accéder à la valeur finale de la variable "counter".

L'assemblage du texte source qui suit "postpone" comprend l'assemblage de tous les blocs supplémentaires déclarés avec "postpone". Donc s'il y a plusieurs blocs de ce type, ils sont assemblés dans l'ordre inverse. Celui qui a été déclaré en dernier est assemblé en premier lorsque la fin du texte source est atteinte.

Lorsque la directive "postpone" est fournie avec un argument consistant en un seul caractère "?", il indique à l'assembleur que le bloc contient des opérations qui ne devraient affecter aucune des valeurs définies dans la source principale et ainsi l'assembleur peut s'abstenir de les évaluer jusqu'à ce que toutes les autres valeurs aient été résolues avec succès. Ces blocs sont traités même plus tard que ceux déclarés par "postpone" sans argument. Ils peuvent être utilisés pour effectuer certaines tâches de finalisation, comme le calcul d'une checksum du code assemblé.

Le "irpv" est une autre instruction répétitive et un itérateur. Il n'a que deux arguments, le premier étant un nom de paramètre et le second un identifiant d'une variable. Il parcourt toutes les valeurs empilées de la variable symbolique, en commençant par la plus ancienne (cela ne s'applique qu'aux valeurs définies précédemment dans la 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 existe 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 début de tout traitement de commande. Cela permet, par exemple, de redéfinir une valeur régulière qui est 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 pour limiter l'itération à la dernière valeur de la variable symbolique. Dans la section suivante, une meilleure solution au même problème sera présentée.

Lorsqu'une variable passée à "irpv" a une valeur qui n'est pas symbolique, le paramètre reçoit un texte qui produit la même valeur lors du calcul. Lorsque la valeur est un nombre positif, le paramètre est remplacé par sa représentation décimale (de la même manière que le paramètre "%" est traité), sinon le paramètre est remplacé par un identifiant d'un symbole proxy contenant la valeur de la pile.

La directive "outscope" est disponible pendant le traitement de toute macro-instruction et modifie la commande qui suit dans la même ligne. Si la commande entraîne la définition de paramètres, ils ne sont pas créés dans le contexte de la macro-instruction actuellement traitée mais dans le contexte 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 qui définissent les paramètres d'un bloc de texte. Parce que "outscope" doit être présent dans le texte d'une macro-instruction spécifique qui le requiert, il est recommandé de l'utiliser en conjonction avec "esc" comme dans l'exemple ci-dessus. Cela garantit qu'il est traité de la même manière même lorsque l'ensemble la définition est placée dans une autre macro-instruction.

12. Correspondance de paramètres

"match" est une directive de contrôle qui entraîne l'assemblage de son bloc d'instructions uniquement lorsque le texte spécifié par son deuxième argument correspond au modèle donné par le premier. Un texte est séparé d'un motif par une virgule et comprend tout ce qui suit ce séparateur jusqu'à la fin de la ligne.

Chaque caractère spécial (à l'exception des "," et de "=", qui ont une signification spécifique dans le modèle) est mis en correspondance littéralement – il doit être associé à un jeton identique dans le texte. Dans l'exemple suivant, le contenu du premier bloc est assemblé, tandis que le contenu du second ne l'est pas.

```
match +,+
        assert 1        ; correspondance (match) positive
end match

match +,-
        assert 0        ; correspondance (match) negative
end match
```

Les chaînes entre guillemets sont également mises en correspondance littéralement, mais les jetons de nom dans le modèle sont traités différemment. Chaque nom agit comme un joker et peut correspondre à n'importe quelle séquence de jetons qui n'est pas vide. Si la correspondance réussit, les paramètres portant ces noms sont créés et chacun reçoit une valeur égale au texte avec lequel le caractère générique a été mis en correspondance.

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

Un nom de paramètre dans un modèle peut avoir un caractère supplémentaire "?" qui lui est attaché pour indiquer qu'il s'agit d'un nom insensible à la casse.

Le caractère "=" fait correspondre littéralement le jeton qui le suit. Il permet d'effectuer la mise en correspondance des jetons de nom, ainsi que des caractères spéciaux qui auraient autrement une signification différente, comme "," ou "=" ou "?" après un nom.

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

Si "=" est suivi du jeton de nom avec le caractère "?" qui lui est attaché, cet élément est mis en correspondance littéralement mais de manière insensible à la casse:

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

```
        db a            ; db 8
end match
```

Lorsqu'il y a de nombreux caractères génériques dans le modèle, chaque caractère consécutif est mis en correspondance avec le moins de jetons possible et le dernier prend ce qui reste. Si les caractères génériques se suivent sans aucun élément correspondant littéralement entre eux, le premier correspond 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 a besoin d'au moins un jeton pour ne pas être vide. Dans l'exemple suivant, il y a des contraintes supplémentaires, mais les mêmes règles générales s'appliquent et le premier caractère générique consomme le moins 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
```

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

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

match ++,+ +
        assert 0        ; correspondance négative
end match

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

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

La présence d'espaces blancs dans le texte devient obligatoire lorsque le motif contient le caractère "=" suivi d'un espace blanc :

```
match += +, ++
        assert 0        ; correspondance negative
end match

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

La commande "match" est analogue à "if" en ce qu'elle permet d'utiliser le "else" ou "else match" pour créer une sélection de blocs à partir desquels 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 mélanger les conditions "if" et "match" dans une séquence de blocs "else". L'ensemble de la construction doit être fermé avec la commande "end" correspondant à celle des deux utilisée en dernier :

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

Le "match" est capable de reconnaître des variables symboliques et avant le début de la mise en correspondance, leurs identifiants dans le texte du deuxième argument sont remplacés par les valeurs correspondantes (tout comme 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 utilisée 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 utilisée 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 eux est que "irpv" exécuterait son bloc même pour une valeur vide, alors que dans le cas de "match", le bloc "else" devrait être ajouté pour gérer un texte vide.

Lorsque l'évaluation des variables symboliques dans le texte correspondant n'est pas souhaitable, un symbole créé avec "define" peut être utilisé comme proxy pour conserver 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 du texte en le dotant d'un contexte local. Mais lorsque la valeur de "define" provient d'un paramètre de macro-instruction (comme dans l'exemple ci-dessus), elle porte déjà son contexte d'origine et "define" ne le modifie pas.

La directive "rawmatch" (avec son synonyme "rmatch") est très similaire à "match", mais elle fonctionne sur le texte brut du deuxième argument. Non seulement elle n'évalue pas les variables symboliques, mais elle supprime également le texte de tout contexte supplémentaire qu'elle aurait pu transporter.

```
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 en vigueur pour la ligne appelant la macro-instruction "has" s'il n'était pas reconverti dans le texte brut par "rmatch".

13. Zones de sortie

L'instruction "org" démarre une nouvelle zone de sortie. Le contenu de cette zone est écrit dans le fichier destination à côté des données précédentes, mais les adresses de la nouvelle zone sont basées sur la valeur spécifiée dans l'argument "org". La zone se ferme automatiquement au démarrage de la suivante ou à la fin de la source.

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

Le "$" est un symbole intégré de la classe d'expression qui est toujours égal à la valeur de l'adresse courante. Par conséquent, la définition d'une constante avec la valeur spécifiée par le symbole "$" équivaut à définir une étiquette au même point :

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

Le symbole "$$" est toujours égal à la base de l'espace d'adressage courant, donc dans la zone commencée par "org", il a la même valeur que l'adresse de base de l'argument de "org". La différence entre "$" et "$$" est donc la position courante par rapport au début de la zone :

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

Le symbole "$@" correspond à l'adresse de base du bloc actuel de données non initialisées. Lorsqu'il n'y avait pas de telles données définies juste avant la position courante, cette valeur est égale à "$", sinon elle est égale à "$" moins la longueur desdites données à l'intérieur de l'espace d'adressage courant. Notez que les données réservées ne comptent plus comme telles lorsqu'elles sont suivies d'une donnée initialisée.

L'instruction "section" est similaire à "org", mais elle supprime en plus toutes les données réservées qui la précèdent de manière analogue à la façon dont les données non initialisées ne sont pas écrites dans la sortie lorsqu'elles sont à la fin du fichier. L'instruction "section" peut donc être suivie de définitions de données initialisées sans provoquer l'initialisation des données précédemment réservées avec des zéros et l'écriture en sortie. Dans l'exemple qui suit, seul le premier des trois tampons réservés est réellement converti en données mises à zéro et écrit dans la sortie, car il est suivi de certaines données initialisées. Le second est coupé à cause de la "section", et le troisième est coupé car il se trouve à la fin du fichier :

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

org 400h
data dw 2
buffer2 rb 20h          ; non-présent dans la sortie

section 1000h
data3 dw 3
buffer3 rb 30h          ; non-présent dans la sortie
```

Le "$%" est un symbole intégré égal au décalage dans le fichier de sortie auquel les données initialisées seraient générées si elles étaient définies à ce stade. Le symbole "$%%" est le décalage courant dans le fichier de sortie. Ces deux valeurs ne diffèrent que lorsqu'elles sont utilisées après que certaines données ont été réservées – le

"$%" est alors plus grand que "$%%" par la longueur des données unifiées qui seraient générées en sortie si elles devaient être suivies par certaines initialisé.

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

Les valeurs dans les commentaires de l'exemple ci-dessus supposent que la 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. Ce type de zone doit résider entre les commandes "virtual" et "end virtual", et après sa fermeture, le générateur de sortie revient à la zone sur laquelle il fonctionnait auparavant, avec la position et l'adresse identiques à celles qu'il y avait juste avant l'ouverture le bloc "virtual". Cela permet également d'imbriquer les blocs "virtual" les uns dans les autres.

Lorsque "virtual" n'a pas d'argument, l'adresse de base de cette zone est la même que l'adresse actuelle dans la zone extérieure. Un argument de "virtual" peut avoir la forme d'un mot-clé "at" suivi d'une expression définissant l'adresse de base de la zone fermée :

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

Au lieu ou en plus de cet 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 de 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, c'est une adresse dans la zone courante. Dans ce cas, les octets chargés doivent déjà avoir 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" nécessite un type spécial d'étiquette, qui est créé avec "::" au lieu de ":". Une telle étiquette a une valeur qui ne peut pas être utilisée directement, mais elle peut l'être avec l'instruction "load" pour accéder aux données de la zone dans laquelle cette étiquette a été définie. L'adresse pour "load" doit alors être spécifiée comme l'étiquette de zone suivie de ":", puis l'adresse dans 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 qui sont générées ultérieurement, même dans la zone actuelle :

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

L'instruction "store" peut 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 d'un nombre d'octets à écrire (lorsque ce paramètre n'est pas présent, la longueur de la chaîne est déterminée par la taille associée à address), puis le mot-clé "at" et l'adresse des données à remplacer, dans l'un des deux mêmes modes que celui autorisé par "load". Cependant, le "store" n'est pas autorisé à modifier les données qui n'ont pas encore été générées, et toute zone qui a été touchée par "store" devient une zone variable, interdisant également au "load" de lire une donnée à partir de cette zone à l'avance.

L'exemple suivant utilise la combinaison de "load" et de "store" pour crypter l'intégralité du contenu de la zone actuelle avec 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 qui a été modifiée par "store" doivent être lues plus tôt dans la source, cela peut être obtenu en copiant ces données dans une zone différente qui ne serait pas contrainte de cette manière. Cela revient à définir une constante avec une valeur finale d'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 en avant par "load", mais elle ne peut jamais être référencée en avant par "store", même si elle fait référence à la zone de sortie actuelle.

L'instruction "virtual" peut avoir une étiquette de zone existante comme seul argument. Cette variante permet d'étendre un bloc préalablement défini et fermé avec des données supplémentaires. L'étiquette de zone doit faire référence à un bloc qui a été créé plus tôt dans la source avec "virtual". Toute définition de données dans un bloc d'extension 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 forme un terme variable d'un polynôme linéaire. Les métadonnées de ce terme sont l'adresse de base de la zone. Les métadonnées d'une étiquette de zone elle-même, accessibles avec l'opérateur "sizeof", sont égales à la longueur actuelle des données dans la zone.

Il existe une variante supplémentaire des directives "load" et "store" qui permet de lire et de modifier des données déjà générées dans le fichier de sortie en fonction d'un simple décalage dans cette sortie. Cette variante est reconnue lorsque le mot-clé "at" ou "from" est suivi du caractère ":" puis de la valeur d'un décalage.

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

L'instruction "restartout" abandonne toute la sortie générée jusqu'à ce point et recommence avec une sortie vide. Un argument facultatif peut spécifier l'adresse de base de la zone de sortie nouvellement démarrée. Lorsque "restartout" n'a aucun argument, l'adresse actuelle est conservée en l'utilisant comme base pour la nouvelle zone.

Les instructions "org", "section" et "restartout" ne peuvent pas être utilisées dans un bloc "virtual". Elles ne peuvent séparer que les zones qui vont dans le fichier de sortie.

14. Contrôle de la source et de la sortie

L'instruction "include" lit le texte source à partir d'un autre fichier et le traite avant de continuer dans la source actuelle. Son argument doit être une chaîne définissant le chemin d'accès vers un fichier (le format du chemin d'accès peut dépendre du système d'exploitation). Si après l'instruction et avant l'argument le "!" est ajouté, l'autre fichier est lu et traité de manière inconditionnelle, même s'il se trouve à l'intérieur d'un bloc ignoré (les instructions inconditionnelles de l'autre fichier peuvent alors être reconnues).

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

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

L'instruction "eval" prend une séquence d'octets définie par ses arguments, la traite comme un texte source et l'assemble. Les arguments sont des chaînes ou des valeurs numériques d'octets simples, séparés par des virgules. Dans l'exemple suivant, "eval" est utilisée pour générer des définitions de symboles nommés comme 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, séparés 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 dans le processus d'assemblage, avec un message personnalisé spécifié par son argument. Elle autorise le même type d'arguments que la directive "display".

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

La directive "format" permet de configurer des options supplémentaires concernant la sortie principale. Actuellement, le seul choix disponible est "format binary" suivi du mot clé "as" et d'une chaîne définissant une extension pour le fichier de sortie. À moins qu'un nom du fichier de sortie ne soit spécifié à partir de la ligne de commande, il est construit à partir du chemin d'accès au fichier source principal en supprimant l'extension et en attachant une nouvelle extension si celle-ci est définie.

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

La directive "format", de manière analogue à "end", utilise un identifiant qui la suit pour rechercher une instruction dans l'espace de noms enfant du symbole insensible à la casse nommé "format". La seule instruction intégrée qui réside dans cet espace de noms est le binaire ("binary"), mais des instructions supplémentaires peuvent être définies sous forme de macro-instructions.

Le symbole intégré "__time__" (avec le synonyme hérité "%t") a la valeur constante de l'horodatage marquant le moment où l'assemblage a été démarré.

Le "__file__" est un symbole intégré dont la valeur est une chaîne contenant le nom du fichier source actuellement traité. Le symbole "__line__" qui l'accompagne indique le nombre de lignes actuellement traitées dans ce fichier. Lorsque ces symboles sont accessibles dans une macro-instruction, ils conservent la même valeur qu'ils avaient pour la ligne appelante. S'il y a plusieurs niveaux de macro-instructions qui s'appellent, ces symboles ont partout la même valeur, correspondant à la ligne qui a appelé la macro-instruction la plus externe.

Le "__source__" est un autre symbole intégré, la valeur étant une chaîne contenant le nom du fichier source principal.

La directive "retaincomments" fait basculer l'assembleur pour qu'il traite un point-virgule comme un jeton régulier et ne supprime donc pas les commentaires des lignes avant le traitement. Cela permet d'utiliser des points-virgules dans des endroits comme le motif 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 ultérieurement les lignes lues à partir du texte source lorsque le saut de ligne est précédé d'une barre oblique inverse.

La directive "removecomments" ramène le comportement par défaut des points-virgules et la directive "combinelines" permet de combiner les lignes du texte source comme d'habitude.

15. Instructions CALM

La directive "calminstruction" permet de définir de nouvelles instructions sous forme de séquences compilées de commandes spécialisées. Contrairement aux macro-instructions ordinaires, qui fonctionnent sur un principe simple de substitution de texte, les instructions CALM (Compiled Assembly-Like Macro) sont capables d'effectuer de nombreuses opérations sans passer de texte dans le cycle de prétraitement et d'assemblage standard. Cela permet un contrôle plus fin, une meilleure gestion des erreurs et une exécution plus rapide.

Toutes les références aux symboles dans le texte définissant une instruction CALM sont fixées au moment de la définition. En conséquence, tous les symboles locaux de l'instruction CALM sont partagés entre toutes ses instances exécutées (par exemple, des instances consécutives peuvent voir les valeurs des symboles locaux laissés par les précédents). Pour faciliter la réutilisation de ces références, les commandes de CALM fonctionnent généralement sur des variables, réécrivant régulièrement les symboles avec de nouvelles valeurs.

Une instruction "calminstruction" suit les mêmes règles que la déclaration "macro", y compris des options comme le modificateur "!" pour définir une instruction inconditionnelle, "*" pour marquer un argument requis, ":" pour lui donner une valeur par défaut et "&" pour indiquer que l'argument final consomme tout le texte restant en ligne.

Cependant, comme l'instruction CALM fonctionne en dehors du cycle de prétraitement et d'assemblage standard, ses arguments ne deviennent pas des paramètres prétraités. Ce sont, en fait, des variables symboliques locales, qui reçoivent de nouvelles valeurs chaque fois que l'instruction est appelée.

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 est l'argument qui va recevoir le texte de l'étiquette.

Dans la définition de l'instruction CALM, seules les déclarations de son langage spécialisé sont identifiées. Le symbole initial de chaque ligne doit être un nom simple sans modificateurs et il n'est reconnu comme instruction valide que si un symbole insensible à la casse avec un tel nom se trouve dans l'espace de noms des commandes CALM (qui, à des fins de personnalisation, est accessible comme l'espace de noms ancré au symbole "calminstruction" insensible à la casse). Lorsqu'aucune instruction nommée 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 pour les commandes de saut CALM (décrites plus loin).

Une instruction "end calminstruction" doit être utilisée pour fermer la définition et rétablir le mode d'assemblage normal. Ce n'est pas une commande "end" régulière, mais une instruction de nom identique dans l'espace de noms CALM, qui n'accepte que "calminstruction" comme argument.

Le terme "assemble" est une commande qui prend un seul argument, qui doit être un identifiant d'une variable symbolique. Le texte de cette variable est passé directement à l'assembleur, sans aucun prétraitement (si le texte provient d'un argument de l'instruction, il a déjà subi un prétraitement lorsque cette ligne a été préparée).

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

please display 'Hi!'
```

La commande "match" est similaire à bien des égards 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 mis en correspondance avec le 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 demeurent non-affectées.

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

please(display 'Hi!')
```

La réussite de la correspondance peut également être testée avec un saut conditionnel "jyes" ou "jno" après la commande "match". Un saut "jyes" n'est effectué que lorsque la correspondance a réussi.

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

please do not display 'Bye!'
```

Pour contrôler davantage le flux de traitement, la commande "jump" permet d'effectuer un saut inconditionnel, et avec "exit" il est possible de terminer le traitement de l'instruction CALM à tout moment (cette commande ne prend aucun argument).

Alors que les symboles utilisés pour les arguments de l'instruction sont implicitement locaux, d'autres identifiants peuvent devenir des références fixes à des symboles globaux s'ils sont considérés comme accessibles au moment de la définition (car dans l'instruction CALM, toutes ces références sont traitées comme des utilisations et non comme des définitions). Une commande comme "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 imposer le traitement d'un symbole comme local, une commande "locale" doit être utilisée, suivie d'un ou plusieurs noms séparés par des virgules :

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

Un symbole rendu local se voit attribuer initialement une valeur définie mais inutilisable.

Si un modèle dans l'instruction CALM a un caractère "?" immédiatement après le nom d'un caractère générique, cela n'affecte pas la façon dont le symbole est identifié (si le symbole utilisé est insensible à la casse dépend de ce qui est présent dans la portée locale au moment où l'instruction est définie). Au lieu de cela, le fait de modifier le nom d'un caractère générique avec "?" lui permet d'être mis en correspondance avec un texte vide.

Puisque le texte source pour "match" est dans cette variante donnée par un seul identifiant, cette syntaxe permet d'avoir plus d'arguments. Un troisième argument facultatif pour correspondre ("match") peut contenir une paire de parenthèses. Tout élément générique doit alors être mis en correspondance avec un texte dont ce type de parenthèses est correctement équilibré.

```
calminstruction please? cmd&
	local first, second
	match first + second, cmd, ()
	jyes split
	assemble cmd
	exit
    split:
```

```
        assemble first
        assemble second
end calminstruction

please display 'H',('g'+2) + display '!'
```

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'est pas nécessaire de le faire précéder d'une virgule avec "=" pour qu'il soit inclus 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, à défaut 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
```

C'est le seul cas où une valeur non symbolique est convertie en symboles pouvant être insérés dans 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 pour "compute" définit une cible où le résultat doit être stocké, tandis que le second peut être n'importe quelle expression numérique, qui devient précompilée au moment de la définition. Lorsque l'expression est évaluée et que l'un des symboles auxquels elle fait référence se révèle avoir 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.

Un "compute" peut donc être utilisé non seulement pour évaluer une expression prédéfinie, mais également pour analyser et calculer une expression à partir d'un texte d'une variable symbolique (comme celle provenant d'un argument de l'instruction), ou d'une combinaison des deux :

```
a = 0

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

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

Étant donné que la variable symbolique est évaluée comme une sous-expression, son utilisation ici n'a aucun effet secondaire qui serait causé par 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 établit en conséquence le flag de résultat qui 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 au-

cun opérateur spécifique à l'expression logique).

```
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 sont pas explicitement documentées comme susceptibles d'agir sur le flag testé par "jyes" et "jno", conservent la valeur de ce flag inchangée.

La commande "publish" permet d'attribuer une valeur à un symbole identifié par le texte contenu dans une variable. Cela permet de définir un symbole avec un nom construit avec une commande comme "arrange", ou un nom qui a été passé dans un 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 affecter (symbolique ou numérique). Le premier argument peut être suivi du caractère ":" pour indiquer que le symbole doit être rendu constant, ou il peut être précédé de ":" pour que la valeur soit empilée au-dessus de la précédente (afin que la précédente puisse être retournée 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 n'est pas possible avec les directives standard de l'assembleur.

Le but de la commande "transform" est de remplacer les identifiants de variables symboliques (ou constantes) par leurs valeurs dans un texte donné, ce qui est la même opération que celle effectuée par la directive "equ" lorsqu'elle prépare la valeur à affecter. L'argument de "transform" doit être une variable symbolique dont la valeur va être traitée de cette façon et ensuite remplacée par le texte transformé.

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

Une 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             ; avertissement: peut s'accrocher aux références cycliques
      assemble cmd
end calminstruction
```

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

Facultativement, "transform" peut avoir deux arguments, le second spécifiant un espace de noms. Les identifiants dans le texte donné par le premier argument sont ensuite interprétés comme des symboles dans cet espace de noms quel que soit leur contexte d'origine.

Le "stringify" est une commande qui convertit le texte d'une variable en une chaîne et l'écrit dans la même variable (spécifiée par le seul argument). Cette opération est similaire à celle effectuée par l'opérateur "`" en prétraitement.

```
calminstruction (var) strcalc? val
      compute val, val        ; calcule l'expression
      arrange val, val        ; convertit le résultat en jeton décimal
      stringify val           ; convertit le jeton décimal en chaîne
      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, "take" est une commande qui permet de travailler avec des piles de valeurs. Elle supprime la valeur la plus élevée du symbole source (spécifiée par le deuxième argument) et la donne au symbole de destination (le premier argument), en la plaçant au-dessus de toutes les valeurs existantes. L'argument de destination peut être vide, auquel cas, la valeur est complètement 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 le même que la source, le flag de résultat peut être utilisé pour vérifier s'il existe une valeur disponible sans l'affecter, pour autant.

```
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 auquel on a accédé en tant que destination ou source par une commande "take" ne peut jamais être référencé en avant, même s'il pouvait l'être autrement.

La définition de macro-instructions dans l'espace de noms de "calminstruction" insensible à la casse permet d'ajouter des commandes personnalisées au langage des instructions CALM. Cependant, ils doivent être définis comme insensibles à la casse pour être reconnus comme tels.

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

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

writeln 'Next!'
```

Ces commandes supplémentaires peuvent même être définies comme des instructions CALM en tant que telles :

```
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 :)
```

La commande "initsym" dans cet exemple est utilisée pour attribuer du texte à la variable symbolique locale au moment où l'instruction "show" est définie. De la même manière que "local" (et contrairement à "stringify" et "assemble"), elle ne produit aucun code réel qui serait exécuté lorsque l'instruction "show" est appelée. Les arguments de "initsym" conservent leur contexte d'origine, donc les symboles dans le texte affecté à la variable "command" sont interprétés comme dans l'espace de noms local de l'instruction "show". Cela permet à la commande "display" d'accéder au "text" même s'il est local à l'instruction CALM et donc normalement visible uniquement dans le cadre de la définition de "show". Ceci est similaire à l'utilisation de "define" pour former des liens symboliques.

La commande "call" permet d'exécuter directement une autre instruction CALM. Son premier argument doit fournir un identifiant d'un symbole de classe d'instruction, et au moment de l'exécution ce symbole doit être défini comme CALM (il n'est pas possible d'appeler une macro-instruction ou une instruction intégrée de cette façon). L'exécution passe ensuite directement au point d'entrée de cette instruction et ne revient qu'une fois l'instruction appelée terminée.

```
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 il n'est pas censé contenir de définitions d'instruction.

Les arguments supplémentaires à appeler ("call") doivent être des identifiants de variables (ou constantes) dont les valeurs vont être transmises comme arguments à l'instruction appelée. Les valeurs de ces symboles sont affectées directement aux variables d'argument, sans aucune validation supplémentaire. Cela permet de transmettre à l'instruction CALM certaines valeurs qui autrement seraient impossibles à transmettre directement, comme les valeurs numériques (car lorsque les instructions sont appelées normalement, les arguments sont traités comme du texte et attribués comme valeurs symboliques). Un argument peut être omis lorsque 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
```

16. Commandes d'assemblage dans les instructions CALM

Un ensemble supplémentaire de commandes pour les instructions CALM permet de les utiliser au-delà du simple prétraitement, mais également pour générer et traiter directement la sortie. Ces commandes effectuent des opérations élémentaires, principalement sur une seule unité de données, mais en même temps, elles peuvent effectuer de nombreux calculs sur place, car leurs arguments, à quelques exceptions près, sont des expressions précompilées, similaires au deuxième argument pour calculer ("compute").

La commande "display" présente une chaîne d'octets sous forme de message dans la sortie standard, tout comme la directive normale du même nom. Elle prend en considération un seul argument, une expression donnant soit une chaîne, soit une valeur numérique d'un seul octet.

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.

La commande "emit" génère des données de longueur spécifiée par le premier argument et de valeur spécifiée par le second. Les deux arguments sont traités comme des expressions précompilées. Le deuxième argument est facultatif. S'il est omis, les données de longueur spécifiée sont générées comme non initialisées. Lorsque le deuxième argument est une chaîne, il doit tenir dans la taille spécifiée (un opérateur "lengthof" peut être utile dans ce cas).

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, ils ont une syntaxe différente, tous deux acceptant toujours trois arguments séparés par des virgules. Contrairement à leurs cousins, ils n'opèrent pas sur des adresses associées à des zones de sortie, mais sur des offsets bruts. Pour pointer vers le premier octet d'une zone, le décalage indiqué doit être nul.

Les arguments à charger ("load") sont, dans l'ordre : variable cible, décalage à partir duquel charger, nombre d'octets à charger. Le premier argument doit être un identifiant d'un symbole, les deux derniers sont des expressions précompilées. Le deuxième argument peut contenir une étiquette de la zone, suivie de ":" puis d'un décalage, ou simplement d'une simple expression numérique, auquel cas il s'agit d'un décalage dans l'ensemble de la sortie générée jusqu'à présent. La valeur chargée est toujours une chaîne de la longueur spécifiée.

Les arguments à stocker ("store") sont, dans l'ordre : le décalage à stocker, le nombre d'octets à stocker, la valeur à stocker (numérique ou chaîne). Les deux derniers arguments sont analogues aux arguments à émettre ("emit"), en même temps les deux premiers arguments sont comme les deux derniers arguments à charger ("load"). Le décalage peut être précédé de l'étiquette d'une zone avec ":" comme séparateur.

Pour effectuer une conversion entre les adresses utilisées par les classiques "load" et "store" et les offsets bruts attendus par les commandes CALM, il suffit d'ajouter ou de soustraire l'adresse de base de la zone. Si l'adresse de base n'est pas connue, elle peut être obtenue à l'aide de l'opérateur "l metadataof" appliqué à une étiquette de zone.
