fasm

flat assembler g
Introduction et vue d'ensemble

par Tomasz Grysztar

Dernière mise à jour du document original : 05 Août 2021

Traduction française

By Calendos
Forum https://flatassembler.net/
Prélevé le 9 mai 2024
Trad. v 2.2 (23/05/2024 à 15:24)

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, l'un des assembleurs reconnus pour les processeurs x86. Il s'agit d'un moteur nu qui, intrinsèquement, est dans l'impossibilité de reconnaître et de coder les instructions de n'importe quel processeur. En contrepartie, il a la capacité de devenir un assembleur pour n'importe quelle architecture de processeur. Il possède un langage de macro-instruction sensiblement amélioré par rapport à celui de l'assembleur *flat* 1 et permet de mettre en œuvre 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* lui-même pour le compiler. Le script contient des clauses qui incluent différents fichiers d'en-tête en fonction de l'assembleur utilisé. Lorsque l'assembleur *flat G* se compile, il utilise l'ensemble d'en-têtes fourni qui implémentent les instructions et les formats x86 avec une syntaxe le plus souvent compatible avec l'assembleur *flat* 1.

Les exemples de programmes pour l'architecture x86 fournis dans ce package sont les exemples sélectionnés fournis à l'origine avec l'assembleur *flat* 1. Ils utilisent des ensembles d'en-têtes qui implémentent les encodeurs d'instructions et les formateurs de sortie nécessaires pour les assembler, tout comme l'assembleur *flat* d'origine l'a fait.

Pour montrer comment les jeux d'instructions de différentes architectures peuvent être implémentés, il existe quelques exemples de programmes pour les microcontrôleurs, 8051 et AVR. Ils sont simples et ne fournissent donc pas un cadre complet pour la programmation de tels processeurs, bien qu'ils puissent fournir une base solide pour la création de tels environnements.

Il existe également un exemple d'assemblage du bytecode JVM, qui est une conversion de l'exemple créé à l'origine pour l'assembleur *flat* 1. Pour cette raison, il est quelque peu grossier et n'utilise pas pleinement les capacités offertes par le nouveau moteur. Cependant, il est bon pour 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 à partir des instructions du code source. Si l'on considère, par exemple, la ligne de texte ci-dessous, l'assembleur générera un seul octet avec la valeur indiquée :

```
db 90h
```

Des macro-instructions peuvent être constituées pour générer des séquences spécifiques de données en fonction des paramètres fournis. Elles peuvent correspondre aux instructions du langage machine choisi, comme dans l'exemple suivant. Mais elles pourraient tout aussi bien 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'assemblage vu de cette façon peut être considéré comme une sorte de langage interprété, et l'assembleur a certainement de nombreuses caractéristiques d'un interpréteur. Cependant, il partage également certains aspects d'un compilateur. Ainsi est-il possible pour une instruction d'utiliser la valeur qui est définie ultérieurement dans le script et peut dépendre des instructions qui précèdent cette définition, 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:
```

Le "jmpi" défini ci-dessus produit le code de l'instruction de saut comme dans l'architecture 8086. Un tel code contient le décalage relatif de la cible d'un saut, stocké dans un seul octet ou un mot de 16 bits. Le décalage relatif est calculé comme une différence entre l'adresse de la cible et celle de l'instruction qui suit immédiatement. Le symbole spécial "$" fournit l'adresse de l'instruction en cours et est utilisé pour calculer le décalage relatif et déterminer s'il peut tenir dans un seul octet.

Par conséquent, le code généré par "jmpi start" dans l'exemple ci-dessus dépend de la valeur d'une adresse étiquetée comme "start", et cela dépend à son tour de la longueur de la sortie de toutes les instructions qui le précèdent, y compris ledit saut. Cela crée une boucle de dépendances et l'assembleur doit trouver une solution qui satisfasse toutes les contraintes créées par le script source. Cela ne serait pas possible si l'assembleur n'était qu'un interpréteur impératif. Son langage est donc, à certains égards, déclaratif.

Trouver une solution pour de telles dépendances circulaires peut s'apparenter à la résolution d'une équation, et il est même possible de construire un exemple où l'assembleur *flat G* est en effet 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 définition unique qui se réfère à elle-même pour construire la valeur. L'assembleur *flat G* est capable de trouver une solution dans ce cas, bien que dans beaucoup d'autres, il puisse échouer. La méthode utilisée par cet assembleur consiste à effectuer plusieurs passes sur le texte source, puis à essayer de prédire toutes les valeurs avec les connaissances recueillies de cette façon. Cette approche est dans la plupart des cas assez bonne pour l'assemblage de codes machine, mais suffit rarement pour résoudre les équations complexes et l'échantillon ci-dessus est une exception parmi d'autres.

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

Toutes les instructions n'ont pas une syntaxe simple comme celle des exemples précédents. Pour faciliter le traitement des arguments qui peuvent contenir des constructions spéciales, l'assembleur *flat G* fournit quelques outils appropriés, illustrés ci-dessous sur les exemples qui implémentent quelques instructions sélectionnées du processeur Z80. Les règles régissant l'utilisation des fonctionnalités présentées se trouvent dans le manuel.

Lorsqu'une instruction a un très petit ensemble d'arguments autorisés, chacun d'eux peut être traité séparément avec 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 générique "?" apparaît en de nombreux endroits pour marquer les noms comme insensibles à la casse. Toutes ces occurrences peuvent être supprimées pour simplifier davantage l'exemple.

Lorsque l'ensemble des valeurs possibles d'un argument est plus grand mais présente quelques régularités, les substitutions textuelles peuvent être définies pour remplacer certains des 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 a un aspect qui n'est pas toujours souhaitable : elle permet d'utiliser une expression comme "[:0:]" directement dans un argument. Mais il est possible d'éviter d'exploiter la syntaxe de cette manière 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 parfois être souhaitable d'autoriser d'autres formes algébriquement équivalentes de l'expression, comme "(d + IX)" ou "(c + IX + d)". Au lieu d'analyser chaque variante possible individuellement, il est possible de laisser l'assembleur évaluer l'expression tout en traitant le symbole sélectionné de manière distincte. Lorsqu'un symbole est déclaré comme "element", il n'a aucune valeur et 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 placée dans une expression, elle peut induire un comportement erratique. Par exemple, si "INC (1|0)" était traité, cela transformerait l'expression "a eq HL" en "1|0 eq HL" et cette expression logique est correcte et vraie même si l'argument était mal formé. Un tel effet secondaire malheureux est une conséquence des macro-instructions fonctionnant sur un simple principe de substitution de texte (et la meilleure façon d'éviter de tels problèmes est d'utiliser CALM à la place). Ici, pour éviter que cela ne se produise, une variable locale peut être utilisée comme proxy contenant 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
```

Il y a un avantage supplémentaire à une telle variable proxy, grâce au fait que sa valeur est calculée avant que la macro-instruction ne commence à générer une sortie. Lorsqu'une expression contient un symbole comme "$", elle peut donner des valeurs différentes selon l'endroit où elle est calculée et l'utilisation de la variable proxy garantit que la valeur prise est celle obtenue en évaluant l'argument avant de générer le code d'une instruction.

Lorsque le jeu de symboles autorisés dans les expressions est plus grand, il est préférable d'avoir une seule construction pour traiter une famille entière d'entre eux. Une déclaration "element" peut associer une valeur supplémentaire à un symbole et cette information peut ensuite être récupérée avec l'opérateur "metadata" appliqué à un polynôme linéaire qui contient un symbole donné en tant que variable. L'exemple suivant est une autre variante de la macro-instruction précédente qui illustre 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" est là pour récupérer tout argument qui contient une expression complexe constituée de plus d'un jeton. Cela empêche l'utilisation d'une syntaxe comme "INC A + 0" ou "INC A + BA". Mais dans le cas de certains des jeux d'instructions, l'inclusion d'une telle contrainte peut dépendre 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 forcé de ne pas contenir plus d'un seul jeton, il est toujours possible qu'il ait une valeur complexe, par exemple s'il y avait des définitions comme "X = A + B" ou "Y = 2 * A". "INC X" et "INC Y" feraient alors tous deux renvoyer à l'opérateur "element 1" la valeur "A", qui diffère de la valeur testée dans les deux cas.

Si une instruction prend un nombre variable d'arguments, un moyen simple de reconnaître ses différentes formes est de déclarer un argument avec le modificateur "&" pour passer le contenu complet 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 d'autres cas plus difficiles, comme lorsque les arguments peuvent contenir des virgules ou sont délimités de différentes manières.

4. Traitement des étiquettes

Une manière standard de définir une étiquette est de suivre son nom avec ":" (cela agit également comme un saut de ligne et toute autre commande, y compris une autre étiquette, peut suivre dans la même ligne). Une telle étiquette définit simplement un symbole avec la valeur égale à l'adresse actuelle, qui est initialement égale à zéro et augmente lorsque des octets sont ajoutés dans la sortie.

Dans certaines variantes du langage d'assemblage, il peut être souhaitable de permettre à l'étiquette de précéder une instruction sans ":" supplémentaire entre les deux. Il est alors nécessaire de créer une macro-instruction étiquetée qui, après avoir défini une étiquette, passe le traitement à la macro-instruction d'origine avec le même nom :

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

start   INC A
      INC B
```

Cela doit être fait pour chaque instruction qui doit autoriser ce type de syntaxe. Une simple boucle comme la suivante pourrait suffire :

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

Chaque instruction intégrée qui définit des données a déjà la variante étiquetée.

En définissant une instruction étiquetée qui a "?" à la place du nom, il est possible d'intercepter chaque ligne commençant par un identifiant qui n'est pas une instruction connue et est donc supposé être une étiquette. La suivante permettrait à une étiquette sans ":" de commencer n'importe quelle ligne dans le texte source (elle gère également les cas spéciaux afin que les étiquettes suivies de ":" ou de "=" et 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 même des étiquettes définies par ":" peut devenir utile lorsque la valeur de l'adresse actuelle nécessite un traitement supplémentaire avant d'être affectée à une étiquette – par exemple lorsqu'un processeur utilise des adresses avec une unité plus grande qu'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 courante utilisée pour définir les étiquettes peut être modifiée avec "org". Si les étiquettes doivent être différenciées des 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 reflété dans la sortie, un bloc "virtual" doit être déclaré. L'exemple suivant prépare les macro-instructions "DATA" et "CODE" pour basculer entre la génération des instructions de programme et des étiquettes de données. Seuls les codes d'instructions iraient à la sortie :

```
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 que le bloc "virtual" se ferme toujours correctement, même si le texte source se termine par des définitions de données.

Dans l'environnement préparé par l'exemple ci-dessus, toute instruction serait capable de distinguer les étiquettes de données de celles définies dans le programme. Par exemple, une instruction de branchement pourrait être faite pour accepter un argument étant soit une étiquette dans un programme ou une valeur absolue, mais pour interdire 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, un "CALL routine" ou un "CALL 1000h" seraient autorisés, alors qu'un "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 relocations pour les instructions qui les utilisent. Un bloc "virtual" spécial peut être utilisé pour stocker les décalages de valeurs à l'intérieur du programme qui doivent être déplacés 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
```

La table des relocalisations ainsi créée est alors accessible avec "load". Les deux lignes suivantes peuvent être utilisées pour placer la table dans son intégralité quelque part dans la sortie :

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

Le "load" lit la table entière en une seule chaîne, puis "dw" l'écrit dans la sortie (complétée au multiple d'un mot, mais dans ce cas la chaîne ne nécessite jamais un tel remplissage).

Pour des types plus complexes de relocations, un modificateur supplémentaire peut être nécessaire. Par exemple, si les parties supérieure et inférieure d'une adresse devaient être stockées dans des endroits séparés (par exemple sur deux instructions) et déplacées séparément, les modificateurs nécessaires pourraient 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 qui enregistreraient la relocation ont été omises pour plus de clarté. Dans ce cas non seulement le décalage dans le code, mais certaines informations supplémentaires devraient être enregistrées dans des structures appropriées. Avec une telle préparation, des unités relogeables dans le code pourraient être générées comme suit :

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

Une telle approche permet d'activer facilement la syntaxe avec des modificateurs dans toute instruction qui utilise en interne une macro-instruction "byte" lors de la génération de code.

5. Génération de plusieurs sections de fichier en parallèle

Ce moteur d'assemblage a une seule sortie principale qui doit être générée séquentiellement. Cela peut sembler problématique lorsque le fichier doit contenir des sections distinctes pour le code et les données, mais le contenu de ces sections doit être collecté à partir d'éléments entrelacés qui peuvent être répartis sur plusieurs fichiers

source. Cependant, il existe des méthodes relativement simples qui permettent de le faire, toutes basées d'une manière ou d'une autre sur les capacités de référencement préalables de l'assembleur.

Une approche naturelle consiste à définir le contenu de la section auxiliaire dans un bloc "virtual" et à le copier à la position appropriée dans la sortie en une seule opération. Lorsqu'un bloc "virtual" est étiqueté, il peut être rouvert plusieurs fois pour y ajouter plus de 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 l'aide de macros supplémentaires.

Une autre méthode pourrait consister à placer les éléments de la section dans des macros et à les exécuter tous à la position requise dans la source. Un inconvénient d'une telle approche est que le traçage des erreurs dans les définitions peut devenir un peu compliqué.

Les techniques qui permettent d'ajouter facilement à une section générée en parallèle peuvent également être très utiles pour générer des structures de données comme des tables de relocalisation. Au lieu des commandes "store" utilisées précédemment lors de la démonstration du concept, des directives de données régulières pourraient être utilisées dans un bloc "virtual" ré-ouvert 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 quelque chose de différent d'un nom d'instruction ou d'une étiquette. Il se peut qu'un nom soit précédé d'un caractère spécial, comme "." ou "!", ou qu'il s'agisse d'un type de construction entièrement différent. Il faut alors utiliser "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 en tant que ".CODE", il ne serait pas possible de l'implémenter directement en tant que macro-instruction, car le point initial amène le symbole à être interprété comme un symbole local et l'instruction définie globalement ne pourrait jamais être exécutée de cette façon. La macro-instruction d'interception apporte, de ce point de vue, une solution :

```
        macro ? line&
              match .=CODE?, line
                    CODE
```

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

Les lignes qui contiennent du texte ".CODE" ou ".DATA" sont traitées ici de telle manière qu'elles invoquent la macro-instruction globale avec le nom correspondant, tandis que toutes les autres lignes interceptées sont exécutées sans modifications. Cette méthode permet de filtrer toute syntaxe spéciale et de laisser l'assembleur traiter les instructions régulières comme d'habitude.

Parfois, une syntaxe non conventionnelle n'est attendue que dans une zone spécifique du texte source, comme à l'intérieur d'un bloc avec des limites définies. La macro-instruction d'analyse doit alors être appliquée uniquement à cet endroit, et supprimée avec "purge" lorsque le bloc se termine:

```
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 qui contiennent des directives contrôlant le flux de l'assemblage, comme "if" ou "repeat", et elles peuvent toujours être utilisées librement à l'intérieur d'un tel bloc. Cela changerait si la déclaration était sous la forme "macro ?! line&". Une telle variante intercepterait chaque ligne sans exception.

Une autre option pour attraper des commandes spéciales pourrait être d'utiliser "struc ?" pour intercepter uniquement les lignes qui ne commencent pas par une instruction connue (le symbole initial est alors traité comme étiquette). Étant donné que celui-ci ne teste que des commandes inconnues, cela devrait entraîner moins de surcharge sur l'opération d'assemblage en tant que telle :

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

Toutes ces approches cachent un piège subtil. Une étiquette définie par ":" peut être suivie d'une autre instruction dans la même ligne. Si cette instruction suivante (qui, ici, devient cachée dans le paramètre "tail") est une directive de contrôle comme "if". La mettre à l'intérieur de la clause "else" va provoquer une rupture d'imbrication des blocs de contrôle. Une solution possible est d'invoquer d'une manière ou d'une autre le contenu "tail" en dehors du bloc "match". Une d'appeler façon pourrait être d'appeler 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 consisterait à appeler directement la ligne d'origine et, lorsque le remplacement est nécessaire, à l'ignorer à l'aide d'un autre intercepteur de ligne (tout en se débarrassant de lui-même 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 de piège est d'utiliser les instructions CALM en lieu et place des macros standard. Là, il est possible de traiter des arguments et d'assembler la ligne d'origine ou modifiée sans utiliser aucune directive de contrôle. Les instructions CALM offrent également de bien meilleures performances, ce qui peut être particulièrement important dans le cas d'intercepteurs appelés pour presque toutes les lignes du texte source.

7. Définition d'une instruction de même nom qu'une des directives principales

Il peut arriver qu'un langage puisse être en général facilement implémenté avec des macros tout en devant inclure une commande de même nom qu'une des directives de l'assembleur. Bien qu'il soit possible de remplacer une instruction par une macro, les macros elles-mêmes peuvent nécessiter un accès à la directive d'origine. Pour permettre au même nom d'appeler une instruction différente en fonction du contexte, le langage implémenté peut être interprété dans un espace de noms contenant une macro de substitution, tandis que toutes les macros nécessitant un accès à la directive d'origine devraient temporairement basculer vers un autre espace de noms où il n'a pas été surpassé. Cela nécessiterait que chaque macro de ce type rassemble son contenu dans un bloc `"namespace"`.

Mais il y a une autre astuce, liée à la façon dont les textes des paramètres de macro ou des variables symboliques préservent le contexte dans lequel les symboles qu'ils contiennent doivent être interprétés (cela inclut l'espace de noms de base et l'étiquette parent pour les symboles commençant par un point).

Contrairement aux deux occurrences mentionnées, le texte d'une macro ne contient normalement pas ces informations supplémentaires, mais si une macro est construite de telle manière qu'elle contient du texte qui était autrefois transporté dans un paramètre vers une autre macro ou dans une variable symbolique, alors ce texte conserve les informations sur le contexte même lorsqu'il fait partie d'une macro nouvellement définie. 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 donné à la macro "definitions" peut sembler ne rien faire, car il remplace chaque instance de "end" par exactement le même mot – mais le texte qui provient du paramètre est doté d'informations supplémentaires sur le contexte, et cet attribut est alors préservé lorsque le texte fait partie d'une nouvelle macro. Grâce à cela, la macro "LABEL" peut être utilisée dans un espace de noms où l'instruction "end" a pris une signification différente, mais les instances de "end" dans son corps se réfèrent toujours au symbole dans l'espace de noms externe.

Dans cet exemple, le paramètre a été rendu insensible à la casse, et ainsi il pourrait même remplacer l'instruction "END" dans "macro" qui est censée définir un symbole dans l'espace de noms "embedded". Pour cette raison, l'identifiant a été divisé avec un opérateur de concaténation pour éviter qu'il ne soit reconnu comme paramètre. Cela ne serait pas nécessaire si le paramètre était sensible à la casse (comme très souvent).

Le même effet peut être obtenu en utilisant des variables symboliques au lieu de paramètres macro, avec l'aide de "match" pour 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 le texte à travers une variable symbolique, car les paramètres définis par des directives de contrôle comme "match" n'ajoutent pas d'information de contexte au texte à moins qu'elle n'y soit déjà.

Les instructions CALM permettent une autre approche de ce type de problème. Si un jeu d'instructions personnalisé est défini entièrement sous la forme CALM, elles peuvent même ne pas avoir besoin d'un accès aux directives de contrôle d'origine. Cependant, si l'instruction CALM a besoin d'assembler une directive qui pourrait ne pas être accessible, la variable symbolique passée à "assemble" doit être définie avec le contexte approprié pour le symbole d'instruction.

Une macro-instruction classique se compose de lignes de texte qui sont prétraitées (en remplaçant les noms des paramètres par leurs valeurs correspondantes) chaque fois que l'instruction est appelée et ces lignes prétraitées sont passées à l'assemblage. Par exemple, la macro-instruction qui suit ne génère qu'une seule ligne à assembler, et elle le fait en remplaçant "number" par le texte donné par le seul argument de l'instruction :

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

Une instruction CALM peut être considérée comme un préprocesseur personnalisé qui doit être écrit dans un langage spécial. Il est capable d'utiliser diverses commandes pour traiter les arguments et générer des lignes à

assembler. Au niveau de base, il est également capable de simuler ce que fait le préprocesseur standard - avec l'aide de la commande "arrange". Après avoir prétraité la ligne, il doit également la passer explicitement à l'assembly avec une 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 elle effectue le même type de prétraitement. Cependant, contrairement au texte de la macro-instruction, un modèle donné à "arrange(r)" doit indiquer explicitement quels jetons de nom doivent être remplacés par leurs valeurs et lesquels (précédés de "=") doivent être laissés inchangés. Les jetons copiés à partir du modèle sont dépouillés de toute information de contexte, tout comme le texte de la macro-instruction n'en porte normalement pas (tandis que les valeurs provenant des arguments conservent le contexte de reconnaissance dans lequel l'instruction a été lancée).

C'est la méthode de conversion la plus simple et une simple séquence de commandes "arrange" et "assemble" pourrait être faite pour générer les mêmes lignes que par la macro-instruction d'origine. Mais il y a une exception : lorsqu'une commande "local" est exécutée par macro-instruction, elle crée un paramètre prétraité avec une valeur spéciale qui pointe vers un symbole dans l'espace de noms unique à une instance donnée de l'instruction.

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

Dans le cas de CALM, un tel espace de noms n'est pas 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 chaque fois que l'instruction est appelée, il doit être construit manuellement. Une méthode évidente pourrait être d'ajouter un numéro 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, "arrange" reçoit une variable qui a une valeur numérique et doit la remplacer par un texte. Cela ne fonctionne que lorsque la valeur est un nombre non négatif de plan. Dans ce cas "arrange" le convertit en un jeton de texte qui contient une représentation décimale de ce nombre. Les lignes passées à l'assembleur vont donc contenir des identifiants tels que "next #1".

Alors que l'incrémentation du compteur global pourrait se faire en préparant une commande d'assemblage standard comme "global_uid = global_uid + 1" avec "arrange" et en la passant à l'assembleur, la commande "compute" permet de le faire directement dans le processeur CALM. De plus, il n'est alors pas affecté par tout ce qui modifie le contexte d'assemblage. Si l'instruction était définie comme inconditionnelle et utilisée dans un bloc IF ignoré, le calcul ("compute") effectuerait toujours sa tâche, car l'exécution des commandes CALM est – tout comme le prétraitement standard – effectuée indépendamment du flux principal de l'assemblage. De plus, les références à "global_uid" pointent toujours vers le même symbole – celui qui était dans la portée lorsque l'instruction CALM a été définie et compilée. Par conséquent, l'incrémentation de la valeur avec "compute" est plus fiable et prévisible.

De la même manière, l'assemblage de la ligne définissant l'étiquette 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" n'effectue que l'attribution 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
```

Dans la mesure où l'instruction CALM en tant que telle est conditionnelle, la publication (`"publish"`) à l'intérieur est également conditionnelle. Par conséquent, elle fonctionne correctement en remplacement de l'assemblage de la ligne avec une étiquette.

Bien qu'un compteur global présente plusieurs avantages, il peut subir des interférences, de sorte que parfois l'utilisation d'un compteur local puisse s'avérer préférable. Cependant, l'espace de noms local de l'instruction CALM n'est normalement pas accessible de l'extérieur. Il est donc un peu plus difficile de donner une valeur initiale à un tel compteur. Une façon pourrait être de vérifier si le compteur a déjà été initialisé avec une valeur en utilisant 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 fois que l'instruction est appelée. Une meilleure solution utilise 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 au moment où l'instruction CALM est définie (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). Il reçoit le nom de variable de la portée locale de l'instruction CALM, et il utilise `"publish"` pour attribuer une valeur numérique initiale à cette variable.

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"` porte le contexte de reconnaissance de la définition de l'instruction CALM qui contient l'instruction `"initsym"`. Il permet donc de préparer un texte pour `"assemble"` contenant des références à des symboles locaux :

```
calminstruction be32? value
```

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

Encore une fois, au terme de la compilation de cette intruction, elle ne contient que deux commandes réelles, "compute" et "assemble", et la valeur du symbole local "command" est un texte qui est interprété dans le même contexte local et fait référence au même symbole "value" comme le fait "compute".

Cet exemple met en relief également un autre avantage de CALM par rapport aux macro-instructions standard : sa sémantique stricte empêche divers types de comportements indésirables autorisés par une simple substitution de texte. Le texte de "value" va être évalué par "compute" comme une sous-expression numérique, signalant une erreur sur toute syntaxe inattendue. Par conséquent, il devrait être avantageux de traiter les arguments entièrement via des commandes CALM et de n'utiliser "assemble" que pour les instructions simples finales.
