fasm

2026

flat assembler g
Introduction et vue d'ensemble

par Tomasz Grysztar

Dernière mise à jour du document original : **08 Juin 2025**

Traduction française

By Calendos
Forum https://flatassembler.net/
Prélevé le 21 juillet 2026
Trad. v 2.3 (22/07/2026 à 15:24)

flat assembler g
Introduction et vue d'ensemble

Table des matières

1. Qu'est-ce que l'assembleur flat G ?

Il s'agit d'un moteur d'assemblage conçu pour succéder à celui utilisé dans l'assembleur *flat* 1, un assembleur reconnu pour les processeurs x86. Ce moteur, de base, est incapable de reconnaître et d'encoder les instructions d'un processeur, mais il peut être transformé en assembleur pour toute architecture de processeur. Son langage de macro-instructions est nettement amélioré par rapport à celui de l'assembleur *flat* 1 et permet d'implémenter facilement des encodeurs d'instructions sous forme de macro-instructions personnalisables.

Le code source de cet outil peut être compilé avec l'assembleur *flat* 1, mais il est également possible d'utiliser l'assembleur *flat G*. Le code source contient des clauses incluant différents fichiers d'en-tête selon l'assembleur utilisé. Lors de sa compilation, l'assembleur *flat G* utilise l'ensemble d'en-têtes fourni, qui implémente les instructions et formats x86 avec une syntaxe largement compatible avec l'assembleur *flat* 1.

Les exemples de programmes pour l'architecture x86 inclus dans ce pack sont des extraits initialement fournis avec l'assembleur *flat* 1. Ils utilisent des ensembles d'en-têtes implémentant les encodeurs d'instructions et les formateurs de sortie nécessaires à leur assemblage, tout comme le faisait l'assembleur *flat* d'origine.

Pour illustrer l'implémentation des jeux d'instructions de différentes architectures, des exemples de programmes sont fournis pour les microcontrôleurs 8051 et AVR. Leur simplicité les rend inadaptés à la programmation de tels processeurs, mais ils constituent une base solide pour la création d'environnements de ce type.

Un exemple d'assemblage du code JVM est également fourni. Il s'agit d'une conversion de l'exemple initialement créé pour l'assembleur *flat* 1. De ce fait, cet exemple est quelque peu rudimentaire et n'exploite pas pleinement les capacités du nouveau moteur. Il permet néanmoins de bien visualiser la structure d'un fichier de classe.

2. Comment cela marche-t-il ?

La fonction essentielle de l'assembleur *flat G* est de générer une sortie définie par les instructions du code source. Pour une ligne de texte comme celle présentée ci-dessous, l'assembleur générerait un seul octet avec la valeur indiquée :

```
db 90h
```

Les macro-instructions peuvent être définies pour générer des séquences de données spécifiques en fonction des paramètres fournis. Elles peuvent correspondre aux instructions du langage machine choisi, comme dans l'exemple suivant, mais elles peuvent également être définies pour générer d'autres types de données, à des fins diverses.

```
macro int number
          if number = 3
                    db 0CCh
          else
                    db 0CDh, number
          end if
end macro

int 20h    ; génère 2 octets
```

L'assembleur, tel qu'on le conçoit, peut être considéré comme un langage interprété, et l'assembleur présente effectivement de nombreuses caractéristiques d'un interpréteur. Cependant, il partage également certains aspects avec un compilateur. Il est possible qu'une instruction utilise une valeur définie plus loin dans le code source et qu'elle dépende des instructions qui la précèdent, comme le montre l'exemple suivant :

```
macro jmpi target
          if target-($+2) < 80h & target-($+2) >= -80h
                    db 0EBh
                    db target-($+1)
          else
                    db 0E9h
                    dw target-($+2)
          end if
end macro

          jmpi start
          db 'some data'
start:
```

L'instruction "`jmpi`", définie précédemment, génère le code d'une instruction de saut, conformément à l'architecture 8086. Ce code contient le décalage relatif de la cible du saut, stocké sur un octet ou un mot de 16 bits. Le décalage relatif est calculé comme la différence entre l'adresse de la cible et l'adresse de l'instruction suivante. Le

symbole spécial “$” indique l'adresse de l'instruction courante et sert à calculer le décalage relatif et à déterminer s'il peut tenir sur un seul octet.

Ainsi, le code généré par “`jmpi start`” dans l'exemple ci-dessus dépend de la valeur de l'adresse “`start`”, qui dépend elle-même de la longueur de la sortie de toutes les instructions précédentes, y compris le saut en question. Ceci crée une boucle de dépendances, et l'assembleur doit trouver une solution respectant toutes les contraintes imposées par le code source. Cela serait impossible si l'assembleur était un simple interpréteur impératif. Son langage est donc, à certains égards, déclaratif.

Trouver une solution à de telles dépendances circulaires peut ressembler à la résolution d'une équation, et il est même possible de construire un exemple où l'assembleur *flat G* est effectivement capable d'en résoudre une :

```
x = (x-1)*(x+2)/2-2*(x+1)
db x
```

La référence circulaire a été réduite ici à une simple définition qui se référence elle-même pour construire la valeur. L'assembleur *flat G* parvient à trouver une solution dans ce cas, bien qu'il puisse échouer dans de nombreux autres situations. La méthode employée par cet assembleur consiste à effectuer plusieurs passages sur le texte source, puis à tenter de prédire toutes les valeurs à partir des connaissances ainsi acquises. Cette approche est généralement suffisante pour l'assemblage de code machine, mais rarement pour résoudre des équations complexes ; l'exemple ci-dessus constitue une exception.

3. Moyens d'analyse des arguments d'une instruction

Toutes les instructions ne possèdent pas une syntaxe aussi simple que celle des exemples précédents. Pour faciliter le traitement des arguments pouvant contenir des constructions particulières, l'assembleur *flat G* fournit plusieurs outils performants, illustrés ci-dessous par des exemples implémentant quelques instructions du processeur Z80. Les règles d'utilisation de ces fonctionnalités sont décrites dans le manuel.

Lorsqu'une instruction accepte un nombre très restreint d'arguments, chacun d'eux peut être traité individuellement grâce à la construction “`match`” :

```
macro EX? first,second
	match (=SP?), first
		match =HL?, second
			db 0E3h
		else match =IX?, second
			db 0DDh,0E3h
		else match =IY?, second
			db 0FDh,0E3h
		else
			err "incorrect second argument"
		end match
	else match =AF?, first
		match =AF'?, second
			db 08h
		else
			err "incorrect second argument"
		end match
	else match =DE?, first
		match =HL?, second
			db 0EBh
		else
			err "incorrect second argument"
		end match
	else
		err "incorrect first argument"
	end match
end macro

EX (SP),HL
EX (SP),IX
EX AF,AF'
EX DE,HL
```

Le caractère "?" apparaît à de nombreux endroits pour indiquer que les noms ne sont pas sensibles à la casse ; toutes ces occurrences pourraient être supprimées afin de simplifier davantage l’exemple.

Lorsque l'ensemble des valeurs possibles d'un argument est plus vaste mais présente certaines régularités, des substitutions textuelles peuvent être définies pour remplacer certains symboles par des constructions soigneusement choisies, qui peuvent ensuite être reconnues et analysées.

```
A? equ [:111b:]
B? equ [:000b:]
C? equ [:001b:]
D? equ [:010b:]
E? equ [:011b:]
H? equ [:100b:]
L? equ [:101b:]

macro INC? argument
      match [:r:], argument
            db 100b + r shl 3
      else match (=HL?), argument
            db 34h
      else match (=IX?+d), argument
            db 0DDh,34h,d
      else match (=IY?+d), argument
            db 0FDh,34h,d
      else
            err "incorrect argument"
      end match
end macro

INC A
INC B
INC (HL)
INC (IX+2)
```

Cette approche présente un inconvénient : elle permet d'utiliser directement une expression comme "`[:0:]`" dans un argument. Il est toutefois possible d'empêcher une telle exploitation de la syntaxe en utilisant un préfixe dans la construction "`match`" :

```
REG.A? equ [:111b:]
REG.B? equ [:000b:]
REG.C? equ [:001b:]
REG.D? equ [:010b:]
REG.E? equ [:011b:]
REG.H? equ [:100b:]
REG.L? equ [:101b:]

macro INC? argument
      match [:r:], REG.argument
            db 100b + r shl 3
      else match (=HL?), argument
            db 34h
      else match (=IX?+d), argument
            db 0DDh,34h,d
      else match (=IY?+d), argument
            db 0FDh,34h,d
      else
            err "incorrect argument"
      end match
end macro
```

Dans le cas d'un argument structuré comme "`(IX+d)`", il peut être souhaitable d'autoriser d'autres formes algébriquement équivalentes de l'expression, telles que "`(d+IX)`" ou "`(c+IX+d)`". Plutôt que d'analyser individuellement chaque variante possible, il est possible de laisser l'assembleur évaluer l'expression en traitant le symbole sélectionné d'une manière particulière. Lorsqu'un symbole est déclaré comme un "élément", il n'a aucune valeur ; lorsqu'il est utilisé dans une expression, il est traité algébriquement comme un terme variable dans un polynôme.

```
element HL?
element IX?
```

```
element IY?

macro INC? argument
      match [:r:], argument
            db 100b + r shl 3
      else match (a), argument
            if a eq HL
                  db 34h
            else if a relativeto IX
                  db 0DDh,34h,a-IX
            else if a relativeto IY
                  db 0FDh,34h,a-IY
            else
                  err "incorrect argument"
            end if
      else
            err "incorrect argument"
      end match
end macro

INC (3*8+IX+1)

virtual at IX
      x db ?
      y db ?
end virtual

INC (y)
```

Il y a un petit problème avec la macro-instruction ci-dessus. Un paramètre peut contenir n'importe quel texte et, lorsqu'une telle valeur est insérée dans une expression, cela peut induire un comportement erratique. Par exemple, si "`INC (1|0)`" était traité, l'expression "`a eq HL`" deviendrait "`1|0 eq HL`", et cette expression logique est correcte et vraie même si l'argument est mal formé. Cet effet secondaire indésirable est une conséquence du fonctionnement des macro-instructions selon un principe simple de substitution de texte (et la meilleure façon d'éviter ce genre de problèmes est d'utiliser CALM). Pour éviter cela, on peut utiliser une variable locale comme proxy pour stocker la valeur d'un argument.

```
macro INC? argument
      match [:r:], argument
            db 100b + r shl 3
      else match (a), argument
            local value
            value = a
            if value eq HL
                  db 34h
            else if value relativeto IX
                  db 0DDh,34h,a-IX
            else if value relativeto IY
                  db 0FDh,34h,a-IY
            else
                  err "incorrect argument"
            end if
      else
            err "incorrect argument"
      end match
end macro
```

L'utilisation d'une variable proxy présente un avantage supplémentaire : sa valeur est calculée avant que la macro-instruction ne génère une quelconque sortie. Lorsqu'une expression contient un symbole comme “$”, sa valeur peut varier selon l'endroit où il est calculé. L'utilisation d'une variable proxy garantit que la valeur utilisée est bien celle obtenue en évaluant l'argument avant la génération du code de l'instruction.

Lorsque l'ensemble des symboles autorisés dans les expressions est vaste, il est préférable d'utiliser une construction unique pour traiter toute une famille de symboles. Une déclaration “`element`” permet d'associer une valeur supplémentaire à un symbole. Cette information peut ensuite être récupérée grâce à l'opérateur “`metadata`” appli-

qué à un polynôme linéaire contenant ce symbole comme variable. L'exemple suivant est une variante de la macro-instruction précédente illustrant l'utilisation de cette fonctionnalité :

```
element register
element A? : register + 111b
element B? : register + 000b
element C? : register + 001b
element D? : register + 010b
element E? : register + 011b
element H? : register + 100b
element L? : register + 101b

element HL?
element IX?
element IY?

macro INC? argument
      local value
      match (a), argument
            value = a
            if value eq HL
                  db 34h
            else if value relativeto IX
                  db 0DDh, 34h, a-IX
            else if value relativeto IY
                  db 0FDh, 34h, a-IY
            else
                  err "incorrect argument"
            end if
      else match any more, argument
            err "incorrect argument"
      else
            value = argument
            if value eq value element 1 & value metadata 1 relativeto register
                  db 100b + (value metadata 1 - register) shl 3
            else
                  err "incorrect argument"
            end if
      end match
end macro
```

La spécification “any more” permet de détecter tout argument contenant une expression complexe composée de plusieurs éléments. Ceci empêche l'utilisation de syntaxes telles que “INC A+0” ou “INC A+B-A”. Cependant, pour certains jeux d'instructions, l'inclusion de cette contrainte peut relever d'une préférence personnelle.

La condition “value eq value element 1” garantit que la valeur ne contient aucun terme autre que le nom d'un registre. Même lorsqu'un argument est contraint de ne contenir qu'un seul élément, il est possible qu'il ait une valeur complexe, par exemple avec des définitions telles que “X = A + B” ou “Y = 2 * A”. Dans ce cas, “INC X” et “INC Y” entraîneraient tous deux le renvoi de la valeur “A” par l'opérateur “element 1”, ce qui diffère de la valeur vérifiée.

Si une instruction prend un nombre variable d'arguments, une méthode simple pour identifier ses différentes formes consiste à déclarer un argument avec le modificateur “&” afin de transmettre l'intégralité des arguments à “match”.

```
element CC

NZ? := CC + 000b
Z?  := CC + 001b
NC? := CC + 010b
C?  := CC + 011b
PO  := CC + 100b
PE  := CC + 101b
P   := CC + 110b
M   := CC + 111b
```

```
macro CALL? arguments&
      local cc,nn
      match condition =, target, arguments
            cc = condition - CC
            nn = target
            db 0C4h + cc shl 3
      else
            nn = arguments
            db 0CDh
      end match
      dw nn
end macro

CALL 0
CALL NC,2135h
```

Cette approche permet également de gérer des cas plus complexes, comme lorsque les arguments contiennent des virgules ou sont délimités de différentes manières.

Les macros CALM offrent un contrôle plus précis du processus d'analyse des arguments, et leur variante “match” propose davantage d'options. Voici comment la macro ci-dessus pourrait être réécrite :

```
calminstruction CALL? target&
      local   condition
      match   condition:name =, target, target, :
      jno     unconditional
      check   defined condition & condition relativeto CC
      jno     error
      emit    1, 0C4h + (condition - CC) shl 3
      jump    address
   error:
     err      "unrecognized syntax"
   unconditional:
     emit     1, 0CDh
   address:
     emit     2, target
end calminstruction
```

Cette fois, aucun proxy n'est nécessaire, car le texte des arguments n'est évalué que lors du calcul des expressions qui les contiennent. De plus, la fonction “match” peut être utilisée pour vérifier la présence d'expressions mal formées, tout comme elle vérifie la validité du nom du symbole de condition dans l'exemple.

4. Traitement des étiquettes

Une méthode standard pour définir une étiquette consiste à la faire suivre du symbole 2 points (ceci fait office de saut de ligne et toute autre commande, y compris une autre étiquette, peut suivre sur la même ligne). Une telle étiquette définit simplement un symbole dont la valeur est égale à l'adresse courante, initialement nulle et incrémentée à chaque ajout d'octets à la sortie.

Dans certaines variantes du langage assembleur, il peut être souhaitable d'autoriser une étiquette à précéder une instruction sans ":" supplémentaire. Il est alors nécessaire de créer une macro-instruction étiquetée qui, après avoir défini une étiquette, transmet le traitement à la macro-instruction d'origine portant le même nom.

```
struc INC? argument
      .:
      INC argument
end struc

start   INC A
      INC B
```

Il faut procéder ainsi pour chaque instruction nécessitant ce type de syntaxe. Une simple boucle comme la suivante suffirait :

```
iterate instruction, EX,INC,CALL
      struc instruction? argument
            .: instruction argument
```

```
        end struc
end iterate
```

Chaque instruction intégrée définissant des données possède déjà sa variante étiquetée.

En définissant une instruction étiquetée avec “?” à la place du nom, il est possible d'intercepter toute ligne commençant par un identifiant qui n'est pas une instruction connue et qui est donc considéré comme une étiquette. L'exemple suivant autoriserait une étiquette sans “:” à commencer n'importe quelle ligne du texte source (il gère également les cas particuliers : les étiquettes suivies de “:” ou de “=”” et d'une valeur fonctionnent toujours) :

```
struc ? tail&
        match :, tail
                .:
        else match : instruction, tail
                .: instruction
        else match == value, tail
                . = value
        else
                .: tail
        end match
end struc
```

De toute évidence, il n'est plus nécessaire de définir des macro-instructions étiquetées spécifiques lorsqu'un effet global de ce type est appliqué. Une variante doit être choisie en fonction du type de syntaxe à autoriser.

L'interception des étiquettes définies avec ":" peut s'avérer utile lorsque la valeur de l'adresse courante nécessite un traitement supplémentaire avant d'être affectée à une étiquette, par exemple lorsqu'un processeur utilise des adresses dont l'unité est supérieure à un octet. La macro-instruction d'interception pourrait alors ressembler à ceci :

```
struc ? tail&
        match :, tail
                label . at $ shr 1
        else match : instruction, tail
                label . at $ shr 1
                instruction
        else
                . tail
        end match
end struc
```

La valeur de l'adresse actuelle utilisée pour définir les étiquettes peut être modifiée avec “org”. Si les étiquettes doivent être différenciées de valeurs absolues, un symbole défini avec “element” peut être utilisé pour former une adresse :

```
element CODEBASE
org CODEBASE + 0

macro CALL? argument
        local value
        value = argument
        if value relativeto CODEBASE
                db 0CDh
                dw value - CODEBASE
        else
                err "incorrect argument"
        end if
end macro
```

Pour définir des étiquettes dans un espace d'adressage qui ne sera pas affiché en sortie, il convient de déclarer un bloc “virtuel”. L'exemple suivant prépare les macro-instructions “DATA” et “CODE” pour alterner entre la génération d'instructions de programme et d'étiquettes de données. Seuls les codes d'instruction seront affichés.

```
element DATA
DATA_OFFSET = 2000h
element CODE
CODE_OFFSET = 1000h

macro DATA?
```

```
      _END
      virtual at DATA + DATA_OFFSET
end macro

macro CODE?
      _END
      org CODE + CODE_OFFSET
end macro

macro _END?
      if $ relativeto DATA
            DATA_OFFSET = $ - DATA
            end virtual
      else if $ relativeto CODE
            CODE_OFFSET = $ - CODE
      end if
end macro

postpone
      _END
end postpone

CODE
```

Le bloc “postpone” est utilisé ici pour garantir la fermeture correcte du bloc “virtual”, même si le texte source se termine par des définitions de données.

Dans l'environnement préparé par l'exemple ci-dessus, toute instruction peut distinguer les étiquettes de données de celles définies dans le programme. Par exemple, une instruction de branchement peut accepter comme argument une étiquette du programme ou une valeur absolue, mais refuser toute étiquette de données.

```
macro CALL? argument
      local value
      value = argument
      if value relativeto CODE
            db 0CDh
            dw value - CODE
      else if value relativeto 0
            db 0CDh
            dw value
      else
            err "incorrect argument"
      end if
end macro

DATA

variable db ?

CODE

routine:
```

Dans ce contexte, soit "CALL routine" soit "CALL 1000h" seraient autorisés, alors que "CALL variable" ne le serait pas.

Lorsque les étiquettes ont des valeurs qui ne sont pas des nombres absolus, il est possible de générer des relocalisations pour les instructions qui les utilisent. Un bloc "virtuel" spécial peut être utilisé pour stocker les décalages des valeurs à l'intérieur du programme qui doivent être déplacées lorsque sa base change :

```
virtual at 0
      Relocations::
      rw RELOCATION_COUNT
end virtual

RELOCATION_INDEX = 0
```

```
postpone
      RELOCATION_COUNT := RELOCATION_INDEX
end postpone

macro WORD? value
      if value relativeto CODE
            store $ - CODE : 2 at Relocations : RELOCATION_INDEX shl 1
            RELOCATION_INDEX = RELOCATION_INDEX + 1
            dw value - CODE
      else
            dw value
      end if
end macro

macro CALL? argument
      local value
      value = argument
      if value relativeto CODE | value relativeto 0
            db 0CDh
            word value
      else
            err "incorrect argument"
      end if
end macro
```

Le tableau des relocalisations ainsi créé est ensuite accessible via la commande “load”. Les deux lignes suivantes permettent d'afficher l'intégralité du tableau dans le résultat :

```
load RELOCATIONS : RELOCATION_COUNT shl 1 from Relocations : 0
dw RELOCATIONS
```

L'instruction “load” lit l'intégralité du tableau et le stocke dans une seule chaîne de caractères, puis l'instruction “dw” l'écrit en sortie (avec un remplissage jusqu'à atteindre une longueur multiple d'un mot, mais dans ce cas précis, la chaîne ne nécessite jamais ce remplissage).

Pour des relocalisations plus complexes, des modificateurs supplémentaires peuvent être nécessaires. Par exemple, si les parties supérieure et inférieure d'une adresse doivent être stockées séparément (probablement sur deux instructions) et relocalisées séparément, les modificateurs nécessaires peuvent être implémentés comme suit :

```
element MOD.HIGH
element MOD.LOW

HIGH? equ MOD.HIGH +
LOW? equ MOD.LOW +

macro BYTE? value
      if value relativeto MOD.HIGH + CODE
            ; register HIGH relocation
            db (value - MOD.HIGH - CODE) shr 8
      else if value relativeto MOD.LOW + CODE
            ; register LOW relocation
            db (value - MOD.LOW - CODE) and 0FFh
      else if value relativeto MOD.HIGH
            db (value - MOD.HIGH) shr 8
      else if value relativeto MOD.LOW
            db (value - MOD.LOW) and 0FFh
      else
            db value
      end if
end macro
```

Les commandes permettant d'enregistrer la relocalisation ont été omises par souci de clarté. Dans ce cas, il ne s'agit pas seulement d'un décalage dans le code, mais aussi d'informations supplémentaires à enregistrer dans les

structures appropriées. Grâce à cette préparation, des unités relocalisables dans le code pourraient être générées comme suit :

```
BYTE HIGH address
BYTE LOW address
```

Cette approche permet d'activer facilement la syntaxe avec des modificateurs dans toute instruction qui utilise en interne la macro-instruction “byte” lors de la génération de code.

5. Génération de plusieurs sections de fichier en parallèle

Ce moteur d'assemblage possède une seule sortie principale qui doit être générée séquentiellement. Cela peut s'avérer problématique lorsque le fichier doit contenir des sections distinctes pour le code et les données, issues de fragments entrelacés et potentiellement répartis sur plusieurs fichiers sources. Il existe cependant plusieurs méthodes pour y remédier, toutes reposant, d'une manière ou d'une autre, sur les capacités de référence anticipée de l'assembleur.

Une approche naturelle consiste à définir le contenu de la section auxiliaire dans un bloc “virtual” et à le copier à l'emplacement approprié dans la sortie en une seule opération. Une fois étiqueté, un bloc “virtual” peut être rouvert plusieurs fois pour y ajouter des données.

```
        include '8086.inc'
        org     100h
        jmp     CodeSection

DataSection:

        virtual
                Data::
        end virtual

        postpone
                virtual Data
                        load Data.OctetString : $ - $$ from $$
                end virtual
        end postpone

        db Data.OctetString

CodeSection:

        virtual Data
                Hello db "Hello!",24h
        end virtual

        mov     ah,9
        mov     dx,Hello
        int     21h

        virtual Data
                ExitCode db 37h
        end virtual

        mov     ah,4Ch
        mov     al,[ExitCode]
        int     21h
```

Cela conduit à une syntaxe relativement simple, même sans macros supplémentaires.

Une autre méthode consiste à placer les éléments de la section dans des macros et à les exécuter à l'emplacement requis dans le code source. L'inconvénient de cette approche est que la détection des erreurs dans les définitions peut s'avérer complexe.

Les techniques permettant d'ajouter facilement des éléments à une section générée en parallèle peuvent également être très utiles pour générer des structures de données telles que des tables de relocalisation. Au lieu des commandes “store” utilisées précédemment lors de la démonstration du concept, des directives de données

classiques peuvent être utilisées à l'intérieur d'un bloc “virtual” rouvert pour créer des enregistrements de relocalisation.

6. Options pour l’analyse d'autres types de syntaxe

Dans certains cas, une commande que l'assembleur doit analyser peut commencer par un élément différent d'un nom d'instruction ou d'une étiquette. Il peut s'agir d'un nom précédé d'un caractère spécial, comme “.” ou “!”, ou d'une construction d'un type totalement différent. Il est alors nécessaire d'utiliser la macro “?” pour intercepter des lignes entières de texte source et traiter toute syntaxe spéciale de ce type.

Par exemple, s'il était nécessaire d'autoriser une commande écrite sous la forme “.CODE”, il serait impossible de l'implémenter directement comme une macro-instruction, car le point initial fait que le symbole est interprété comme un symbole local, et une instruction définie globalement ne pourrait jamais être exécutée de cette manière. La macro-instruction d'interception apporte une solution :

```
macro ? line&
      match .=CODE?, line
            CODE
      else match .=DATA?, line
            DATA
      else
            line
      end match
end macro
```

Les lignes contenant “.CODE” ou “.DATA” sont traitées ici de manière à invoquer la macro-instruction globale correspondante, tandis que toutes les autres lignes interceptées sont exécutées sans modification. Cette méthode permet de filtrer toute syntaxe particulière et de laisser l'assembleur traiter les instructions classiques.

Il arrive que la syntaxe non conventionnelle soit attendue uniquement dans une zone spécifique du texte source, par exemple à l'intérieur d'un bloc aux limites définies. La macro-instruction d'analyse syntaxique doit alors être appliquée uniquement à cet endroit, puis supprimée par “purge” à la fin du bloc.

```
macro concise
      macro ? line&
            match =end =concise, line
                  purge ?
            else match dest+==src, line
                  ADD dest,src
            else match dest-==src, line
                  SUB dest,src
            else match dest==src, line
                  LD dest,src
            else match dest++, line
                  INC dest
            else match dest--, line
                  DEC dest
            else match any, line
                  err "syntax error"
            end match
      end macro
end macro

concise
      C=0
      B++
      A+=2
end concise
```

Une macro-instruction définie de cette manière n'intercepte pas les lignes contenant des directives contrôlant le flux d'exécution, telles que “if” ou “repeat”, et ces directives restent utilisables librement à l'intérieur d'un tel bloc. Le comportement serait différent si la déclaration était de la forme “macro ?! line&”. Dans ce cas, chaque ligne serait interceptée sans exception.

Une autre option pour intercepter les commandes spéciales consiste à utiliser “struc ?” afin d'intercepter uniquement les lignes ne commençant pas par une instruction connue (le symbole initial étant alors considéré

comme une étiquette). Comme cette méthode ne teste que les commandes inconnues, elle devrait engendrer une surcharge moindre sur l'assembleur.

```
struc (head) ? tail&
      match .=CODE?, head
            CODE tail
      else
            head tail
      end match
end struc
```

Toutes ces approches dissimulent un piège subtil. Une étiquette définie avec : peut être suivie d'une autre instruction sur la même ligne. Si cette instruction (ici masquée dans le paramètre "tail") est une directive de contrôle comme "if", la placer dans la clause "else" entraînera une imbrication incorrecte des blocs de contrôle. Une solution possible consiste à appeler le contenu de "tail" en dehors du bloc "match". On pourrait par exemple utiliser une macro spéciale :

```
struc (head) ? tail&
      local invoker
      match .=CODE?, head
            macro invoker
                  CODE tail
            end macro
      else
            macro invoker
                  head tail
            end macro
      end match
      invoker
end struc
```

Une option plus simple consiste à appeler directement la ligne d'origine et, en cas de nécessité de la remplacer, à la faire ignorer à l'aide d'un autre intercepteur de ligne (qui se libère immédiatement après) :

```
struc (head) ? tail&
      match .=CODE?, head
            CODE tail
            macro ? line&
                  purge ?
            end macro
      end match
      head tail
end struc
```

Cependant, une bien meilleure façon d'éviter ce genre d'écueils consiste à utiliser les instructions CALM plutôt que les macros standard. Il est ainsi possible de traiter les arguments et d'assembler la ligne originale ou modifiée sans recourir à aucune directive de contrôle. Les instructions CALM offrent également des performances nettement supérieures, ce qui peut s'avérer particulièrement important dans le cas d'intercepteurs appelés quasiment à chaque ligne du texte source.

7. Contrôle du contexte de reconnaissance des identifiants de symboles

Lors de la conception d'une macro à usage général, il est important de s'assurer de son bon fonctionnement quel que soit l'endroit où elle est appelée. Bien que la directive "local" permette de créer des symboles privés pour chaque instance de la macro sans interférer avec un environnement inconnu, il arrive qu'une variable doive être partagée entre plusieurs appels de la macro. Un symbole défini globalement convient généralement à cet effet, mais il peut être sujet à des interférences. Prenons l'exemple suivant :

```
GLOBAL_STATE = 0

macro state
      db GLOBAL_STATE
end macro

macro switch value
      GLOBAL_STATE = GLOBAL_STATE xor (value)
end macro
```

Cette solution présente quelques problèmes. Premièrement, si le symbole nommé “GLOBAL_STATE” était également utilisé ailleurs à d'autres fins, cela interférerait avec ces macros. Choisir un nom rare et spécifique pour une variable globale peut atténuer le problème (et s'il est nécessaire d'utiliser plusieurs symboles globaux, il est préférable de créer un espace de noms dédié, de préférence avec un nom inhabituel), mais il serait préférable de séparer ces symboles d'une manière ou d'une autre.

Un autre problème se manifesterait si les macros ci-dessus étaient utilisées à l'intérieur d'un bloc “namespace”, car la définition de “GLOBAL_STATE” créerait alors un symbole dans l'autre espace de noms. Il existe une solution simple à ce second problème : il suffit d'ajouter un point à la fin de l'identificateur pour garantir que le symbole défini est bien celui dont la valeur est accédée à droite :

```
GLOBAL_STATE. = GLOBAL_STATE xor (value)
```

Il existe cependant une autre option. Les arguments fournis à la macro conservent le contexte de tous les identificateurs présents dans leur texte (sauf indication contraire, lorsque le nom d'un argument est précédé de “&”). L'exemple suivant illustre cet effet :

```
macro tester arg
      namespace X
            a = 0
            db a
            db arg
      end namespace
end macro

a = 3
tester a
```

Les deux définitions de données assemblées contiennent le même texte : “db a”, mais la seconde interprète “a” dans le contexte de l’appel de la macro et génère l’octet 3 au lieu de 0. Ceci est rendu possible grâce à un mécanisme comparable à une coloration syntaxique. La valeur du paramètre “arg” possède une propriété supplémentaire, telle qu’une couleur de texte, qui différencie la première occurrence de “db a” de la seconde. Ainsi, si l’on définit une macro à l’intérieur d’une autre, le texte de cette dernière conservera sa coloration. L’exemple suivant utilise ce mécanisme pour améliorer la définition des macros “state” / “switch” de l’exemple précédent :

```
macro setup variable

      variable = 0

      macro state
            db variable
      end macro

      macro switch value
            variable = variable xor (value)
      end macro

end macro

setup GLOBAL_STATE

namespace Program

      switch 8

      GLOBAL_STATE = 0

      state

end namespace
```

Lorsque le fichier “setup GLOBAL_STATE” est assemblé, la macro “switch” est définie avec un corps contenant le texte :

```
GLOBAL_STATE = GLOBAL_STATE xor (value)
```

où les deux instances de "GLOBAL_STATE" sont associées au contexte global. D’autres parties du texte ne sont associées à aucune information supplémentaire de ce type. Cependant, lorsque le “switch 8” est assemblé, la ligne générée par la macro est :

```
        GLOBAL_STATE = GLOBAL_STATE xor (8)
```

et, cette fois, un autre texte coloré apparaît : la valeur “8” indique le contexte d’appel de la macro “switch”.

La variante de “switch” et “state” ci-dessus utilise toujours la variable globale, quel que soit l’endroit où elle est appelée. Ceci est similaire au concept de fermeture présent dans de nombreux langages de programmation.

La variable reste néanmoins globale et d’autres portions de code peuvent toujours y accéder. Pour la rendre totalement inaccessible, on peut utiliser la directive “local”, qui crée un paramètre spécial remplaçant son nom par le même texte, mais coloré spécifiquement pour indiquer le contexte propre à l’instance de la macro.

```
macro setup

      local variable

      variable = 0

      macro state
            db variable
      end macro

      macro switch value
            variable = variable xor (value)
      end macro

end macro

setup
```

Ce type de transfert de contexte se produit également lors de la définition d'une variable symbolique. Lorsqu'un symbole est défini avec “equ” ou “define”, sa valeur entière est marquée par le contexte présent au moment de sa définition. Lorsque le texte de cette valeur est affecté à un paramètre avec “match” ou “irpv”, les informations contextuelles véhiculées par les fragments de texte sont préservées.

```
First:
      .x = 1

define LIST .x

Second:
      .x = 2

LIST equ LIST, .x

match values, LIST
      display `values
      db values
end match
```

La directive “display” indique que le texte extrait de la variable “LIST” avec la fonction “match” est “.x, .x”. Or, les textes de ces deux identifiants ont des contextes différents, et il s'avère que la liste contient deux valeurs distinctes.

Lorsqu'une fonction “match” découpe le texte d'un identifiant en plusieurs parties, chacune conserve les informations contextuelles du texte original. Ainsi, si un identifiant est reconstitué à partir de fragments de texte provenant de différentes sources, seul le contexte associé à la partie initiale influe sur la reconnaissance du symbole. Cela permet de forcer la reconnaissance de n'importe quel symbole dans son contexte actuel en le faisant précéder d'un dièse (#).

```
macro tester name
      namespace my
            name db ?       ; symbole défini dans son espace de nom d’origine
            #name db ?      ; symbole défini dans l’espace de nom "my"
      end namespace
end macro
```

Si cette astuce ne suffit pas et qu'il est nécessaire de dépouiller un texte de toute information contextuelle, la directive “rawmatch” permet précisément cela. Un effet similaire peut être obtenu en convertissant la valeur d'un paramètre en chaîne de caractères à l'aide de l'opérateur puis en interprétant ce texte brut avec “eval”.

Un argument de macro préfixé par “&” n'ajoute aucune information contextuelle à sa valeur, mais ne supprime pas non plus les informations déjà présentes dans le texte ou certaines de ses parties. Cela permet de transmettre du texte en toute sécurité sans perte d'information, mais aussi sans le contaminer avec un contexte indésirable, comme le montre cet exemple en plusieurs étapes :

```
calminstruction (var) transparent_equ &val&
	publish :var, val
end calminstruction

namespace Windows
	EOL := string 0x0A0D
	link equ EOL
end namespace

EOL := 0x0A

match WinEOL, Windows.link
	list	equ		WinEOL, EOL	; contexte ajouté
	list	transparent_equ	WinEOL, EOL	; pas de contexte ajouté
end match

namespace C64
	EOL := 0x0D
	irpv items, list
		db items
	end irpv
end namespace
```

Tous les éléments transmis à “db” contiennent le texte “EOL”, mais interprétés différemment : dans le contexte de l'espace de noms “Windows”, d'un espace de noms global, et enfin comme un texte brut sans contexte (auquel cas il finit par pointer vers la définition la plus proche, celle de l'espace de noms “C64”).

8. Définition d’une instruction partageant un nom avec l'une des directives principales

Il peut arriver qu'un langage soit généralement facile à implémenter avec des macros, mais qu'il doive inclure une commande portant le même nom qu'une directive de l'assembleur. Bien qu'il soit possible de redéfinir n'importe quelle instruction par une macro, les macros elles-mêmes peuvent nécessiter un accès à la directive d'origine. Pour permettre à une instruction portant le même nom d'appeler une instruction différente selon le contexte, le langage implémenté peut être interprété dans un espace de noms contenant la macro de redéfinition, tandis que toutes les macros nécessitant un accès à la directive d'origine devraient temporairement basculer vers un autre espace de noms où celle-ci n'a pas été redéfinie. Cela impliquerait que chaque macro de ce type encapsule son contenu dans un bloc “namespace”.

Il existe cependant une autre astuce, liée à la manière dont les textes des paramètres de macro ou des variables symboliques préservent le contexte d'interprétation des symboles qu'ils contiennent (y compris l'espace de noms de base et l'étiquette parente pour les symboles commençant par un point).

Contrairement aux deux cas mentionnés, le texte d'une macro ne contient généralement pas d'informations supplémentaires. Toutefois, si une macro est construite de manière à inclure du texte ayant appartenu à un paramètre d'une autre macro ou à une variable symbolique, ce texte conserve les informations de contexte même lorsqu'il est intégré à une nouvelle macro. Par exemple :

```
macro definitions end?
	namespace embedded
	struc LABEL? size
		match , size
			.:
		else
			label . : size
		end match
	end struc
	macro E#ND? name
		end namespace
		match any, name
			ENTRYPOINT := name
```

```
            end match
            macro ?! line&
            end macro
      end macro
end macro

definitions end

start LABEL
END start
```

Le paramètre passé à la macro “definitions” peut sembler inopérant, puisqu'il remplace chaque occurrence de “end” par le même mot. Cependant, le texte issu de ce paramètre est enrichi d'informations contextuelles supplémentaires, et cet attribut est conservé lorsqu'il est intégré à une nouvelle macro. Grâce à cela, la macro “LABEL” peut être utilisée dans un espace de noms où l'instruction “end” a une signification différente, tandis que ses occurrences font toujours référence au symbole de l'espace de noms externe.

Dans cet exemple, le paramètre est insensible à la casse ; il remplacerait donc même le “END” de l'instruction “macro” censée définir un symbole dans l'espace de noms “embedded”. C'est pourquoi l'identificateur a été scindé à l'aide d'un opérateur de concaténation afin d'éviter qu'il ne soit reconnu comme un paramètre. Cela ne serait pas nécessaire si le paramètre était sensible à la casse (comme c'est généralement le cas).

On peut obtenir le même résultat en utilisant des variables symboliques à la place des paramètres de macro, grâce à la fonction “match” qui permet d'extraire le texte d'une variable symbolique.

```
define link end
match end, link
      namespace embedded
      struc LABEL? size
            match , size
                  .:
            else
                  label . : size
            end match
      end struc
      macro END? name
            end namespace
            match any, name
                  ENTRYPOINT := name
            end match
            macro ?! line&
            end macro
      end macro
end match

start LABEL
END start
```

Cela ne fonctionnerait pas sans passer par une variable symbolique, car les paramètres définis par des directives de contrôle comme “match” n'ajoutent pas d'informations contextuelles au texte, sauf si elles y sont déjà présentes.

Les instructions CALM offrent une autre approche à ce type de problèmes. Si un jeu d'instructions personnalisé est entièrement défini sous forme d'instructions CALM, il peut même se passer d'accès aux directives de contrôle d'origine. Toutefois, si une instruction CALM doit assembler une directive potentiellement inaccessible, la variable symbolique passée à “assemble” doit être définie avec le contexte approprié pour le symbole d'instruction.

9. Conversion d’une macro-instruction en CALM

Une macro-instruction classique se compose de lignes de texte prétraitées (en remplaçant les noms des paramètres par leurs valeurs correspondantes) à chaque appel de l'instruction, puis transmises à l'assembleur. Par exemple, la macro-instruction qui suit génère une seule ligne à assembler, en remplaçant “number” par le texte fourni par son unique argument :

```
macro octet value*
      db value
end macro
```

Une instruction CALM peut être vue comme un préprocesseur personnalisé, écrit dans un langage spécifique. Elle utilise diverses commandes pour traiter les arguments et générer les lignes à assembler. De manière générale, elle peut simuler le fonctionnement d'un préprocesseur standard grâce à la commande “arrange”. Après le prétraitement de la ligne, elle doit également la transmettre explicitement à l'assembleur avec la commande “assemble”.

```
calminstruction octet value*
      arrange value, =db value
      assemble value
end calminstruction
```

Cela donne le même résultat que la macro-instruction d'origine, car le prétraitement est identique. Cependant, contrairement au texte de la macro-instruction, un motif fourni à “arrange” doit indiquer explicitement quels jetons de nom doivent être remplacés par leurs valeurs et lesquels (précédés de “=”) doivent rester inchangés. Les jetons copiés du motif sont dépouillés de toute information contextuelle, tout comme le texte de la macro-instruction n'en contient généralement aucune (tandis que les valeurs issues des arguments conservent le contexte de reconnaissance dans lequel l'instruction a été lancée).

Il s'agit de la méthode de conversion la plus simple ; une simple séquence de commandes “arrange” et “assemble” permet de générer les mêmes lignes que la macro-instruction d'origine. Il existe toutefois une exception : lorsqu'une commande “locale” est exécutée par une macro-instruction, elle crée un paramètre prétraité avec une valeur spéciale pointant vers un symbole de l'espace de noms propre à l'instance donnée de l'instruction.

```
macro pointer
      local next
      dd next
    next:
end macro
```

Dans le cas de CALM, aucun espace de noms n'est disponible ; l'espace de noms local d'une instruction CALM est partagé entre toutes ses instances. Par conséquent, si un nouveau symbole unique est nécessaire à chaque appel de l'instruction, il doit être construit manuellement. Une méthode simple consiste à ajouter un nombre unique au nom.

```
global_uid = 0

calminstruction pointer
      compute global_uid, global_uid + 1
      local command
      arrange command, =dd =next#global_uid
      assemble command
      arrange command, =next#global_uid:
      assemble command
end calminstruction
```

Ici, la fonction “arrange” reçoit une variable numérique et doit la remplacer par du texte. Cela fonctionne uniquement si la valeur est un nombre entier positif ou nul ; dans ce cas, “arrange” la convertit en un jeton texte contenant la représentation décimale de ce nombre. Les lignes transmises à l'assembleur contiendront donc des identificateurs tels que “next#1”.

Alors que l'incrémentation du compteur global pourrait être réalisée en préparant une commande assembleur standard comme “global_uid = global_uid + 1” avec “arrange” et en la transmettant à l'assembleur, la commande “compute” permet de le faire directement dans le processeur CALM. De plus, elle n'est alors pas affectée par quoi que ce soit qui modifie le contexte de l'assembleur. Si l'instruction était définie comme inconditionnelle et utilisée dans un bloc IF ignoré, “compute” exécuterait tout de même sa tâche, car l'exécution des commandes CALM est, tout comme le prétraitement standard, indépendante du flux principal de l'assembleur. De plus, les références à “global_uid” pointent toujours vers le même symbole : celui qui était dans la portée lors de la définition et de la compilation de l’instruction CALM. Par conséquent, l’incrémentation de sa valeur avec “compute” est plus fiable et prévisible.

De manière similaire, l'ensemble des lignes définissant le label peut être remplacé par une commande "publish". Ici, la valeur de l'étiquette (qui doit être égale à l'adresse après l'assemblage de la ligne contenant "dd") doit être calculée en premier, car "publish" effectue uniquement l'affectation d'une valeur au symbole :

```
global_uid = 0

calminstruction pointer
      compute global_uid, global_uid + 1
      local symbol, command
```

```
      arrange symbol, =next#global_uid
      arrange command, =dd symbol
      assemble command
      local address
      compute address, $
      publish symbol:, address
end calminstruction
```

L'instruction CALM étant conditionnelle, la fonction “publish” qu'elle contient l'est également. Elle fonctionne donc correctement en remplacement de l'assemblage d'une ligne avec une étiquette.

Bien qu'un compteur global présente plusieurs avantages, il peut être modifié ; l'utilisation d'un compteur local peut donc parfois s'avérer préférable. Cependant, l'espace de noms local de l'instruction CALM n'est généralement pas accessible de l'extérieur, ce qui complique l'initialisation de ce compteur. Une solution consiste à vérifier si le compteur a déjà été initialisé à l'aide de la commande “take”.

```
calminstruction pointer
      local id
      take id, id
      jyes increment
      compute id, 0
    increment:
      compute id, id + 1
      local symbol, command
      arrange symbol, =next#id
      arrange command, =dd symbol
      assemble command
      local address
      compute address, $
      publish symbol:, address
end calminstruction
```

Mais cela ajoute des commandes qui sont exécutées à chaque appel de l'instruction. Une meilleure solution consiste à exploiter la possibilité de définir des instructions personnalisées traitées lors de la définition de l'instruction CALM :

```
calminstruction calminstruction?.init? var*, val:0
      compute val, val
      publish var, val
end calminstruction

calminstruction pointer
      local id
      init id, 0
      compute id, id + 1
      local symbol, command
      arrange symbol, =next#id
      arrange command, =dd symbol
      assemble command
      local address
      compute address, $
      publish symbol:, address
end calminstruction
```

L'instruction personnalisée “init” est appelée lors de la définition de l'instruction CALM (elle ne génère aucune commande à exécuter par l'instruction définie ; elle devrait elle-même utiliser des commandes “assemble” pour générer des instructions à compiler). Elle reçoit le nom d'une variable de la portée locale de l'instruction CALM et utilise “publish” pour lui assigner une valeur numérique initiale.

Pour initialiser une variable locale avec une valeur symbolique, une instruction personnalisée encore plus simple suffirait :

```
calminstruction calminstruction?.initsym? var*, val&
      publish var, val
end calminstruction
```

Le texte de l'argument “val” contient le contexte de reconnaissance de la définition de l'instruction CALM qui contient l'instruction “initsym”, permettant ainsi de préparer un texte pour “assembler” contenant des références à des symboles locaux :

```
calminstruction bigendian32? value
	local command
	initsym command, dd value
	compute value, value bswap 4
	assemble command
end calminstruction
```

Une fois compilée, cette instruction ne contient plus que deux commandes : “compute” et “assemble”. La valeur du symbole local “command” est un texte interprété dans le même contexte local et faisant référence au même symbole “value” que “compute”.

Cet exemple illustre un autre avantage de CALM par rapport aux macro-instructions standard : sa sémantique stricte empêche divers comportements indésirables permis par une simple substitution de texte. Le texte de “value” sera évalué par “compute” comme une sous-expression numérique, signalant une erreur en cas de syntaxe inattendue. Il est donc préférable de traiter les arguments entièrement via des commandes CALM et de n'utiliser “assemble” que pour les instructions simples finales. Ces dernières peuvent même être éliminées lorsqu'une commande CALM existe pour effectuer la même tâche.

```
calminstruction bigendian32? value
	emit 4, value bswap 4
end calminstruction
```

Cela effectue des opérations complètement indépendamment du processus d'assemblage standard, donc si l'instruction contenant était définie comme inconditionnelle, la sortie serait générée même si elle était invoquée à l'intérieur d'un bloc "if" ignoré ou lors de la définition d'une macro-instruction (et cela se produirait sans ajouter de ligne à la définition).
