Procédures: Techniques avancées Chapitre 12

Le dernier chapitre a décrit comment créer des procédures, passer des paramètres et assigner et accèder à des variables locales. Ce chapitre reprend à l'endroit où le précédent s'est terminé et décrit comment accéder aux variables non-locales d'autres procédures, passer des procédures comme paramètres et mettre en application quelques structures de contrôle définies par l'utilisateur.

12.0 Vue d'ensemble du chapitre

Ce chapitre complète la discussion des procédures, des paramètres, et des variables locales commencées dans le chapitre précédent. Il décrit comment des langages structurés par blocs comme Pascal, Modula-2, Algol, et ADA accèdent à des variables locales et non-locales, ainsi que comment mettre en application une structure de contrôle définie par l'utilisateur, l'*iterator*. La majeure partie de ce chapitre est d'un grand intérêt pour les créateurs de compilateurs et ceux qui veulent apprendre comment les compilateurs produisent du code pour certains types de construction de programme. Peu de programmes en assembleur pur emploieront ces techniques. Par conséquent, aucun sujet traité ici ne sera particulièrement important pour ceux qui veulent seulement étudier l'assembleur. Cependant, si vous voulez écrire un compilateur, ou vous voulez apprendre comment les compilateurs produisent du code de manière à pouvoir écrire des programmes efficaces dans des HLL, il vous faudra aborder ce contenu tôt ou tard.

Ce chapitre commence par aborder la notion de *portée* et comment des HLLs comme Pascal accèdent aux variables dans des procédures imbriquées. La première section discute le concept de l'imbrication lexicale et de l'utilisation des liens statiques et des tables lexicales pour accéder à des variables non-locales. Ensuite, on discute de la manière de passer des variables comme paramètres à différents niveaux lexicaux. La troisième section montre comment passer des paramètres d'une procédure comme paramètres d'une autre procédure. Le quatrième sujet principal que ce chapitre couvre est le passage de procédures comme paramètres et on conclut avec une discussion des *iterateurs*, une structure de contrôle définie par l'utilisateur.

Ce contenu présume une connaissance d'un langage structuré par bloc comme le Pascal ou l'ADA. Si votre seule expérience de HLL est avec une langue structurée sans bloc comme C, C++, BASIC, ou Fortran, certains des concepts suivants risquent d'être complètement nouveaux et vous aurez des difficultés à les comprendre. N'importe quelle introduction à Pascal ou à Ada vous expliquera tout concept que vous ne comprendriez pas et que ce chapitre suppose être nécessaire.

12.1 Imbrication lexicale, liens statiques et tables lexicales

Dans des langages structurés par bloc comme le Pascal[11], il est possible *d'imbriquer* des procédures et des fonctions. Imbriquer une procédure dans une autre limite l'accès à la procédure imbriquée; vous ne pouvez pas accéder à la procédure imbriquée depuis l'extérieur de la procédure qui la contient. De même, les variables que vous déclarez dans une procédure sont visibles à l'intérieur de la procédure et à toutes les procédures imbriquées dans celle-ci[22]. C'est la notion standard de *portée* des langages structurés par bloc qui devrait être tout à fait familière à quiconque a écrit des programmes en Pascal ou ADA.

Dans les langages de haut niveau structués par blocs, il y a beaucoup de complexité cachée derrière le concept de portée, ou d'imbrication lexicale. Alors qu'accéder à une variable locale dans l'enregistrement d'activation local est efficace, accéder à des variables globales dans un langage structuré par blocs peut être très inéficcient. Cette section décrira comment un langage de haut niveau comme le Pascal traite les identifiants non-locaux et comment accéder à des variables globales et appeler des procédures et des fonctions non-locales.

[1]Notez que C et C++ ne sont pas des langages structurés par bloc. D'autres langages structurés par bloc incluent l'Algol, l'ADA et le Modula-2.

[2]Sous réserve, bien sûr, de ne pas réutiliser l'identifier dans la procédure imbriquée.

12.1.1 Portée

La portée, dans la plupart des langages de haut niveau, est un concept statique ou relatif en temps de compilation[33]. La portée est la notion permettant de savoir quand un nom est visible ou accessible, dans un programme. Cette capacité de cacher des noms est utile dans un programme parce qu'il est souvent commode de réutiliser certains noms (non-descriptifs). La variable i employée pour contrôler la plupart des boucles for dans des langages de haut niveau en est un exemple parfait. Dans tout ce chapitre vous avez vu des equates comme xyz_i, xyz_j, etc... La raison pour choisir de tels noms est que MASM ne supporte pas la même notion de portée des noms que les langages de haut niveau. Heureusement, MASM 6.x et plus récent *supporte* la portée des noms.

Par défaut, MASM 6.x traite les étiquettes d'instruction (celles avec deux points après elles) comme locales à une procédure. C'est-à-dire, vous ne pouvez référencer de telles étiquettes que dans la procédure dans laquelle elles sont déclarées. *Ceci demeure vrai même si vous imbriquez une procédure à l'intérieur d'une autre*. Heureusement, il n'y a aucune bonne raison pour laquelle on voudrait imbriquer des procédures dans un programme MASM.

Que les étiquettes soient locales dans une procédure est pratique. Cela vous permet de réutiliser des étiquettes d'instruction (par exemple, des étiquettes de boucle ou similaires) sans vous inquiéter des conflits de noms avec d'autres procédures. Parfois, cependant, vous pouvez avoir besoin de débloquer la portée des noms d'une procédure; un bon exemple est quand vous avez une instruction case dont la table de saut apparaît en dehors de la procédure. Si les étiquettes de l'instruction case sont locales, elles ne seront pas visibles en dehors de la procédure et vous ne pouvez pas les employer dans la table de saut de l'instruction case (voir "Instructions CASE" à la section 11.5). Il y a deux manières de débloquer la portée des étiquettes dans MASM 6.x. La première est d'inclure dans votre listing l'instruction:

```
option      noscoped
```

Ceci débloquera la portée des variable qui suivent cette déclaration dans le fichier source de votre programme. Vous pouvez rebloquer la portée avec une instruction de la forme

```
option      scoped
```

En plaçant ces instructions autour de votre procédure, vous pouvez sélectivement contrôler la portée.

Une autre manière de contrôler la portée de différents noms est de placer des deux points doubles ("::") après une étiquette. Ceci informe l'assembleur que ce nom particulier est global pour la procédure qui le contient.

MASM, comme le langage de programmation de C, supporte trois niveaux de portée: public, global (ou statique), et local. Les symboles locaux sont visibles seulement dans la procédure dans laquelle ils sont définis. Les symboles globaux sont accessibles dans tout un fichier source, mais ne sont pas visibles dans d'autres modules de programme. Les symboles publics sont visibles dans tout un programme, depuis tous les modules. MASM emploie les règles suivantes de portée par défaut:

- Les étiquettes d'instruction apparaissant dans une procédure sont locales à cette procédure
- Tous les noms de procédure sont publics
- La plupart des autres symboles sont globaux.

Notez que ces règles s'appliquent à MASM 6.x seulement. D'autres assembleurs et des versions antérieures de MASM suivent des règles différentes.

[3] Il existe des langages qui supportent la portée dynamique, ou d'exécution, mais ce texte n'en tient pas compte.

Le dépassement du défaut pour la première règle ci-dessus est facile, utilisez soit l'instruction option noscoped, soit des doubles deux points pour rendre une étiquette globale. Cependant, on ne peut pas rendre publique une étiquette locale en utilisant les directives public ou externdef. Vous devez rendre le symbole global (en utilisant l'une ou l'autre technique) avant que vous le rendiez public.

Avoir tous les noms de procédure publics par défaut ne pose habituellement pas de problème. Cependant, il se peut que vous vouliez employer le même nom de procédure (local) dans des modules différents. Si MASM rend automatiquement publics de tels noms, l'éditeur de liens vous signalera une erreur parce qu'il y a plusieurs procédures publiques ayant le même nom. Vous pouvez bloquer et débloquer cette action par défaut en utilisant les instructions suivantes:

```
option      proc:private        ;les procédures sont globales

option      proc:export         ;les procédures sont publiques
```

Notez que certains débugueurs ne fournissent d'information symbolique que si un nom de procédure est public. C'est pourquoi MASM 6.x rend par défaut les noms publics. Ce problème n'existe pas avec CodeView; ainsi vous pouvez employer le défaut le plus pratique. Naturellement, si vous choisissez de maintenir les noms de procédure privés (globaux seulement), alors vous devrez employer les directives public ou externdef pour rendre publics les noms de procédure voulus.

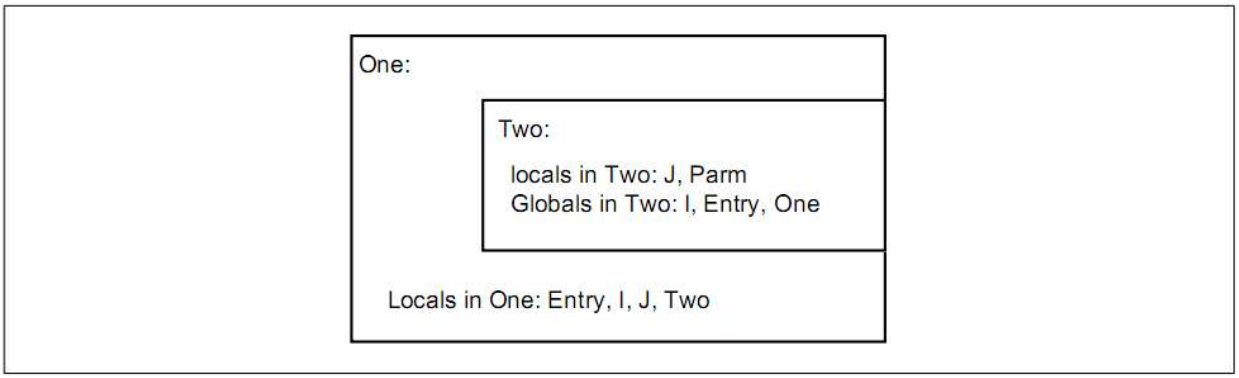

Figure 12.1 Identificateur de portée

Cette discussion des symboles locaux, globaux, et publics s'applique *principalement* aux étiquettes d'instruction et de procédure. Elle ne s'applique *pas* aux variables que vous avez déclaré dans votre segment de données, aux equates, aux macros, aux typedefs, ou à la plupart des autres symboles. De tels symboles sont toujours globaux indépendamment de l'endroit où vous les définissez. La seule manière de les rendre publics est de spécifier leur nom dans une directive public ou externdef.

Il y a un moyen de déclarer des noms de paramètre et des variables locales, alloués sur la pile, de manière à ce que leurs noms soient locaux à une procédure donnée. Voyez la directive proc du manuel de référence de MASM pour les détails.

La portée d'un nom limite sa visibilité dans un programme. C'est-à-dire, un programme a accès à un nom de variable seulement dans la portée de ce nom. En dehors de ça, le programme ne peut pas accéder à ce nom. Beaucoup de langages de programmation, comme le Pascal et le C++, vous permettent de réutiliser des identifiants si les portées de ces utilisations multiples ne se superposent pas. Comme vous l'avez vu, MASM fournit quelques dispositifs de portée minimaux pour les étiquettes d'instructions. Il y a, cependant, un autre problème relié à la portée*: la liaison d'adresse* et *la durée de vie d'une variable*. La liaison d'adresse est le processus consistant à associer une adresse de mémoire à un nom de variable. La vie d'une variable est cette partie d'une exécution de programme pendant laquelle un endroit de mémoire est lié à une variable. Considérez les procédures suivantes en Pascal:

```
procedure One(Entry:integer);
var
```

```
        i,j:integer;

        procedure Two(Parm:integer);
        var j:integer;
        begin

                for j:= 0 to 5 do writeln(i+j);
                if Parm < 10 then One(Parm+1);

        end;

begin {One}
        for i := 1 to 5 do Two(Entry);
end;
```

La Figure 12.1 montre à la portée des identifiants One, Two, Entry, i, j et Parm.

La variable locale j dans Two masque l'identifiant j de la procédure One tant qu'on est à l'intérieur de Two.

12.1.2 Activation d'unité, liaison d'adresse et durée de vie d'une variable

L'activation d'unité est le processus consistant à appeler une procédure ou une fonction. La combinaison d'un bloc d'activation et d'un certain code d'exécution est considérée une *instance* d'une routine. Quand l'activation d'unité se produit, une routine lie des adresses machine à ses variables locales. La liaison d'adresse (pour des variables locales) se produit quand la routine ajuste le pointeur de pile pour faire de la place pour les variables locales. La durée de vie de ces variables part de ce point jusqu'à ce que la routine détruise le bloc d'activation, éliminant le stockage local de variables.

Bien que la portée limite la visibilité d'un nom à une certaine section de code et ne permette pas la duplication des noms dans la même portée, ceci ne signifie pas qu'il y a seulement une adresse liée à un nom. Il est tout à fait possible d'avoir plusieurs adresses liées au même nom en même temps. Considérez un appel de procédure récursive. À chaque activation, la procédure construit un nouvel enregistrement d'activation. Puisque l'instance précédente existe toujours, il y a maintenant deux blocs d'activation sur la pile, contenant des variables locales pour cette procédure. Au fur et à mesure que des activation récursives supplémentaires se produisent, le système établit d'autres blocs d'activation, chacun avec une adresse liée au même nom. Pour résoudre l'ambiguïté possible (à quelle l'adresse accèdez-vous en agissant sur la variable ?), le système manipule toujours la variable dans le bloc d'activation le plus récent.

Notez que les procédures One et Two dans la section précédente sont *indirectement récursives*. C'est-à-dire, les deux appellent des routines qui, à leur tour, s'appellent. En supposant que le paramètre pour One est inférieur à 10 lors de l'appel initial, ce code produira des blocs d'activation multiples (et, en conséquence, des copies multiples des variables locales) sur la pile. Par exemple, si vous vouliez lancer l'appel One(9), la pile ressemblerait à la Figure 12.2 la première fois qu'on rencontre le end associé à la procédure Two.

Comme vous pouvez le voir, il y a plusieurs copies de I et de J sur la pile à ce pont. La procédure Two (la routine actuellement en exécution) accéderait à J dans le bloc d'activation le plus récent qui est en bas de la figure. L'instance précédente de deux accéde seulement à la variable J dans son bloc d'activation quand l'instance en cours retourne à One et puis de nouveau à Two.

La vie d'une instance de variable part du point de création du bloc d'activation jusqu'au point de destruction du bloc d'activation. Notez que la première instance de J ci-dessus (celle en haut du diagramme) a la vie la plus longue et que les vies de toutes les instances de J se chevauchent.

Figure 12.2 Récursion indirecte

12.1.3 Liens statiques

Le Pascal permet à la procédure Two d'accéder à I dans la procédure One. Cependant, quand il y a possibilité de récursion, il peut y avoir plusieurs instances de i sur la pile. Le Pascal, bien sûr, va seulement permettre à la procédure Two l'accès à l'instance la plus récente de I. Dans le diagramme de pile de la Figure 12.2, ceci correspond à la valeur de i dans le bloc d'activation qui commence par "One(9+1) parameter". Le seul problème est *comment savez-vous où trouver le bloc d'activation contenant i ?*

Une réponse rapide, mais mal aboutie, est de simplement constituer un index négatif dans la pile. Après tout, vous pouvez facilement voir dans le diagramme ci-dessus que I se trouve à l'offsett huit du bloc d'activation de Two. Malheureusement, ce n'est pas toujours le cas. Supposez que la procédure Three appelle également la procédure Two et que l'instruction suivante apparaît dans la procédure One:

```
If (Entry < 5) then Three(Entry*2) else Two(Entry);
```

Avec cette instruction en place, il est tout à fait possible d'avoir deux cadres de pile différents à l'entrée dans la procédure Two : un avec le bloc d'activation pour la procédure Three en sanwich entre les blocs d'activation de One et de Two et un avec les blocs d'activation pour les procédures One et Two côte à côte. Certainement, un offsett fixe à partir du bloc d'activation de Two ne pointera pas toujours sur la variable I sur le bloc d'activation le plus récent de One.

Le lecteur astucieux pourrait noter que la valeur de bp sauvée dans le bloc d'activation de Two pointe sur le bloc d'activation de l'appelant. Vous pourriez penser que vous pouvez l'employer comme pointeur sur le bloc d'activation de One. Mais cette technique échoue pour la même raison que la technique d'offset fixe échoue. L'ancienne valeur de Bp, le *lien dynamique*, pointe sur le bloc d'activation de l'appelant. Puisque celui-ci n'est pas nécessairement la procédure englobante, le lien dynamique pourrait ne pas pointer sur le bloc d'activation de cette dernière.

Ce dont on a en fait besoin, c'est d'un pointeur sur le bloc d'activation de cette procédure. Beaucoup de compilateurs pour des langages structurés par bloc créent un tel pointeur, *le lien statique*. Considérez le code Pascal suivant :

```
procedure Parent;
var i,j:integer;

      procedure Child1;
      var j:integer;
      begin

            for j := 0 to 2 do writeln(i);

      end {Child1};

      procedure Child2;
      var i:integer;
      begin

            for i := 0 to 1 do Child1;

      end {Child2};

begin {Parent}

      Child2;
      Child1;

end;
```

Juste aprés être entré dans Child1 pour la première fois, la pile ressemblerait à la Figure 12.3. Quand Child1 essaye d'accéder à la variable i de Parent, il aura besoin d'un pointeur, le lien statique, sur le bloc d'activation de Parent. Malheureusement, il n'y a aucune manière pour Child1, dès l'entrée, de trouver tout seul où le bloc d'activation de Parent se situe dans la mémoire. Il faudra que l'appelant, (Child2 en l'occurence) passe le lien statique à Child1. En général, l'appelé peut traiter le lien statique tout comme un autre paramètre; habituellement poussé sur la pile juste avant d'exécuter l'instruction d'appel.

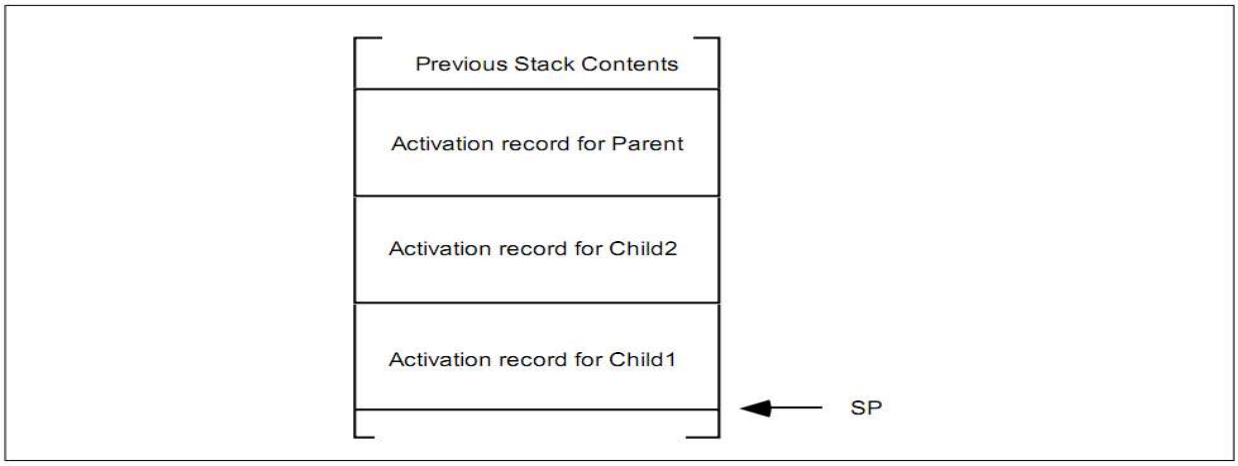

Figure 12.3 Enregistrement d'activation après plusieurs appels imbriqués

Pour comprendre entièrement comment passer des liens statiques d'appel en appel, vous devez d'abord comprendre le concept de niveau lexical. Les niveaux lexicaux en Pascal correspondent aux niveaux d'imbrication statique des procédures et des fonctions. La plupart des créateurs de compilateur définissent le niveau lexical zéro comme le programme principal. C'est-à-dire, tous les symboles que vous déclarez dans votre programme principal existent au niveau lexical zéro. Les noms de procédure et de fonction apparaissant dans votre programme principal définissent le niveau lexical un, *peu importe le nombre de procédures ou fonctions apparaissant dans le programme principal*. Ils commencent tous une nouvelle copie du niveau lexical un. Pour chaque niveau d'emboîtement, le Pascal présente un nouveau niveau lexical. La Figure 12.4 le montre.

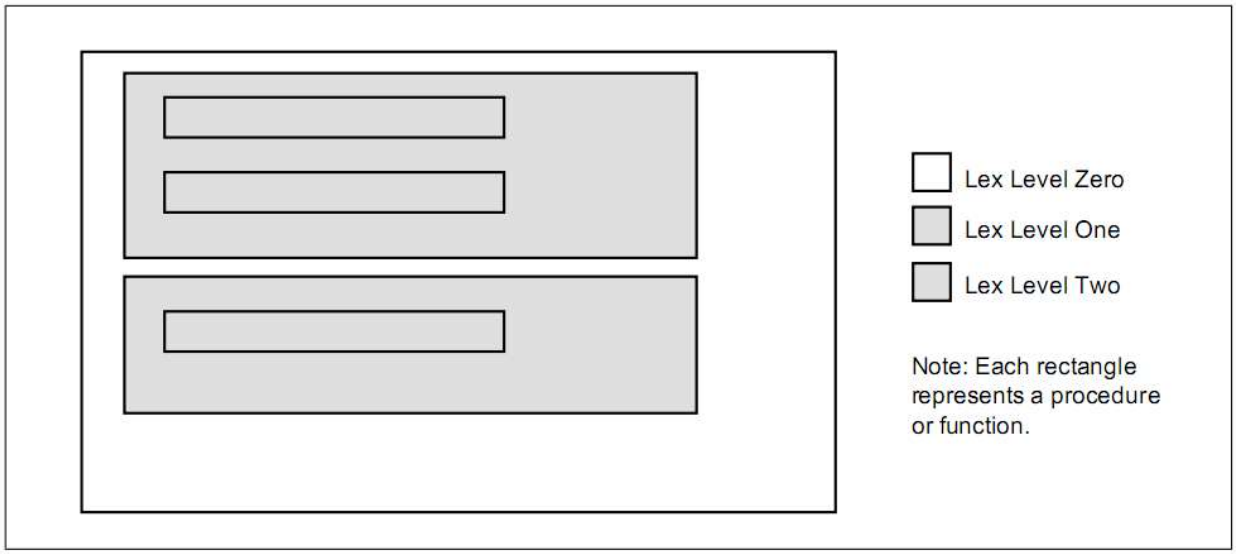

Figure 12.4 Schéma de Procédures Montrant les Niveaux Lexicaux.

Pendant l'exécution, un programme ne peut accéder qu'à des variables à un niveau lexical inférieur ou égal au niveau de la routine en cours. En outre, un seul ensemble de valeurs à un niveau lexical donné sont accessibles à un moment donné[44] et ces valeurs sont toujours dans le bloc d'activation le plus récent à ce niveau lexical.

Avant de se demander comment accéder à des variables non-locales en utilisant un lien statique, vous devez trouver comment passer le lien statique comme paramètre. Quand on passe le lien

[4] Il y a une exception. Si vous avez un pointeur sur une variable et que le pointeur reste accessible, vous pouvez accéder aux données sur lesquelles il pointe même si la variable contenant réellement les données est inaccessible. Naturellement, en Pascal (standard) vous ne pouvez pas prendre l'adresse d'une variable locale et la mettre dans un pointeur. Cependant, certains dialectes de Pascal (par exemple, Turbo) et d'autres langages structurés par bloc permettent cette opération.

statique comme paramètre à une unité de programme (procédure ou fonction), il y a trois types de séquences d'appel que l'on peut rencontrer :

- Une unité de programme appelle une procédure ou une fonction *enfant*. Si le niveau lexical en cours est n, alors une procédure ou une fonction enfant est au niveau lexical *n+1* et est locale à l'unité de programme en cours. Notez que la plupart des langages structurés par bloc ne permettent pas d'appeler des procédures ou des fonctions à des niveaux lexicaux supérieurs à *n+1*.
- Une unité de programme appelle une procédure ou une fonction *sœur (peer)*. Une procédure ou une fonction sœur est une procédure au même niveau lexical que l'appelant en cours et une unité unique de programme englobe les deux unités de programme.
- Une unité de programme appelle une procédure ou une fonction *ancêtre*. Une unité ancêtre est soit l'unité parent, soit un parent d'une unité ancêtre, soit une soeur d'une unité ancêtre.

Les séquences d'appel pour les deux premiers types d'appels ci-dessus sont très simples. Pour cet exemple, supposez que le bloc d'activation pour ces procédures prenne la forme générique de la Figure 12.5.

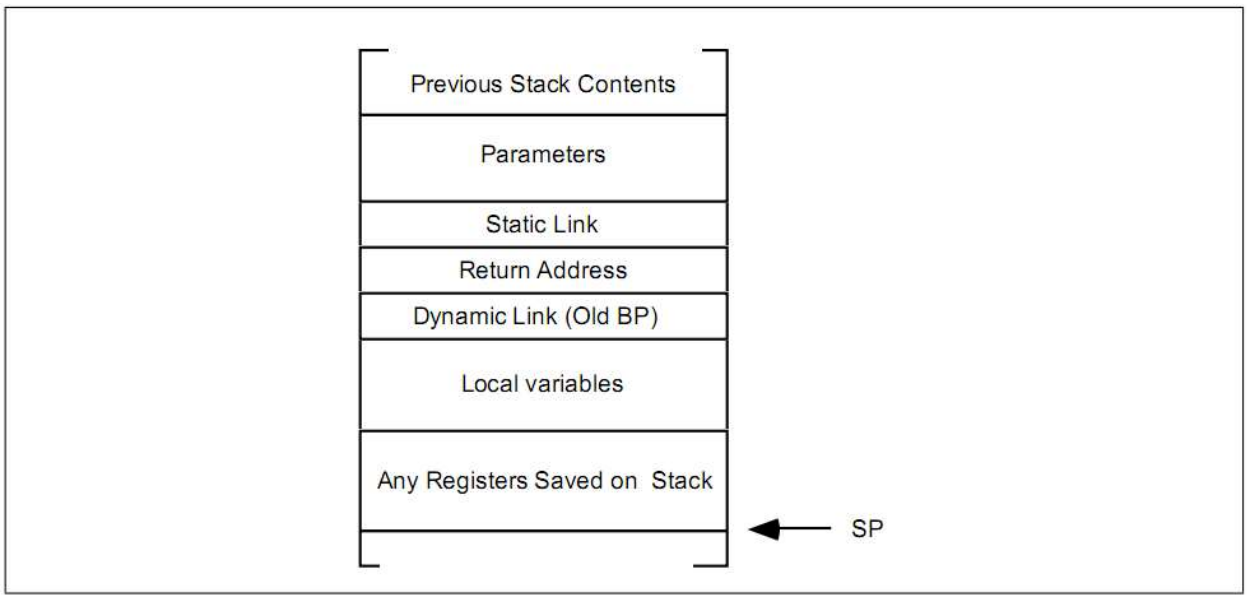

Figure 12.5 Bloc d'Activation Générique

Quand une procédure ou une fonction parent appelle une unité de programme fils, le lien statique n'est rien d'autre que la valeur dans le registre de bp immédiatement avant l'appel. Par conséquent, pour passer le lien statique à l'unité enfant, il suffit de pousser bp avant d'exécuter l'instruction d'appel:

```
<Pousser d'autres paramètres sur le pile>

        push   bp

        call   ChildUnit
```

Bien sûr, l'unité d'enfant peut traiter le lien statique sur la pile tout comme n'importe quel autre paramètre. Dans ce cas, les liens statiques et dynamiques sont exactement identiques. En général, cependant, ce n'est pas vrai.

Si une unité de programme appelle une procédure ou une fonction soeur, la valeur actuelle dans bp n'est pas le lien statique. C'est un pointeur sur les variables locales de l'appelant et la procédure soeur ne peut pas accéder à ces variables. Cependant, comme soeurs, l'appelant et l'appelé partagent la même unité de programme parent, aussi l'appelant peut simplement pousser une copie

de son lien statique sur la pile avant d'appeler la procédure ou la fonction soeur. Le code suivant réalise ceci, supposant que toutes les procédures et fonctions sont near:

```
<Pousser d'autres paramètres sur la pile>
push   [bp + 4]           ;Pousser le lien statique sur la pile
call   PeerUnit
```

Si la procédure ou la fonction est far, le lien statique se trouverait deux bytes plus haut sur la pile, aussi vous devriez employer le code suivant:

```
<Pousser d'autres paramètres sur la pile>
push   [bp + 6]           ;Pousser le lien statique sur la pile
call   PeerUnit
```

Appeller un ancêtre est un peu plus complexe. Si vous êtes actuellement au niveau lexical *n* et que vous souhaitez appeler un ancêtre au niveau lexical *m (m < n),* vous devrez traverser la liste des liens statiques pour trouver le bloc d'activation désiré. Les liens statiques forment une liste de blocs d'activation. En suivant cette chaîne de blocs d'activation jusqu'au bout, vous pouvez tracer à travers ces derniers les plus récents de toutes les procédures et fonctions entourant une unité de programme donnée. Le diagramme de pile de la Figure 12.6 montre les liens statiques pour une séquence d'appels de procédures statiquement imbriquée sur cinq niveaux lexicaux de profondeur.

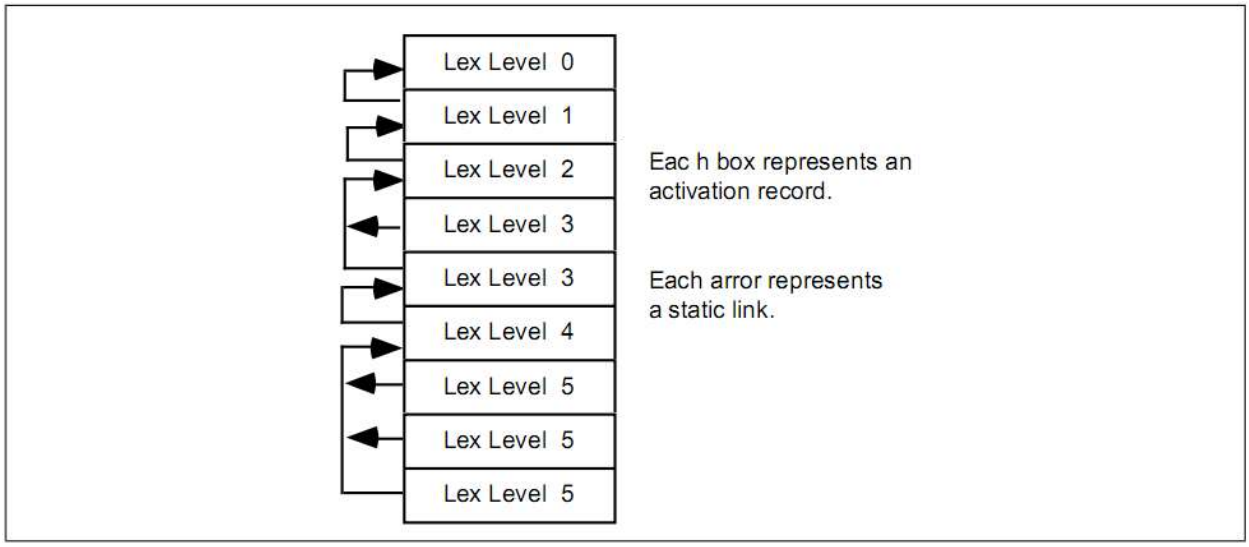

Figure 12.6 Liens Statiques

Si l'unité de programme qui s'exécute actuellement au niveau lexical cinq souhaite appeler une procédure au niveau lexical trois, elle doit pousser un lien statique sur l'untité de programme activée le plus récemment au niveau lexical deux. Pour trouver ce lien statique, vous devrez *traverser* la chaîne de liens statiques. Si vous êtes au niveau lexical *n* et que vous voulez appeler une procédure au niveau lexical *m,* vous devrez traverser *(n-m)+1* liens statiques. Le code pour réaliser ceci est

```
; Le niveau lexical actuel est 5. Ce code localise le lien statique pour une
; procédure au niveau 2 et puis l'appelle. Supposez que tous les appels sont near:

      <Pousser les paramètres nécéssaires>

            mov    bx, [bp+4]          ;Traverser le lien statique vers LL 4.
            mov    bx, ss:[bx+4]       ;Vers Lex Level 3.
            mov    bx, ss:[bx+4]       ;Vers Lex Level 2.
            push   ss:[bx+4]           ;Ptr sur le B.A. LL1 le plus récent
            call   ProcAtLL2
```

Notez le préfixe ss: dans les instructions ci-dessus. Rappelez-vous, les blocs d'activation sont tous dans le segment de pile et bx indexe le segment de données par défaut.

12.1.4 Accès à des variables non-locales en utilisant des liens statiques

Afin d'accéder à une variable non-locale, vous devez traverser la chaîne des liens statiques jusqu'à ce que vous obteniez un pointeur sur le bloc d'activation désiré. Cette opération revient à localiser le lien statique pour un appel de procédure comme décrit dans la section précédente, sauf que vous traversiez seulement *n-m* liens statiques au lieu de *(n-m)+1* pour obtenir un pointeur sur le bloc d'activation approprié. Considérez le code suivant en Pascal:

```
procedure Outer;
var i:integer;

      procedure Middle;
      var j:integer;

            procedure Inner;
            var k:integer;
            begin

                  k := 3;
                  writeln(i+j+k);

            end;

      begin {middle}

            j := 2;
            writeln(i+j);
            Inner;

      end; {middle}

begin {Outer}

      i := 1;
      Middle;

end; {Outer}
```

La procédure Inner accède à des variables globales au niveau lexical *n-1* et *n-2* (où *n* est le niveau lexical de la procédure Inner). La procédure Middle accède à une variable globale unique au niveau lexical *m-1* (où *m* est le niveau lexical de la procédure Middle). Le code suivant en assembleur pourrait implémenter ces trois procédures:

```
Outer       proc  near
            push  bp
            mov   bp, sp
            sub   sp, 2               ;Fait de la place pour I.

            mov   word ptr [bp-2],1   ;Met I à un.
            push  bp                  ;Lien Statique pour Middle.
            call  Middle

            mov   sp, bp              ;Enlève les variables locales.
            pop   bp
            ret   2                   ;Enlève le lien statique au retour.

Outer       endp

Middle      proc  near
            push  bp                  ;Sauve le lien dynamique
            mov   bp, sp              ;Établit le bloc d'activation.
            sub   sp, 2               ;Fait de la place pour J.
```

```
                mov     word ptr [bp-2],2   ;J := 2;
                mov     bx, [bp+4]          ;Met le lien statique au LL précédent.
                mov     ax, ss:[bx-2]       ;Obtient la valeur de I.
                add     ax, [bp-2]          ;L'aditionne à J, puis
                puti                        ; affiche la somme.
                putcr
                push    bp                  ;Lien Statique pour Inner.
                call    Inner

                mov     sp, bp
                pop     bp
                ret     2                   ;Enlève le lien statique au RET.
Middle          endp

Inner           proc    near
                push    bp                  ;Sauve le lien dynamique
                mov     bp, sp              ;Établit le bloc d'activation.
                sub     sp, 2               ;Fait de la place pour K.

                mov     word ptr [bp-2],2   ;K := 3;
                mov     bx, [bp+4]          ;Met le lien statique au LL précédent.
                mov     ax, ss:[bx-2]       ;Obtient la valeur de J.
                add     ax, [bp-2]          ;L'aditionne à K

                mov     bx, ss:[bx+4]       ;Obtient ptr sur Bloc d'Act de Outer.
                add     ax, ss:[bx-2]       ;Additionne la valeur de I, puis
                puti                        ; affiche la somme.
                Putcr

                mov     sp, bp
                pop     bp
                ret     2                   ;Enlève le lien statique au RET.
Inner           endp
```

Comme vous pouvez le voir, accéder à des variables globales peut être très inefficace[55].

Notez que comme la différence entre les blocs d'activation augmente, il devient de moins en moins efficace d'accéder à des variables globales. L'accès à ces variables dans le bloc d'activation précédent exige seulement une instruction supplémentaire par accès, à deux niveaux lexicaux, vous avez besoin de deux instructions supplémentaires, etc... Si vous analysez un grand nombre de programmes en Pascal, vous constaterez que la plupart d'entre eux n'imbriquent pas les procédures et les fonctions et dans ceux où il y a des unités de programme imbriquées, elles accèdent rarement à des variables globales. Il y a une exception majeure, cependant. Bien que les procédures et les fonctions Pascal accèdent rarement à des variables locales à l'intérieur d'autres procédures et fonctions, elles accèdent fréquemment à des variables globales déclarées dans le programme principal. Puisque de telles variables apparaissent au niveau lexical zéro, l'accès à celles-ci serait aussi inefficace que possible en utilisant des liens statiques. Pour résoudre ce problème mineur, la plupart des langages structurés par bloc basés sur le 80x86 assignent des variables au niveau lexical zéro directement dans le segment de données et y accèdent directement.

12.1.5 La Table Lexicale

À la lecture de la section précédente, vous pourriez avoir l'idée qu'on devrait ne jamais employer des variables non-locales, ou limiter les accès non-locaux aux variables déclarées au niveau lexical zéro seul. Après tout, it souvent assez facile de mettre toutes les variables partagées au niveau lexical zéro. Si vous concevez un langage de programmation, vous pouvez adopter la philosophie des concepteurs du C et tout simplement ne pas fournir de structures de bloc. De tels compromis s'avèrent être inutiles. Il y a une structure de données, la *table lexicale (display)*, qui fournit un accès efficace à *tout* ensemble de variables non-locales.

[5] En fait, une des raisons principales pour laquelle le langage de programmation C n'est pas structuré par blocs, est peut-être parce que les concepteurs du langage ont voulu éviter l'accès inefficace à des variables non-locales.

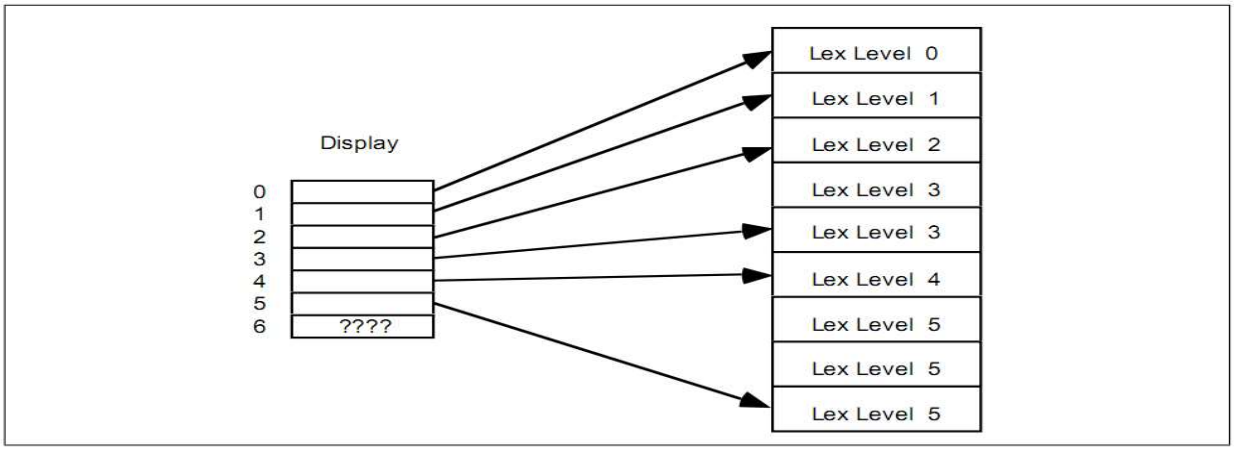

Figure 12.7 La Table Lexicale

Une table lexicale est simplement un tableau de pointeurs sur des blocs d'activation. Display[0] contient un pointeur sur le bloc d'activation le plus récent pour le niveau lexical zéro, Display[1] contient un pointeur sur le bloc d'activation le plus récent pour le niveau lexical un, et ainsi de suite. En partant du principe que vous avez placé la table Display dans le segment de données en cours (toujours un bon endroit où la garder), cela prend seulement deux instructions pour accéder à n'importe quelle variable non-locale. D'une manière imagée, une table lexicale fonctionne comme représenté sur la Figure 12.7.

Notez que les entrées dans la table lexicale pointent toujours sur le bloc d'activation le plus récent pour une procédure à un niveau lexical donné. S'il n'y a aucun bloc d'activation actif pour un niveau lexical particulier (par exemple, niveau lexical six ci-dessus), alors l'entrée dans la table contient des données incohérentes.

Le niveau maximal d'imbrication lexicale dans votre programme détermine combien d'éléments il doit y avoir dans la table lexicale. La plupart des programmes ont seulement trois ou quatre procédures emboîtées (s'ils en ont), aussi la table lexicale est habituellement très petite. En général, vous aurez rarement besoin de plus de 10 éléments dans la table lexicale.

Un autre avantage à utiliser une table lexicale est que chaque procédure individuelle peut maintenir l'information de la table elle-même, l'appelant n'a pas besoin d'être impliqué. Quand on employe des liens statiques, le code appelant doit calculer et passer le lien statique approprié à une procédure. Non seulement c'est lent, mais le code pour faire ceci doit apparaître avant chaque appel. Si votre programme utilise une table lexicale, l'appelé, au lieu de l'appelant, maintient la table lexicale ce qui fait que vous n'avez besoin que d'une copie du code par procédure. En outre, comme le montre le prochain exemple, le code pour manipuler la table lexicale est court et rapide.

Le maintien de la table lexicale est très facile. Lors de l'entrée initiale dans une procédure vous devez d'abord sauvegarder le contenu de la table lexicale au niveau lexical en cours et ensuite stocker le pointeur sur le bloc d'activation en cours dans ce même endroit. L'accès à une variable non-locale exige seulement deux instructions, une pour charger un élément de la table lexicale dans un registre et une seconde pour accéder à la variable. Le code suivant met en application les procédures Outer, Middle, et Inner des exemples de lien statique.

```
; Supposez qu'Outer est au niveau lexical 1, Middle au niveau lexical 2
; et Inner au niveau lexical 3. Gardez à l'esprit que chaque entrée de la
; table lexicale fait deux octets. Vraisemblablement, la variable Display
; est définie dans le segment de données.

Outer        proc   near
             push   bp
             mov    bp, sp
             push   Display[2]          ;Sauve l'Entrée courante de Display
             sub    sp, 2               ;Fait de la place pour I.
             mov    word ptr [bp-4],1   ;Met I à un.
             call   Middle
```

```
            add     sp, 2               ;Enlève les variables locales
            pop     Display[2]          ;Restaure la valeur précédente.
            pop     bp
            ret
Outer       endp

Middle      proc    near
            push    bp                  ;Sauve le lien dynamique.
            mov     bp, sp              ;Établit notre bloc d'activation.
            push    Display[4]          ;Sauve l'ancienne valeur de Display.
            sub     sp, 2               ;Fait de la place pour J.

            mov     word ptr [bp-2],2   ;J := 2;
            mov     bx, Display[2]      ;Obtient le lien statique du LL précédent.
            mov     ax, ss:[bx-4]       ;Obtient la valeur de I.
            add     ax, [bp-2]          ;Ajoute à J et puis
            puti                        ; affiche la somme.
            putcr
            call    Inner

            add     sp, 2               ;Enlève la variable locale.
            pop     Display[4]          ;Restaure l'ancienne valeur de Display.
            pop     bp
            ret
Middle      endp

Inner       proc    near
            push    bp                  ;Sauve le lien dynamique
            mov     bp, sp              ;Établit le bloc d'activation.
            push    Display[6]          ;Sauve l'ancienne valeur de Display.
            sub     sp, 2               ;Fait de la place pour K.

            mov     word ptr [bp-2],2   ;K := 3;
            mov     bx, Display[4]      ;Obtient le lien statique du LL précédent..
            mov     ax, ss:[bx-4]       ;Obtient la valeur de J.
            add     ax, [bp-2]          ;Ajoute à K
            mov     bx, Display[2]      ;Obtient le ptr sur le Bloc d'Act d'Outer.
            add     ax, ss:[bx-4]       ;Ajoute la valeur de I et puis
            puti                        ; affiche la somme.
            putcr
            add     sp, 2
            pop     Display [6]
            pop     bp
            ret
Inner       endp
```

Bien que ce code ne paraisse pas particulièrement meilleur que l'ancien code, le fait d'employer une table lexicale est souvent beaucoup plus efficace qu'utiliser des liens statiques.

12.1.6 Les instructions ENTER et LEAVE du 80286

Pour le 80286, les concepteurs de CPU d'Intel ont décidé d'ajouter deux instructions pour aider à maintenir les tables lexicales. Malheureusement, bien que leur solution fonctionne, soit très générale, et n'exige que des données dans le segment de pile, elle est très lente; beaucoup plus lente qu'en utilisant les techniques de la section précédente. Bien que beaucoup de compilateurs non-optimisés utilisent ces instructions, les meilleurs compilateurs évitent, si possible, de les employer.

L'instruction **leave** est très simple à comprendre. Elle effectue la même opération que les deux instructions:

```
mov sp, bp
pop bp
```

Par conséquent, vous pouvez employer l'instruction pour le code standard de sortie de procédure si vous avez un microprocesseur 80286 ou postérieur. Sur un processeur 80386 ou plus ancien,

l'instruction **leave** est plus courte et plus rapide que la séquence mov et pop équivalente. Cependant, elle est plus lente sur les processeurs 80486 et postérieurs.

L'instruction **enter** prend deux opérandes. La première est le nombre de bytes de stockage local que la procédure en cours exige, la seconde est le niveau lexical de la procédure en cours. L'instruction **enter** fait ce qui suit:

```
; ENTER Locals, LexLevel

                    push   bp                ;Sauve le lien dynamique.
                    mov    tempreg, sp       ;Sauve pour plus tard.
                    cmp    LexLevel, 0       ;Fini si on est au niveau lex zéro.
                    je     Lex0
lp:                 dec LexLevel
                    jz     Done              ;Quitte si au dernier niveau lex.
                    sub    bp, 2             ;Index dans la table lex dans le BA préc
                    push   [bp]              ; et y pousse chaque élément.
                    jmp    lp                ;Répète pour chaque entrée.

Done:               push   tempreg           ;Ajoute entrée pour niveau lex en cours.
Lex0:               mov    bp, tempreg       ;Ptr sur bloc d'act en cours.
                    sub    sp, Locals        ;Alloue stockage local
```

Comme vous pouvez le voir dans ce code, l'instruction **enter** copie la table lexicale de bloc d'activation en bloc d'activation. Ceci peut se révéler très couteux si vous imbriquez les procédures à n'importe quelle profondeur. La plupart des langages de haut niveau, si jamais ils utilisent l'instruction **enter**, ils indiquent toujours un niveau d'imbrication de zéro pour éviter de copier la table lexicale à travers toute la pile.

L'instruction **enter** met la valeur de l'entrée display[n] dans l'emplacement BP-(n*2). *L'instruction **enter** ne copie pas la valeur de display[0] dans chaque cadre de pile*. Intel suppose que vous maintiendrez les variables globales du programme principal dans le segment de données. Pour économiser du temps et de la mémoire, elles ne prennent pas la peine de copier l'entrée display[0].

L'instruction **enter** est très lente, en particulier sur les processeurs 80486 et postérieurs. Si vous voulez vraiment copier la table lexicale de bloc d'activation en bloc d'activation c'est probablement une meilleure idée de pousser les éléments vous-même. Les petits bouts de code suivants montrent comment faire:

```
; enter n, 0              ;14 cycles on the 486

      push   bp            ;1 cycle on the 486
      sub    sp, n         ;1 cycle on the 486

; enter n, 1              ;17 cycles on the 486

      push   bp            ;1 cycle on the 486
      push   [bp-2]        ;4 cycles on the 486
      mov    bp, sp        ;1 cycle on the 486
      add    bp, 2         ;1 cycle on the 486
      sub    sp, n         ;1 cycle on the 486

; enter n, 2              ;20 cycles on the 486

      push   bp            ;1 cycle on the 486
      push   [bp-2]        ;4 cycles on the 486
      push   [bp-4]        ;4 cycles on the 486
      mov    bp, sp        ;1 cycle on the 486
      add    bp, 4         ;1 cycle on the 486
      sub    sp, n         ;1 cycle on the 486

; enter n, 3;23 cycles on the 486

      push   bp            ;1 cycle on the 486
      push   [bp-2]        ;4 cycles on the 486
      push   [bp-4]        ;4 cycles on the 486
```

```
        push   [bp-6]       ;4 cycles on the 486
        mov    bp, sp       ;1 cycle on the 486
        add    bp, 6        ;1 cycle on the 486
        sub    sp, n        ;1 cycle on the 486.

; enter n, 4;26 cycles on the 486

        push   bp           ;1 cycle on the 486
        push   [bp-2]       ;4 cycles on the 486
        push   [bp-4]       ;4 cycles on the 486
        push   [bp-6]       ;4 cycles on the 486
        push   [bp-8]       ;4 cycles on the 486
        mov    bp, sp       ;1 cycle on the 486
        add    bp, 8        ;1 cycle on the 486
        sub    sp, n        ;1 cycle on the 486

; etc.
```

Si vous voulez bien faire confiance aux chronmétrages de cycle d'Intel, vous pouvez voir que l'instruction **enter** n'est presque *jamais* plus rapide qu'une séquence banale d'instructions qui accomplissent la même chose. Si vous êtes intéressé par l'économie d'espace plutôt que l'écriture de code rapide, l'instruction enter est généralement une meilleure alternative. La même chose est généralement vraie pour l'instruction leave. Elle ne fait qu'un byte de long, mais elle est plus lente que les instructions correspondantes **mov bp,sp** et **pop bp**.

L'accès à des variables non-locales en utilisant les tables lexicales créées par **enter** apparaît dans les exercices.

12.2 Passage de variables à différents niveaux lexicaux comme paramètres

L'accès des variables à différents niveaux lexicaux dans un programme structuré par bloc apporte des complications à un programme. La section précédente vous a présenté la complication de l'accès à des variables non-locales. Ce problème devient encore pire quand vous essayez de passer de telles variables comme paramètres à une autre unité de programme. Les sous-sections suivantes traitent des stratégies pour chacun des mécanismes principaux de passage de paramètre.

Pour l'intérêt de l'argument, les sections suivantes présumeront que “ locales ” se réfère à des variables dans le bloc d'activation en cours, “ globales ” se réfère aux variables dans le segment de données et “ intermédiaires ” à des variables dans un bloc d'activation autre que le bloc d'activation en cours. Notez que les sections suivantes ne supposeront pas que **ds** est égal à **ss**. Ces sections passeront également tous les paramètres sur la pile. Vous pouvez facilement modifier les détails pour passer ces paramètres ailleurs.

12.2.1 Passage de paramètres par valeur dans un langage structuré par blocs

Passer des paramètres par valeur à une unité de programme n'est pas plus difficile que d''accéder aux variables correspondantes; tout ce dont vous avez besoin est de pousser la valeur sur la pile avant d'appeler la procédure associée.

Pour passer une variable globale par valeur à une autre procédure, vous pourriez employer un code comme ce qui suit:

```
        push   GlobalVar          ; Supposer que “GlobalVar” est dans DSEG.
        call   Procedure
```

Pour passer une variable locale par valeur à une autre procédure, vous pourriez employer le code suivant[66]:

```
push   [bp-2]              ;Variable locale dans le bloc d'activation
call   Procedure           ; en cours.
```

Pour passer une variable intermédiaire comme paramètre par valeur, vous devez d'abord localiser le bloc d'activation de cette variable intermédiaire et puis pousser sa valeur sur la pile. Le mécanisme exact que vous utilisez dépend de l'emploi de liens statiques ou d'une table lexicale pour garder une trace des blocs d'activation de la variable intermédiaire. Si vous utilisez des liens statiques, vous pourriez employer un code comme le suivant pour passer une variable depuis deux niveaux lexicaux au-dessus de la procédure en cours:

```
mov    bx, [bp+4]          ;Supposer que L.S. est à l'offset 4.
mov    bx, ss:[bx+4]       ;Traverse deux liens statiques
push   ss:[bx-2]           ;Pousse le valeur des variables.
call   Procedure
```

Passer une variable intermédiaire par valeur quand vous employez une table lexicale est légèrement plus facile. Vous pourriez employer du code comme le suivant pour passer une variable intermédiaire depuis le niveau lexical un:

```
mov    bx, Display[1*2]    ;Obtient l'entrée Display[1].
push   ss:[bx-2]           ;Pousse la valeur de la variable.
call Procedure
```

12.2.2 Passage de paramètres par référence, résultat et valeur-résultat dans un langage structuré par blocs

Les mécanismes de passage de paramètre par référence, résultat, et valeur-résultat passent généralement l'adresse du paramètre sur la pile[77]. Si les variables globales résident dans le segment de données, les blocs d'activation existent tous dans le segment de pile et ds≠ss, alors vous devez passer des pointeurs far pour accéder à toutes les variables possibles[88].

Pour passer un pointeur far, vous devez pousser une valeur de segment suivie d'une valeur d'offset sur la pile. Pour des variables globales, la valeur de segment se trouve dans le registre **ds**; pour des valeurs non-globales, ss contient la valeur de segment. Pour calculer la partie offset de l'adresse vous devriez normalement utiliser l'instruction **lea**. La séquence de code suivante passe une variable globale par référence:

```
push   ds                  ;Pousse d'abord adrs du segment.
lea    ax, GlobalVar       ;Calcule l'offset.
push   ax                  ;Pousse l'offset de GlobalVar
call   Procedure
```

Les variables globales sont un cas spécial parce que l'assembleur peut calculer leurs offsetts à l'exécution lors de l'assemblage. Par conséquent, *pour des variables globales scalaires seulement*, nous pouvons raccourcir la séquence de code ci-dessus en

```
push   ds                  ;Pousse adrs du segment.
push   offset GlobalVar    ;Pousse la partie offset.
call   Procedure
```

Pour passer une variable locale par référence, votre code doit d'abord pousser la valeur de **ss** sur la pile et ensuite pousser l'offset de la variable locale. *Cet offset est l'offset de la variable dans le segment de pile, pas l'offset dans le bloc d'activation!* Le code suivant passe l'adresse d'une variable locale par référence:

```
push   ss                  ;Pousse l'adresse du segment.
```

[6] Les exemples non-globaux présument tous que la variable est à l'offsett -2 dans leur bloc d'activation. Modifiez ceci comme approprié dans votre code.

[7] Comme vous pouvez vous le rappeler, les passages par référence, valeur-résultat, et résultat utilisent tous la même séquence d'appel. Les différences se situent dans les procédures elles-mêmes.

[8] Vous pouvez employer des pointeurs near si ds=ss ou si vous maintenez les variables globales dans le bloc d'activation du programme principal dans le segment de pile

```
lea    ax, [bp-2]          ;Calcule l'offset de la variable
push   ax                  ; locale et la pousse.
call   Procedure
```

Pour passer une variable intermédiaire par référence, vous devez d'abord localiser le bloc d'activation contenant cette variable de manière à pouvoir calculer l'adresse réelle dans le segment de pile. En utilisant des liens statiques, le code pour passer l'adresse du paramètre pourrait ressembler à ce qui suit:

```
push   ss                  ;Pousse la partie segment.
mov    bx, [bp+4]          ;Suppose que S.L. est à l'offset 4.
mov    bx, ss:[bx+4]       ;Traverse deux liens statiques
lea    ax, [bx-2]          ;Calcule l'addresse réelle
push   ax                  ;Pousse la partie offset.
call   Procedure
```

En utilisant une table lexicale, la séquence d'appel pourrait ressembler à ce qui suit:

```
push   ss                  ;Pousse la partie segment.
mov    bx, Display[1*2]    ;Obtient l'entrée Display[1].
lea    ax, [bx-2]          ;Obtient l'offset de la variable
push   ax                  ; et la pousse.
call   Procedure
```

Comme vous pouvez vous le rappeler du chapitre précédent, il y a une deuxième manière de passer un paramètre par valeur-résultat. Vous pouvez pousser la valeur sur la pile et ensuite, quand la procédure retourne, extraire cette valeur de la pile et la stocker de nouveau dans la variable d'où elle est venue en premier lieu. Ceci est juste un cas spécial du mécanisme de passage par valeur décrit dans la section précédente.

12.2.2 Passage de paramètres par nom et évaluation-paresseuse dans un langage structuré par blocs

Puisque vous passez l'adresse d'un *thunk*[aa] lorsque vous passez des paramètres par nom ou par évaluation-paresseuse, la présence de variables globales, intermédiaires et locales n'affecte pas la séquence d'appel de la procédure. Au lieu de cela, le *thunk* doit trouver les emplacements différents de ces variables. Les exemples suivants présenteront des *thunks* pour le passage par nom, vous pouvez facilement modifier ces *thunks* pour des paramètres par évaluation-paresseuse.

Le plus grand problème qu'un *thunk* a est de localiser le bloc d'activation contenant la variable dont il renvoie l'adresse. Dans le chapitre précédent, ce n'était pas trop un problème puisque les variables existaient soit dans le bloc d'activation en cours, soit dans l'espace global de données. En présence de variables intermédiaires, cette tâche devient légèrement plus complexe. La solution la plus facile est de passer deux pointeurs lorsqu'on passe une variable par nom. Le premier pointeur devrait être l'adresse du thunk, le deuxième pointeur devrait être l'offsett du bloc d'activation contenant la variable à laquelle le thunk doit accéder[99]. Quand la procédure appelle le thunk, il doit passer comme paramètre au thunk l'offset de ce bloc d'activation. Considérez les procédures suivantes de Panacea:

```
TestThunk:procedure(name item:integer; var j:integer);
begin TestThunk;
      for j in 0..9 do item := 0;
end TestThunk;
```

[a] En programmation informatique, un thunk est une sous-routine utilisée pour injecter un calcul dans une autre sous-routine. Les thunks sont principalement utilisés pour retarder un calcul jusqu'à ce que son résultat soit nécessaire, ou pour insérer des opérations au début ou à la fin de l'autre sous-routine [*cette note est du traducteur, c'est pourquoi elle une numérotation par lettres*].

[9] En fait, vous pouvez avoir besoin de passer plusieurs pointeurs sur des blocs d'activation. Par exemple, si vous passez la variable " A[i, j, k] " par nom et si A, I, j et k sont tous dans différents blocs d'activation, vous devrez passer des pointeurs sur chaque bloc d'activation . Nous ignorerons ce problème ici.

```
CallThunk:procedure;
var
      A: array[0..9] : integer;
      I: integer;
endvar;
begin CallThunk;
      TestThunk(A[I], I);
end CallThunk;
```

Le code assembleur pour ci-dessus pourrait ressembler à ce qui suit:

```
; TestThunk BA:
;
;     BP+10-       Addresse du thunk

;     BP+8-        Ptr sur BA pour params Item et J (doivent être dans
; le même BA).
;     BP+4-        Far ptr sur J.

TestThunk    proc  near
             push  bp
             mov   bp, sp
             push  ax
             push  bx
             push  es

             les   bx, [bp+4]          ;Obtient ptr sur J.
             mov   word ptr es:[bx], 0 ;J := 0;
ForLoop:     cmp   word ptr es:[bx], 9 ;J > 9?
             ja    ForDone
             push  [bp+8]              ;Pousse BA passé par appelant.
             call  word ptr [bp+10]    ;Appelle le thunk.
             mov   word ptr ss:[bx], 0 ;Thunk retourne adrs dans BX.
             les   bx, [bp+4]          ;Obtient ptr sur J.
             inc   word ptr es:[bx]    ;Y ajoute un.
             jmp   ForLoop

ForDone:     pop   es
             pop   bx
             pop   ax
             pop   bp
             ret   8
TestThunk    endp

CallThunk    proc  near
             push  bp
             mov   bp, sp
             sub   sp, 12              ;Fait de la place pour locales.

             jmp   OverThunk
Thunk        proc
             push  bp
             mov   bp, sp
             mov   bp, [bp+4]          ;Obtient address du BA.
             mov   ax, [bp-22]         ;Obtient valeur de I.
             add   ax, ax              ;Double, car A est tableau word.
             add   bx, -20             ;Offset du début de A
             add   bx, ax              ;Calcule addresse de A[I] e
             pop   bp                  ; la renvoie dans BX.
             ret   2                   ;Enlève paramètre de la pile.
Thunk        endp

OverThunk:   push  offset Thunk        ;Pousse addresse (near) de thunk
             push  bp                  ;Pousse ptr sur BA de A/I pour thunk
             push  ss                  ;Pousse addresse de I sur la pile.
             lea   ax, [bp-22]         ;Partie offset de I.
             push  ax
             call  TestThunk
             mov   sp, bp
```

```
                ret
CallThunk       endp
```

12.3 Passage de paramètres comme paramètres à une autre procédure

Quand une procédure passe un de ses propres paramètres comme paramètre à un autre procédure, certains problèmes se développent qui n'existent pas quand on passe des variables comme paramètres. En effet, dans quelques cas (rares) il n'est pas possible logiquement de passer certains types de paramètre à une autre procédure. Cette section traite des problèmes de passage par une procédure de ses paramètres à une autre procédure.

Les paramètres passés par valeur ne sont pas essentiellement différent des variables locales. Toutes les techniques dans les sections précédentes s'appliquent aux paramètres passés par valeur. Les sections suivantes traitent les cas où la procédure appelante passe un paramètre passé à elle par référence, valeur-résultat, résultat, nom et évaluation-paresseuse.

12.3.1 Passage de paramètres par référence à d'autres procédures

Passer un paramètre par référence jusqu'à une autre procédure est l'endroit où la complexité commence. Considérez (le pseudo) squelette de procédure Pascal suivant:

```
procedure HasRef(var refparm:integer);

            procedure ToProc(???? parm:integer);
            begin
            .
            .
            .
            end;

            begin {HasRef}
            .
            .
            .
            ToProc(refParm);
            .
            .
            .
end;
```

Le “ ???? ” dans la liste de paramètre de **ToProc** indique que nous compléterons le mécanisme approprié de passage de paramètre comme la discussion le demande.

Si **ToProc** attend un paramètre passé par valeur (c.-à-d. ???? est juste une chaîne vide), alors HasRef doit chercher la valeur du paramètre **refparm** et passer cette valeur à **ToProc**. Le code suivant accomplit ceci[110]:

```
            Les    bx, [bp+4]              ;Trouve l'addresse de refparm
            push   es:[bx]                 ;Pousse l'entier pointé par refparm
            call   ToProc
```

Passer un paramètre par référence par référence, valeur-résultat ou paramètre de résultat est facile - copiez juste le paramètre de l'appelant tel-quel sur la pile. C'est-à-dire, si le paramètre **refparm** dans **ToProc** ci-dessus est un paramètre par référence, un paramètre par valeur-résultat, ou un paramètre par résultat, vous utiliseriez la séquence d'appel suivante:

```
            push [bp+6]                    ;Pousse partie segment de refparm.
            push [bp+4]                    ;Pousse partie offset de refparm.
```

[10] Tous les exemples dans cette section présument l'utilisation d'une table lexicale. Si vous utilisez des liens statiques, soyez sûr d'ajuster tous les offsets et le code pour tenir compte du lien statique que l'appelant doit pousser immédiatement avant un appel de procédure.

```
            call ToProc
```

Passer un paramètre par référence par nom est assez facile. Écrivez juste un thunk qui saisit l'adresse du parameter par référence et renvoie cette valeur. Dans l'instance ci-dessus, l'appel à **ToProc** pourrait ressembler à ce qui suit :

```
            jmp    SkipThunk
Thunk0      proc   near
            les    bx, [bp+4]               ;Supposer que BP pointe sur BA d'HasRef.
            ret
Thunk0      endp

SkipThunk:  push   offset Thunk0            ;Addresse du thunk.
            push bp                         ;BA contenant les vars du thunk.
            call ToProc
```

À l'intérieur de ToProc, une référence au paramètre pourrait ressembler à ce qui suit:

```
            push   bp                               ;Sauve notre ptr sur BA.
            mov    bp, [bp+4]                       ;Ptr sur le BA de Parm.
            call   near ptr [bp+6]                  ;Appelle le thunk.
            pop    bp                               ;Récupère notrr ptr sur BA.
            mov    ax, es:[bx]                      ;Accède à la variable.
            .
            .
            .
```

Le passage d'un paramètre par référence par évaluation-paresseuse est très semblable au passage par nom. La seule différence (dans la séquence d'appel de **ToProc**) est que le thunk doit renvoyer la valeur de la variable au lieu de son adresse. Vous pouvez facilement accomplir ceci avec le thunk suivant:

```
Thunk1      proc   near
            push   es
            push   bx
            les    bx, [bp+4]               ;Supposer que BP pointe sur BA d'HasRef.
            mov    ax, es:[bx]              ;Retourne val de refparm dans ax.
            pop    bx
            pop    es
            ret
Thunk1      endp
```

1. 12.3.2 Passage de paramètres par valeur-résultat et résultat comme paramètres

À supposer que vous avez créé une variable locale qui contient la valeur d'un paramètre par valeur-résultat ou par résultat, passer un de ces paramètres à une autre procédure n'est aucunement différent que passer des paramètres par valeur à l'autre code. Une fois qu'une procédure fait une copie locale du paramètre par valeur-résultat ou assigne le stockage pour un paramètre par résultat, vous pouvez traiter cette variable tout comme un paramètre par valeur ou une variable locale en ce qui concerne sa transmission à d'autres procédures.

Naturellement, cela n'a pas de sens d'utiliser la valeur d'un paramètre par résultat avant que vous ayez stocké une valeur dans l'emplacement de stockage local de ce paramètre. Par conséquent, assurez-vous, en passant des paramètres par résultat à d'autres procédures, d'avoir initialisé un paramètre par résultat avant d'utiliser sa valeur.

12.3.3 Passage de paramètres par nom à d'autres procédures

Puisqu'un *thunk* de parameter passé par nom renvoie l'adresse d'un paramètre, passer un paramètre par nom à une autre procédure est très semblable à passer un paramètre par référence

à une autre procédure. Les différences primaires se produisent quand vous transmettez le paramètre comme paramètre par nom.

En passant un paramètre par nom comme paramètre par valeur, vous appelez d'abord le *thunk*, déréférencez l'adresse que retourne le *thunk*, et passez ensuite la valeur à la nouvelle procédure. Le code suivant démontre un tel appel quand le thunk renvoie l'adresse de la variable dans **es:bx** (supposez que le pointeur sur le BA du paramètre passé par nom est à l'adresse **bp+4** et que le pointeur sur le *thunk* est à l'adresse **bp+6**):

```
push   bp                       ;Sauve notre ptr sur le BA.
mov    bp, [bp+4]               ;Ptr sur le BA du Parm.
Call   near ptr [bp+6]          ;Appelle le thunk.
push   word ptr es:[bx]         ;Pousse la valeur du paramètre.
pop    bp                       ;Récupère notre ptr sur le BA.
call   ToProc                   ;Appelle la procédure.
.
.
.
```

Le passage d'un paramètre par nom à une autre procédure par référence est très facile. Il vous suffit de pousser l'adresse que le *thunk* retourne sur la pile. Le code suivant, qui est très semblable au code ci-dessus, le réalise:

```
push   bp                       ;Sauve notre ptr sur le BA.
mov    bp, [bp+4]               ;Ptr sur le BA du Parm.
call   near ptr [bp+6]          ;Appelle le thunk.
pop    bp                       ;Récupère notre ptr sur le BA.
push   es                       ;Pousse la partie seg de l'adrs.
push   bx                       ;Pousse la partie offset de l'adrs.
call   ToProc                   ;Appelle la procédure.
.
.
.
```

Passer un paramètre par nom à une autre procédure comme paramètre passé par nom est très facile; il vous suffit de passer le *thunk* (et les pointeurs associés) à la nouvelle procédure. Le code suivant réalise ceci:

```
push   [bp+6]                   ;Passe l'addresse du Thunk.
push   [bp+4]                   ;Passe l'adrs du BA du Thunk.
call   ToProc
```

Pour passer un paramètre par nom à une autre procédure par évaluation paresseuse, vous devez créer un thunk pour le paramètre d'évaluation-paresseuse qui appelle le *thunk* du paramètre passé par nom, déréférence le pointeur, et puis renvoie cette valeur. L'implémentation vous en est laissée comme projet de programmation.

12.3.4 Passage de paramètres par évaluation-paresseuse comme paramètres

Les paramètres par évaluation-paresseuse se composent typiquement de trois composants: l'adresse d'un *thunk*, un emplacement pour contenir la valeur que le *thunk* retourne, et une variable booléenne qui détermine si la procédure doit appeler le *thunk* pour obtenir la valeur du paramètre ou si elle peut simplement utiliser la valeur précédemment retournée par ce dernier (voyez les exercices dans le chapitre précédent pour voir comment mettre en application des paramètres par évaluation-paresseuse). Quand on passe un paramètre par évaluation-paresseuse à une autre procédure, le code appelant doit d'abord contrôler la variable booléenne pour voir si le champ valeur est valide. Sinon, le code doit d'abord appeler le thunk pour obtenir cette valeur. Si le champ booléen est vrai, le code appelant peut simplement utiliser les données dans le champ valeur. Dans tous les cas, une fois que le champ de valeur a des données valides, passer ces données à une autre procédure n'est en rien différent de passer une variable locale ou un paramètre par valeur à une autre procédure.

12.3.5 Résumé des passages de paramètres

Table 48: Passage de paramètres: comme paramètres à une autre procédure

	Passage par valeur	**Passage par référence**	**Passage par valeur-résultat**	**Passage par Résultat**	**Passage par nom**	**Passage par évaluation-paresseuse**
Valeur	Passer la valeur	Passer l'adresse du paramètre par valeur	Passer l'adresse du paramètre par valeur	Passer l'adresse du paramètre par valeur	Créer un *thunk* qui renvoie l'adresse du paramètre par valeur	Créer un thunk qui renvoie la valeur
Référence	Déréférencer le paramètre et passer la valeur sur laquelle il pointe	Passer l'adresse (valeur du paramètre par référence)	Passer l'adresse (valeur du paramètre par référence)	Passer l'adresse (valeur du paramètre par référence)	Créer un *thunk* qui passe l'adresse (valeur du paramètre par référence)	Créer un *thunk* qui déréférence le paramètre par référence et retourne sa valeur
Valeur-résultat	Passer la valeur locale comme paramètre par valeur	Passer l'adresse de la valeur locale comme paramètre	Passer l'adresse de la valeur locale comme paramètre	Passer l'adresse de la valeur locale comme paramètre	Créer un *thunk* qui retourne l'adresse de la valeur locale du paramètre par valeur-résultat	Créer un *thunk* qui retourne la valeur dans la valeur locale du paramètre par valeur-résultat
Résultat	Passer la valeur locale comme paramètre par valeur	Passer l'adresse de la valeur locale comme paramètre	Passer l'adresse de la valeur locale comme paramètre	Passer l'adresse de la valeur locale comme paramètre	Créer un *thunk* qui retourne l'adresse de la valeur locale du paramètre par résultat	Créer un *thunk* qui retourne la valeur dans la valeur locale du paramètre par résultat
Nom	Appeler le *thunk*, déréférencer le pointeur et passer la valeur à l'adresse que le *thunk* retourne	Appeler le *thunk* et passer l'adresse que le *thunk* retourne comme paramètre	Appeler le *thunk* et passer l'adresse que le *thunk* retourne comme paramètre	Appeler le *thunk* et passer l'adresse que le *thunk* retourne comme paramètre	Passer l'adresse du *thunk* et toutes les autres valeurs associées avec le paramètre par nom	Écrire un *thunk* qui appelle le *thunk* du paramètre par nom, déréférencer l'adresse qu'il retourne et ensuite renvoyer la valeur à cette adresse
Évaluation-Paresseuse	Si nécessaire, appeler le *thunk* pour obtenir la	Si nécessaire, appeler le *thunk* pour obtenir la	Si nécessaire, appeler le *thunk* pour obtenir la	Si nécessaire, appeler le *thunk* pour obtenir la	Si nécessaire, appeler le *thunk* pour obtenir la	Créer un *thunk* qui vérfie le champ booléen du paramètre

	valeur du paramètre par évaluation-paresseuse. Passer la valeur locale comme paramètre par valeur	valeur du paramètre par évaluation-paresseuse. Passer l'adresse de la valeur locale comme paramètre	valeur du paramètre par évaluation-paresseuse. Passer l'adresse de la valeur locale comme paramètre	valeur du paramètre par évaluation-paresseuse. Passer l'adresse de la valeur locale comme paramètre	valeur du paramètre par évaluation-paresseuse. Créer un *thunk* qui retourne l'adresse du champ valeur de l'évaluation paresseuse	par évaluation paresseuse. Il devrait appeler le *thunk.* correspondant si cette variable est fausse. Il devrait mettre le champ booléen à vrai et ensuite renvoyer les données dans le champ valeur

12.4 Passages de Procédures comme Paramètres

Beaucoup de langages de programmation vous permettent de passer un nom de procédure ou de fonction comme paramètre. Ceci permet à l'appelant de passer des actions à exécuter à l'intérieur d'une procédure. L'exemple classique est une procédure **graphe** (plot) qui représente graphiquement une fonction générique de maths passée comme paramètre à **graphe**. Le Pascal standard vous permet de passer des procédures et des fonctions en les déclarant comme suit:

```
procedure DoCall(procedure x);
begin

      x;

end;
```

L'instruction **DoCall(xyz);** appelle **DoCall** qui, à son tour, appelle la procédure **xyz**.

Passer une procédure ou une fonction comme paramètre peut sembler une tâche facile - il suffit de passer l'adresse de la fonction ou de la procédure comme l'exemple suivant le démontre:

```
procedure PassMe;
begin
      Writeln('PassMe was called');
end;

procedure CallPassMe(procedure x);
begin
      x;
end;

begin {main}
      CallPassMe(PassMe);
end.
```

Le code 80x86 pour mettre en application ce qui précède pourraient ressembler à ce qui suit:

```
PassMe      proc near
            print
            byte   "PassMe was called",cr,lf,0
            ret
PassMe      endp

CallPassMe  proc   near
            push   bp
            mov    bp, sp
            call   word ptr [bp+4]
```

```
            pop    bp
            ret 2
CallPassMe  endp

Main        proc   near
            lea    bx, PassMe          ;Passe l'addresse de PassMe à
            push   bx                  ; CallPassMe
            call   CallPassMe
            ExitPgm
Main        endp
```

Pour un exemple aussi simple que celui ci-dessus, cette technique fonctionne très bien. Cependant, cela ne fonctionne pas toujours correctement si **PassMe** doit accéder à des variables non-locales. Le code suivant de Pascal démontre le problème qui pourrait se produire:

```
program main;
      procedure dummy;
      begin end;

      procedure Recurse1(i:integer; procedure x);

            procedure Print;
            begin

                  writeln(i);

            end;

            procedure Recurse2(j:integer; procedure y);
            begin

                  if (j=1) then y
                  else if (j=5) then Recurse1(j-1, Print)
                  else Recurse1(j-1, y);

            end;

      begin {Recurse1}

            Recurse2(i, x);

      end;

begin {Main}

      Recurse1(5,dummy);

end.
```

Ce code produit la séquence d'appel suivante:

Recurse1(5,dummy) → Recurse2(5,dummy) → Recurse1(4,Print) →
Recurse2(4,Print) → Recurse1(3,Print) → Recurse2(3,Print) →
Recurse1(2,Print) → Recurse2(2,Print) → Recurse1(1,Print) →
Recurse2(1,Print) → Print

Print affichera la valeur de la variable **i** de **Recurse1** sur la sortie standard. Cependant, il y a plusieurs blocs d'activation présents sur la pile, ce qui amènera la question évidente, « Quelle copie de **i Print** affiche-t-il ». Sans trop y réfléchir, vous pourriez conclure qu'il devrait afficher la valeur "**1**" puisque **Recurse2** appelle **Print** quand la valeur de **Recurse1** pour **i** est un. Remarquez, cependant, que quand **Recurse2** passe l'adresse de **Print** à **Recurse1**, valeur de **i** est quatre. Le Pascal, comme la plupart des langages structurés par bloc, utilisera la valeur de **i** au moment où **Recurse2** passe l'adresse de **Print** à **Recurse1**. Par conséquent, le code ci-dessus devrait afficher la valeur quatre, pas la valeur un.

Ceci crée un problème d'implémentation difficile. Quoi qu'il en soit, **Print** ne peut tout simplement pas accéder à la table lexicale pour avoir accès à la variable globale **i** - l'entrée de la table lexicale pour **Recurse1** pointe sur la dernière copie du bloc d'activation de **Recurse1**, pas sur l'entrée contenant la valeur quatre qui est qui ce que vous voulez.

La solution la plus courante dans les systèmes utilisant une table lexicale est de faire une copie locale de chaque table lexicale chaque fois qu'on appelle une procédure ou une fonction. En passant une procédure ou une fonction comme paramètre, le code appelant copie la table lexicale avec l'adresse de la procédure ou de la fonction. C'est pourquoi l'instruction Intel **enter** fait une copie de la table lexicale en construisant le bloc d'activation.

Si vous passez des procédures et des fonctions comme paramètres, vous pouvez envisager d'utiliser des liens statiques plutôt qu'une table lexicale. En utilisant un lien statique vous devez seulement passer un pointeur unique (le lien statique) avec l'adresse de la routine. Naturellement, cela demande plus de travail pour accéder à des variables non-locales, mais vous ne devez pas copier la table lexicale à chaque appel, ce qui est très coûteux.

Le code 80x86 suivant fournit l'implémentation du code ci-dessus en utilisant des liens statiques:

```
wp          textequ     <word ptr>
Dummy       proc        near
            ret
Dummy       endp

; PrintIt; (Utilise le nom PrintIt pour éviter des conflits).
;
;     stack:
;
;     bp+4:  lien statique.

PrintIt     proc  near
            push  bp
            mov   bp, sp
            mov   bx, [bp+4]          ;Obtient le lien statique
            mov   ax, ss:[bx-10]      ;Obtient la valeur de i.
            puti
            pop   bp
            ret   2
PrintIt     endp

; Recurse1(i:integer; procedure x);
;
;     stack:
;
;     bp+10: i
;     bp+8:  lien statique de x
;     bp+6:  adresse de x

Recurse1    proc  near
            push  bp
            mov   bp, sp
            push  wp [bp+10]          ;Pousse la valeur de i sur la pile.
            push  wp [bp+8]           ;Pousse le lien statique de x.
            push  wp [bp+6]           ;Pousse l'adresse de x.
            push  bp                  ;Pousse lien statique de Recurse1.
            call  Recurse1
            pop   bp
            ret   6
Recurse1    endp

; Recurse2(i:integer; procedure y);
;
;     stack:
;
;     bp+10: j
;     bp+8:  lien statique de y
;     bp+6:  adresse de y.
;     bp+4:  lien statique de Recurse2.
```

```
Recurse2    proc   near
            push   bp
            mov    bp, sp
            cmp    wp [bp+10], 1       ;Est-ce que j=1?
            jne    TryJeq5
            push   [bp+8]              ;lien statique de y.
            call   wp [bp+6]           ;Appelle y.
            jmp    R2Done
TryJeq5:    cmp    wp [bp+10], 5       ;Est-ce que j=5?
            jne    Call1
            mov    ax, [bp+10]
            dec    ax
            push   ax
            push   [bp+4]              ;Pousse lien statique pour R1.
            lea    ax, PrintIt         ;Pousse l'adresse de PrintIt.
            push   ax
            call   Recurse1
            jmp    R2Done
Call1:      mov    ax, [bp+10]
            dec    ax
            push   ax
            push   [bp+8]              ;Passe aussi le lien
            push   [bp+6]              ; et l'adresse existants.
            call   Recurse1

R2Done:     pop    bp
            ret    6
Recurse1    endp

main        proc
            push   bp
            mov    bp, sp
            mov    ax, 5               ;Pousse le premier paramètre.
            push   ax
            push   bp                  ;Faux lien statique.
            lea    ax, Dummy           ;Pousse l'adresse du faux lien.
            push   ax
            call   Recurse1
            pop    bp
            ExitPgm
main        endp
```

Il y a plusieurs manières d'améliorer ce code. Naturellement, ce programme particulier n'a pas vraiment besoin de maintenir une table lexicale ou une liste statique parce que seul **PrintIt** accède à des variables non-locales; cependant, ignorez ce fait pour l'instant et faites comme s'il en avait besoin. Puisque vous savez que **PrintIt** accède à des variables seulement à un niveau lexical particulier et que le programme appelle **PrintIt** seulement indirectement, vous pouvez passer un pointeur sur le bloc d'activation approprié ; c'est ce que le code ci-dessus fait, bien qu'il maintienne en même temps des liens statiques complets. Les compilateurs doivent toujours supposer le cas le plus mauvais et souvent produisent du code inefficace. Si vous passez en revue vos besoins particuliers, cependant, vous pouvez améliorer l'efficacité de votre code en évitant une grande partie de la surcharge due au maintien de listes statiques ou à la copie de tables lexicales.

Gardez à l'esprit que les *thunks* sont des cas spéciaux de fonctions que vous appelez indirectement. Ils souffrent des mêmes problèmes et inconvénients que des paramètres procédure et fonction en ce qui concerne l'accès à des variables non-locales. Si de telles routines accèdent à des variables non-locales (et les *thunks* le feront presque toujours), alors vous devez faire attention en appelant de telles routines. Heureusement, ils ne causent jamais de récursion indirecte (qui est responsable des problèmes tordus de l'exemple **Recurse1/Recurse2**), aussi vous pouvez utiliser une table lexicale pour accéder à toutes les variables non-locales apparaissant dans le *thunk*.

12.5 Itérateurs

Un itérateur est un croisement entre une structure de contrôle et une fonction. Bien que les langages de haut niveau courants supportent rarement les itérateurs, ils sont présents dans quelques langages de très haut niveau[11111]. Les itérateurs fournissent une combinaison de mécanisme d'appel de l'état machine/fonction qui permet à une fonction de reprendre où elle cessé précédemment à chaque nouvel appel. Ils font également partie d'une structure de contrôle de boucle, fournissant la valeur de la variable de contrôle de boucle à chaque itération.

Pour comprendre ce qu'est un itérateur, considérez la boucle **for** suivante en Pascal:

```
for I := 1 to 10 do <some statement>;
```

En apprenant le Pascal, on vous a probablement enseigné que cette instruction initialise **I** avec un, compare **I** à 10, et exécute l'instruction si **I** est inférieur ou égal à 10. Après exécution de l'instruction, l'instruction **for** incrémente **i** et le compare à 10 de nouveau, répétant ceci jusqu'à ce que **I** soit plus grand que 10.

Certes, cette description est sémantiquement correcte, et en effet, c'est la manière dont la plupart des compilateurs Pascal mettent en application la boucle **for**, mais ceci n'est pas le seul point de vue qui décrit comment celle-ci fonctionne. Supposez, au contraire, que vous deviez traiter le mot réservé "**to**" comme un opérateur. Un opérateur qui attend deux paramètres (un et dix dans ce cas-ci) et renvoie la plage des valeurs à chaque exécution successive. C'est-à-dire, au premier appel, l'opérateur "**to**" renverrait un, au deuxième appel, renverrait deux, etc... Après le dixième appel, l'opérateur "**to**" *échouerait*, ce qui terminerait la boucle. C'est exactement la description d'un itérateur.

En général, un itérateur contrôle une boucle. Les différents langages utilisent des noms différents pour les boucles contrôlées par itérateur, ce texte n'utilisera que le nom *foreach* comme suit:

```
foreach variable in iterator() do
        statements;
endfor;
```

variable est une variable dont le type est compatible avec le type de retour de **iterator**. Un itérateur renvoie deux valeurs: une valeur booléenne *succès* ou *échec* et un résultat de fonction. Tant que l'itérateur renvoie succès, l'instruction **foreach** assigne l'autre valeur de retour à **variable** et exécute **statements**. Si **iterator** renvoie échec, la boucle **foreach** se termine et exécute l'instruction séquentielle suivant le corps de la boucle **foreach**. En cas d'échec, l'instruction **foreach** n'affecte pas la valeur de **variable**.

Les itérateurs sont considérablement plus complexes que les fonctions normales. Un appel de fonction classique comporte deux opérations de base: un appel et un retour. Les invocations d'Itérateurs comportent quatre opérations de base:

1) Appel initial d'itérateur
2) Production d'une valeur
3) Reprise de l'itérateur
4) Arrêt de l'itérateur.

Pour comprendre comment un itérateur fonctionne, considérez le court exemple suivant tiré du langage de programmation Panacea[11112]:

```
Range:iterator(start,stop:integer):integer;
begin range;

     while (start <= stop) do

           yield start;
           start := start + 1;
```

[11] ADA et PL/I supportent des formes très limitées d'itérateurs, bien qu'ils ne supportent pas le type d'itérateurs qu'on trouve dans CLU, SETL, Icon et d'autres langages.

[12] Panacea est un langage de niveau très élevé développé par Randall Hyde pour être utilisé dans des cours de compilation à l'université UC de Riverside.

```
        endwhile;

    end Range;
```

Dans ce le langage de programmation, les appels d'itérateur ne peuvent apparaître que dans l'instruction foreach. A part l'instruction *yeld* ci-dessus, quiconque est familiarisé avec le Pascal ou le C++ devrait pouvoir comprendre la logique de base de cet itérateur.

Un itérateur dans le langage de programmation Panacea peut retourner à son appelant en utilisant un parmi deux mécanismes séparés, il peut *retourner* à l'appelant, en sortant par l'instruction **end Range;** ou il peut *produire (yeld*) une valeur en exécutant l'instruction **yeld**. Un itérateur *réussit* s'il exécute l'instruction **yeld**, il *échoue* s'il retourne simplement à l'appelant. Par conséquent, l'instruction **foreach** exécutera son instruction correspondante seulement si vous quittez un iterateur avec un **yeld.** L'instruction **foreach** se termine si vous retournez simplement de l'itérateur. Dans l'exemple ci-dessus, l'itérateur renvoie les valeurs **start..stop** par l'intermédiaire d'un **yeld** et ensuite l'itérateur se termine. La boucle

```
foreach i in Range(1,10) do
      write(i);
endfor;
```

est comparable à l'instruction Pascal:

```
for I := 1 to 10 do write(i);
```

Quand un programme Panacea exécute la première fois l'instruction **foreach**, il fait un *appel initial* à l'itérateur. L'itérateur fonctionne jusqu'à ce qu'il exécute un **yeld** ou retourne. S'il exécute l'instruction **yeld**, il renvoie la valeur de l'expression suivant le **yeld** comme résultat d'itérateur et renvoie succès. S'il retourne simplement, l'itérateur renvoie échec et aucun résultat d'itérateur. Dans l'exemple en cours, l'appel initial à l'itérateur renvoie succès et la valeur un.

En supposant un retour réussi (comme dans l'exemple en cours), l'instruction **foreach** assigne la valeur de retour d'itérateur à la variable de contrôle de boucle et exécute le corps de boucle de **foreach**. Après exécution du corps de boucle, l'instruction **foreach** appelle de nouveau l'itérateur. Cependant, cette fois l'instruction **foreach** *reprend* l'itérateur au lieu de faire un appel initial*. Une reprise d'itérateur continue avec la première instruction suivant le dernier yeld qu'il a exécuté*. Dans l'exemple **range**, une reprise continuerait l'exécution à l'instruction **start := start + 1;**. A la première reprise, l'itérateur **Range** ajouterait un à **start**, produisant la valeur deux. Deux est inférieur à dix (valeur de **stop**) aussi la boucle **while** se répéterait et l'itérateur produirait la valeur deux. Ce processus se répéterait à plusieurs reprises jusqu'à ce que l'itérateur produise dix. A la reprise après la production de dix, l'itérateur incrémenterait **start** à onze et puis retournerait, au lieu de produire, puisque cette nouvelle valeur n'est pas inférieure ou égale à dix. Quand l'itérateur **range** retourne (échoue), la boucle **foreach** se termine.

12.5.1 Implémentation d'Itérateurs en Utilisant l'Expansion en Ligne

L'implémentation d'un itérateur est plutôt complexe. Pour commencer, considérez une première tentative d'implémentation en assembleur de l'instruction foreach ci-dessus:

```
            push   1                        ;Suppose 286 ou mieux
            push   10                       ; et parms passés sur la pile.
            call   Range_Initial            ;Fait un appel initial à iter.
            jc     Failure                  ;C=0, signifie succès, C=1, échec.
ForLoop:    puti                            ;Suppose résultat est dans AX.
            call   Range_Resume             ;Reprend l'itérateur.
            jnc    ForLoop                  ;Carry à zéro égale succès!
Failure:
```

Bien que ceci ressemble à un projet d'implémentation simple, il y a plusieurs détails à considérer. D'abord, l'appel à **Range_Resume** ci-dessus semble assez simple, mais il n'y a aucune adresse fixe qui corresponde à l'adresse de reprise. Bien que cet exemple **Range** n'a, vraisemblablement,

qu'une seule adresse de reprise, en général vous pouvez avoir autant de instructions yeld que vous voulez dans un itérateur. Par exemple, l'itérateur suivant renvoie les valeurs 1, 2, 3, et 4:

```
OneToFour:iterator:integer;
begin OneToFour;

       yield 1;
       yield 2;
       yield 3;
       yield 4;

end OneToFour;
```

L'appel initial exécuterait l'instruction **yeld 1**. La première reprise exécuterait l'instruction **yeld 2**, la deuxième reprise exécuterait **yeld 3**, etc... De toute évidence, il y a aucune adresse de reprise sur laquelle le code appelant peut compter.

. Il y a deux autres détails à considérer. D'abord, un itérateur est libre d'appeler des procédures et des fonctions[11113]. Si une telle procédure ou fonction exécute l'instruction **yeld,** alors la reprise par l'instruction **foreach** continue l'exécution dans la procédure ou la fonction qui a exécuté le **yeld**. Ensuite, la sémantique d'un itérateur exige que toutes les variables locales et les paramètres maintiennent leurs valeurs jusqu'à ce que l'itérateur se termine. C'est-à-dire, la production (yelding) ne désaffecte pas des variables et des paramètres locaux. De même, aucune adresse de retour laissée sur la pile (par exemple, l'appel à une procédure ou à une fonction qui exécute l'instruction **yeld**) ne doit être perdue quand un morceau de code produit et que l'instruction **foreach** correspondante reprend l'itérateur. En général, ceci signifie que vous ne pouvez pas utiliser la séquence standard d'appel et de retour pour produire ou reprendre avec un itérateur parce que vous devez préserver le contenu de la pile.

Alors qu'il y a plusieurs manières d'implémenter des itérateurs en assembleur, la méthode la plus pratique est peut-être de faire appeler par l'itérateur la boucle contrôlée par celui-ci et faire retourner la boucle à la fonction itérateur. Bien sûr, c'est anti-intuitif. Normalement, on pense à l'itérateur comme fonction que la boucle appelle chaque itération, pas l'inverse. Cependant, étant donné la structure de la pile pendant l'exécution d'un itérateur, l'approche anti-intuitive s'avère être plus facile à mettre en application.

Certains langages de haut niveau supportent les itérateurs exactement de cette manière. Par exemple, le Professional Pascal Compiler pour le PC de Metaware le fait[11114]. Si vous deviez créer une séquence de code comme suit:

```
iterator OneToFour:integer;
begin

       yield 1;
       yield 2;
       yield 3;
       yield 4;

end;
```

et l'appeler dans le programme principal comme suit:

```
for i in OneToFour do writeln(i);
```

le Professional Pascal réarrangerait complètement votre code. Au lieu de transformer l'itérateur en une fonction en assembleur et appeller cette fonction depuis l'intérieur du corps de la boucle for, ce code changerait ce corps en une fonction, développerait l'itérateur en ligne (tout comme une macro) et appellerait la fonction de corps de boucle **for** à chaque **yeld**. C'est-à-dire, le Professional Pascal produirait probablement de l'assembleur qui ressemblerait peu ou prou à ce qui suit:

[13] En Panacea, un itérateur pourrait également appeler d'autres types d'unités de programme, y compris d'autres itérateurs, mais vous pouvez ignorer ceci pour l'instant.

[14] Évidemment, c'est une extension non standard du langage de programmation Pascal fourni avec le Professional Pascal.

```
; La procédure suivante correspond au corps de la boucle for
; avec un seul paramètre (I) correspondant à la variable
; de contrôle de boucle:

ForLoopCode  proc   near
             push   bp
             mov    bp, sp
             mov    ax, [bp+4]          ;Obtient la valeur du contrôle de boucle et
             puti                       ; l'affiche.
             putcr
             pop    bp
             ret    2                   ;Extrait la valeur du ctrl de boucle du stk.
ForLoopCode endp

; Le code suivante serait émis en ligne en rencontrant la
; boucle for dans le programme principal, cela correspond à une expansion
; en ligne de l'itérateur comme s'il était une macro,
; en substituant un appel pour les instructions yield:

             push   1                   ;Sur les processeurs 286 et ultérieurs slmnt.
             call   ForLoopCode
             push   2
             call   ForLoopCode
             push   3
             call   ForLoopCode
             push   4
             call   ForLoopCode
```

Cette méthode pour implémenter des itérateurs est commode et produit du code (rapide) relativement efficace. Cependant, elle souffre de deux inconvénients. D'abord, puisque vous devez développer l'itérateur en ligne partout où vous l'appelez, tout comme une macro, votre programme pourrait grossir si l'itérateur n'est pas court et que vous l'utilisez souvent. En second lieu, cette méthode d'implémentation de l'itérateur cache complètement la logique sous-jacente du code et rend vos programmes en assembleur difficiles à lire et à comprendre.

12.5.2 Implémentation d'Itérateurs avec des blocs de reprise

L'expansion en ligne n'est pas la seule manière d'implémenter des itérateurs. Il y a une autre méthode qui préserve la structure de votre programme aux dépens d'une exécution légèrement plus complexe. Plusieurs langages de niveau élevé, dont Icon et CLU, utilisent cette implémentation.

Pour commencer, vous aurez besoin d'un autre cadre de pile: le *bloc de reprise*. Une bloc de reprise contient deux entrées: une adresse de retour de produit (c'est-à-dire, l'adresse de l'instruction suivant l'instruction **yeld**) et un *lien dynamique*, qui est un pointeur sur le bloc d'activation de l'itérateur. En général, le lien dynamique est juste la valeur dans le registre **bp** lorsque vous exécutez l'instruction **yeld**. Cette version implémente les quatre parties d'un itérateur comme suit:

1) Une instruction d'appel pour l'appel initial de l'itérateur,
2) Une instruction d'appel pour l'instruction **yeld**,
3) Une instruction de retour pour l'opération de reprise et
4) Une instruction de retour pour terminer l'itérateur.

Pour commencer, un itérateur exigera *deux* adresses de retour au lieu de l'adresse de retour unique que vous attendez normalement. La première adresse de retour apparaissant sur la pile est l'adresse de retour terminale. La deuxième adresse de retour est l'endroit où la routine transfère le contrôle lors d'une opération **yeld**. Le code appelant doit pousser ces deux adresses de retour lors de l'invocation initiale de l'itérateur. La pile, à l'entrée initiale dans l'itérateur, devrait ressembler à quelque chose comme la Figure 12.8.

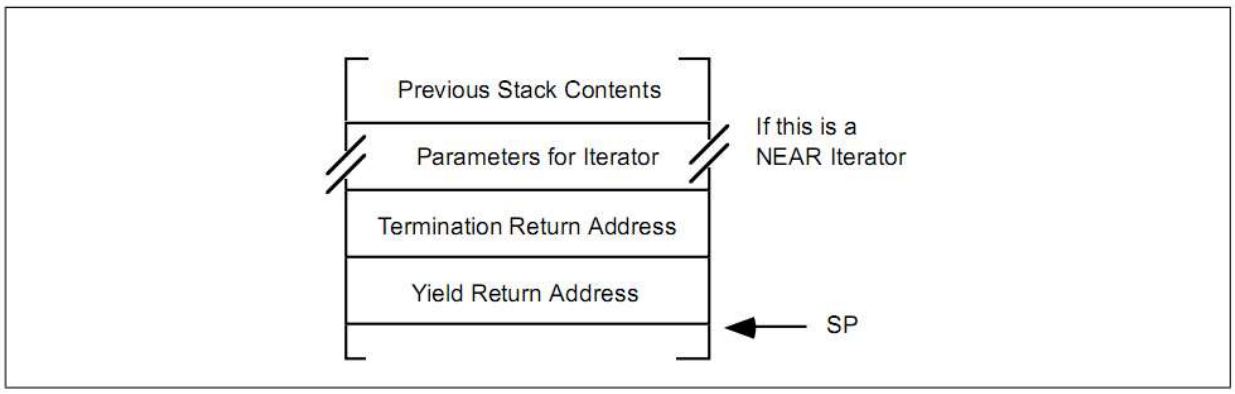

Figure 12.8 Le Bloc d'Activation de l'Itérateur

Comme exemple, considérez l'itérateur **Range** présenté plus tôt. Cet itérateur exige deux paramètres, une valeur de début et une valeur de fin:

```
foreach i in Range(1,10) do writeln(i);.
```

Le code pour faire l'appel initial à l'itérateur Range, produisant une pile comme la pile ci-dessus, pourrait être comme suit:

```
push   1                        ;Pousse la valeur du paramètre start.
push   10                       ;Pousse la valeur du paramètre stop.
push   offset ForDone           ;Pousse l'adresse de terminaison.
call   Range                    ;Pousse l'addresse de retour de yield.
```

ForDone est la première instruction qui suit immédiatement la boucle **foreach**, c.-à-d., l'instruction à exécuter quand l'itérateur renvoie *échec*. Le corps de boucle **foreach** doit commencer par la première instruction suivant l'appel à **Range**. À la fin de la boucle **foreach**, au lieu de sauter au début de la boucle, ou d'appeler de nouveau l'itérateur, ce code devrait seulement exécuter une instruction **ret**. La raison en sera dévoilée dans un moment. Aussi l'implémentation de l'instruction de **foreach** ci-dessus pourrait être la suivante:

```
            push   1                        ; Évidemment, ceci nécessite un
            push   10                       ; processeur 80286 oo ultérieur.
            push   offset ForDone
            call   Range
            mov    bp, [bp]           ; Expliqué un peu plus loin.
            puti
            putcr
            ret
ForDone:
```

D'accord, ça n'a pas du tout l'air d'une boucle. Cependant, en faisant quelques manipulations *majeures* sur la pile, vous verrez que ce code tourne vraiment dans le corps de boucle (puti et putcr) comme prévu.

Considérez maintenant l'itérateur Range lui-même, voici le code pour faire le travail:

```
Range_Start  equ    word ptr <[bp+8]>   ;Adresse du paramètre Start.
Range_Stop   equ    word ptr <[bp+6]>   ;Adresse du paramètre Stop.
Range_Yield  equ    word ptr <[bp+2]>   ;Adresse de retour de Yield.

Range        proc   near
             push   bp
             mov    bp, sp
RangeLoop:   mov    ax, Range_Start     ;Obtient le paramètre start et
             cmp    ax, Range_Stop      ; le compare avec stop.
             ja     RangeDone           ;Termine si start > stop

; OK, construisons le bloc de reprise:
```

```
                push    bp                  ;Sauve le lien dynamique.
                call    Range_Yield         ;Exécute l'operation YIELD.
                pop     bp                  ;Restaure le lien dynamique.
                inc     Range_Start         ;Augmente la valeur de  start
                jmp     RangeLoop           ;Répète jusqu'à ce que start > stop.

RangeDone:      pop     bp                  ;Restaure ancien BP
                add     sp, 2               ;Extrait l'adresse de retour de YIELD
                ret     4                   ;Termine l'itérateur.
Range           endp
```

Bien que cette routine soit plutôt courte, ne laissez pas sa taille vous tromper; elle tout à fait complexe. La meilleure manière de décrire comment cet itérateur fonctionne est de prendre quelques instructions à la fois. Les deux premières instructions sont la séquence standard d'entrée pour une procédure. Lors de l'implémentation de ces deux instructions, la pile ressemble à celle sur la Figure 12.9.

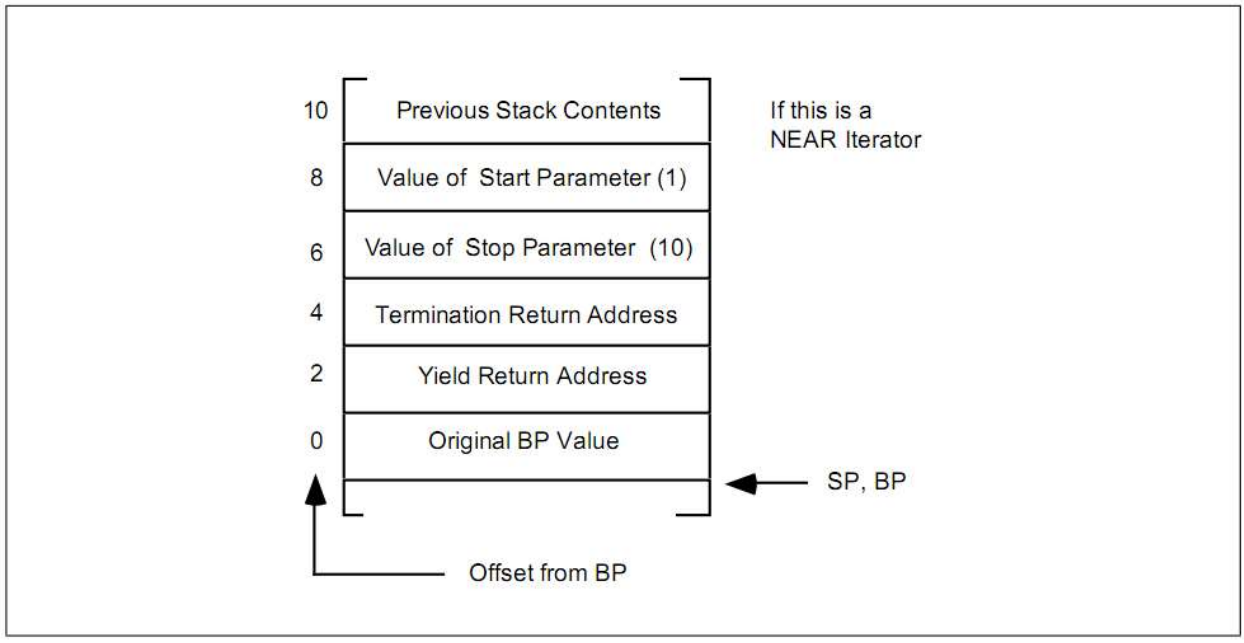

Figure 12.9 Le bloc d'activation de range

Les trois instructions suivantes de l'itérateur **Range**, à l'étiquette **RangeLoop**, implémentent le test d'arrêt de la boucle while. Quand le paramètre **Start** contient une valeur plus grande que le paramètre **Stop**, le contrôle passe à l'étiquette **RangeDone** où le code extrait la valeur de **bp** de la pile, extrait l'adresse de retour de **yeld** de la pile (puisque ce code ne retournera *pas* au corps de la boucle d'itérateur) et puis retourne via l'adresse de retour de terminaison qui est immédiatement au-dessus de l'adresse de retour de yeld sur la pile. L'instruction de retour extrait également les deux paramètres de la pile.

Le vrai travail de l'itérateur se produit dans le corps de la boucle while. Les instructions **push**, **call** et **pop** implémentent l'instruction yeld. Les instructions **push** et **call** établissent le bloc de reprise et ensuite renvoient le contrôle au corps de la boucle **foreach**. L'instruction **call** n'appelle *pas* une routine. En fait, elle met ici la dernière main au bloc de reprise (en stockant l'adresse de reprise de yeld dans le bloc de reprise) et ensuite, elle *renvoie* le contrôle de nouveau au corps de la boucle **foreach** en sautant indirectement via l'adresse de retour de yeld poussée sur la pile par l'appel initial à l'itérateur. Après que l'exécution de cet appel, le bloc de pile ressemble à celui sur la Figure 12.9.

Notez en outre que le registre **ax** contient la valeur de retour pour l'itérateur. Comme avec les fonctions, ax est un bon endroit pour renvoyer le résultat de retour d'itérateur.

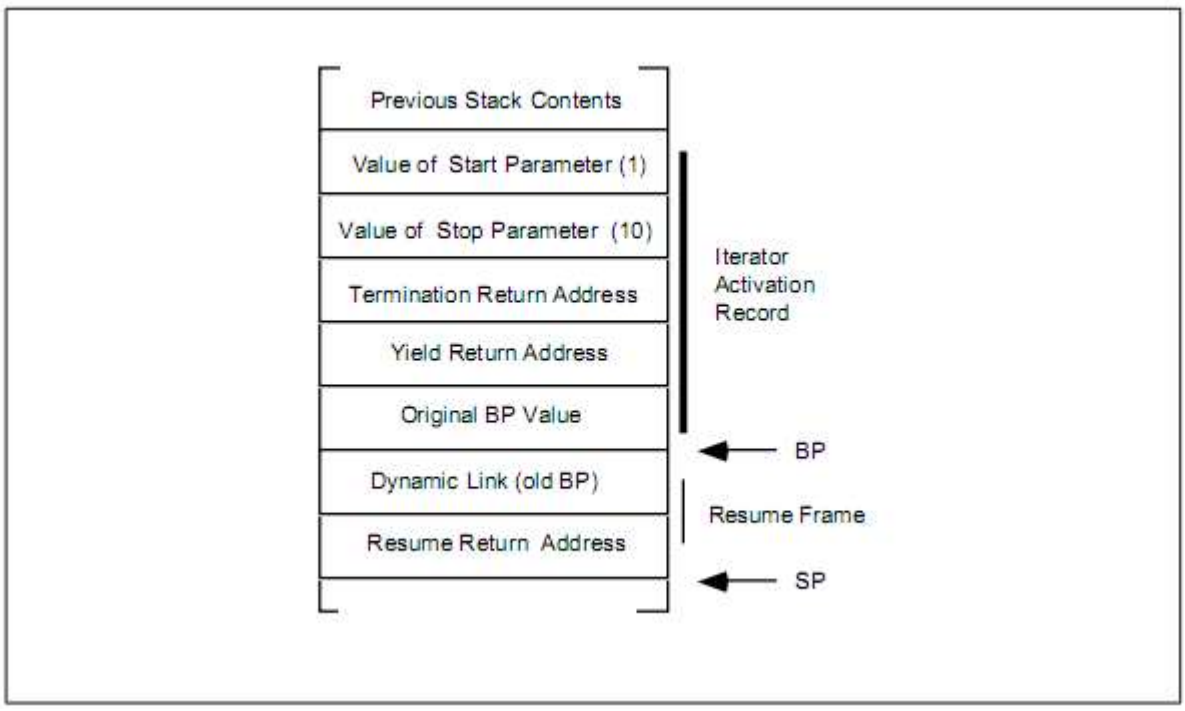

Figure 12.10 Le bloc de reprise de range

Juste après avoir renvoyé le yeld à la boucle **foreach**, le code doit recharger **bp** avec la valeur originale avant l'invocation de l'itérateur. Ceci permet au code appelant d'accéder correctement à des paramètres et à des variables locales dans son propre bloc d'activation plutôt que dans le bloc d'activation de l'itérateur. Puisque **bp** pointe justement sur la valeur originale de **bp** pour le code appelant, exécuter l'instruction **mov bp, [bp]** recharge **bp** comme voulu. Naturellement, dans cet exemple recharger **bp** n'est pas nécessaire parce que le corps de la boucle **foreach** ne met en référence aucun endroit de mémoire à partir du registre **bp**, mais en général vous devrez restaurer **bp**.

À la fin du corps de la boucle **foreach,** l'instruction **ret** reprend l'itérateur. L'instruction **ret** extrait l'adresse de retour de la pile, ce qui renvoie le contrôle de nouveau à l'itérateur juste après l'appel. L'instruction à ce moment, extrait **bp** de la pile, incrémente la variable **start** et répète ensuite la boucle while.

Bien sûr, c'est beaucoup de travail pour créer un morceau de code qui répète simplement une boucle dix fois. Une simple boucle *for* aurait été beaucoup plus facile et tout à fait plus efficace que l'implémentation de **foreach** décrite dans cette section. Cette section a utilisé l'itérateur **Range** parce qu'il était facile de montrer comment les itérateurs fonctionnent en utilisant **Range**, *pas* parce qu'implémenter réellement **Range** comme itérateur est une bonne idée.

12.6 Exemples de programmes

Les exemples de programmes en ce chapitre fournissent deux exemples d'itérateurs. Le premier est un itérateur simple qui traite des caractères dans une chaîne et renvoie les voyelles trouvées dans cette chaîne. Le deuxième est un programme synthétique (c.-à.-d., écrit juste pour illustrer des itérateurs) qui est considérablement plus complexe puisqu'il manipule des liens statiques. Le deuxième exemple de programme démontre également une autre manière de construire le bloc de reprise pour un itérateur. Regardez de près les macros que ce programme utilise. Elles peuvent simplifier l'utilisation des itérateurs dans vos programmes.

12.6.1 Un exemple d'itérateur

L'exemple suivant illustre un itérateur simple. Ce code lit une chaîne fournie par l'utilisateur et puis localise toutes les voyelles (a, e, i, o, u, w, y) sur la ligne et afficher leur index dans la chaîne, la voyelle à cette position et compte les occurrences de chaque voyelle. Ce n'est pas un exemple d'itérateur particulièrement efficace, toutefois il permet d'en démontrer une implémentation et une utilisation.

D'abord, une version pseudo-Pascal du programme:

```
program DoVowels(input,output);
const
      Vowels = ['a', 'e', 'i', 'o', 'u', 'y', 'w',
                  'A', 'E', 'I', 'O', 'U', 'Y', 'W'];
var

ThisVowel : integer;
VowelCnt : array [char] of integer;

iterator GetVowel(s:string) : integer;
var
      CurIndex : integer;
Begin

      for CurIndex := 1 to length(s) do
            if (s [CurIndex] in Vowels) then begin

                  { Si nous avons une voyelle, augmenter le cnt de 1 }
                  Vowels[s[CurIndex]] := Vowels[s[CurIndex]]+1;

                  ( Retourne l'index dans la chaîne de la voyelle en cours }
                  yield CurIndex;

            end;
end;

begin {main}

      { D'abord, initialiser nos compteurs de voyelles }

      VowelCnt ['a'] := 0;
      VowelCnt ['e'] := 0;
      VowelCnt ['i'] := 0;
      VowelCnt ['o'] := 0;
      VowelCnt ['u'] := 0;
      VowelCnt ['w'] := 0;
      VowelCnt ['y'] := 0;
      VowelCnt ['A'] := 0;
      VowelCnt ['E'] := 0;
      VowelCnt ['I'] := 0;
      VowelCnt ['O'] := 0;
      VowelCnt ['U'] := 0;
      VowelCnt ['W'] := 0;
      VowelCnt ['Y'] := 0;

      { Lit et traite la chaîne en entrée}

      Write('Enter a string: ');
      ReadLn(InputStr);
      foreach ThisVowel in GetVowel(InputStr) do
      WriteLn('Vowel ',InputStr [ThisVowel],
      ' at position ', ThisVowel);

      { Affiche les comptes de voyelles }

      WriteLn('# of A''s:',VowelCnt['a'] + VowelCnt['A']);
      WriteLn('# of E''s:',VowelCnt['e'] + VowelCnt['E']);
      WriteLn('# of I''s:',VowelCnt['i'] + VowelCnt['I']);
      WriteLn('# of O''s:',VowelCnt['o'] + VowelCnt['O']);
      WriteLn('# of U''s:',VowelCnt['u'] + VowelCnt['U']);
```

```
      WriteLn('# of W''s:',VowelCnt['w'] + VowelCnt['W']);
      WriteLn('# of Y''s:',VowelCnt['y'] + VowelCnt['Y']);
end.
```

Voici la version assembleur correcte:

```
            .286                    ;Pour l'instr PUSH imm.
            .xlist
            include stdlib.a
            includelib stdlib.lib
            .list

; Quelques equates "futés":

Iterator    textequ     <proc>
endi        textequ     <endp>
wp          textequ     <word ptr>

; Variables globales nécéssaires:

dseg        segment     para public 'data'.

; Comme spécifié dans les instructions de l'UCR StdLib, InputStr doit
; contenir au moins 128 caractères.

InputStr    byte        128 dup (?)

; Notez que l'instruction suivante initialise le
; tableau VowelCnt à zéro, nous évitant d'avoir
; à le faire dans le programme principal.

VowelCnt    word  256 dup (0)

dseg        ends

cseg        segment     para public 'code'
            assume      cs:cseg, ds:dseg

; GetVowel- Cet itérateur cherche la voyelle suivante dans la
;           chaîne entrée et retourne l'index à la valeur
;           comme résultat. A l'entrée, ES:DI pointe
;           sur la chaîne à traiter. Au yield, AX retourne
;           l'index dans la chaîne basé sur zéro de la
;           voyelle courante.
;
; GVYield-  Adresse à appeler en exécutant le yield.
; GVStrPtr- Une variable locale qui pointe sur notre chaîne.

GVYield     textequ     <word ptr [bp+2]>
GVStrPtr    textequ     <dword ptr [bp-4]>

GetVowel    Iterator
            push  bp
            mov   bp, sp

; Crée et initialise GVStrPtr. C'est un pointeur sur le
; caractère suivant à traiter dans la chaîne entrée.

            push  es
            push  di

; Sauve la valeur originale de ES:DI de manière à le restaurer lors de YIELD
; et à la terminaison.

            push  es
            push  di

; OK, voilà le corps principal de l'itérateur. Prendre chaque
```

```
; caractère jusqu'à la fin de la chaîne et voir si c'est
; une voyelle. Si c'est le cas, yield son index. Si
; ce n'est pas une voyelle, avancer au caractère suivant.

GVLoop:     les    di, GVStrPtr        ;Ptr sur car suivant.
            mov    al, es:[di]         ;Obtient ce caractère.
            cmp    al, 0               ;Fin de chaîne ?
            je     GVDone

; L'instruction suivante convertira tous les caractères
; minuscules en majuscules. Cela convertira aussi les autres
; caractères en n'importe quoi, mais on s'en moque puisque
; nous regardons seulement A, E, I, O, U, W, et Y.

            and    al, 5fh

; Voir si ce caractère est une voyelle. Ceci est une operation
; d'appartenance à un ensemble plutôt nulle.

            cmp    al, 'A'
            je     IsAVowel
            cmp    al, 'E'
            je     IsAVowel
            cmp    al, 'I'
            je     IsAVowel
            cmp    al, 'O'
            je     IsAVowel
            cmp    al, 'U'
            je     IsAVowel
            cmp    al, 'W'
            je     IsAVowel
            cmp    al, 'Y'
            jne    NotAVowel

; Si nous avons une voyelle, nous avons besoin d'effectier un yield sur
; l'index dans la chaîne pour cette voyelle. Pour calculer l'index, nous
; restaurons la valeur originale de ES:DI (qui pointe sur le début de la
; chaîne) et soustrayons la position courante (maintenant dans AX)
; de la première position. Ceci produit unindex dans la chaîne basé sur zéro.
; Ce code doit aussi incrémenter l'entrée correspondante dans le tableau
; VowelCnt pour que nous puissions afficher les résultats plus tard. Contrairement
; au code Pascal, nous avons converti les minuscules en majuscules pour que le
; compte des caractères minuscules et majuscules apparaisse dans la case majuscules.

IsAVowel:   push   bx                  ;Incrémente le compte
            mov    ah, 0               ; de la voyelle.
            mov    bx, ax
            shl    bx, 1
            inc    VowelCnt[bx]
            pop    bx

            mov    ax, di
            pop    di                  ;Restaure le DI original
            sub    ax, di              ;Calcule l'index.
            pop    es                  ;Restore le ES original

            push   bp                  ;Sauve notre pointeur de bloc
            call   GVYield             ;Yield à l'applelant
            pop    bp                  ;Restaure notre pointeur de bloc
            push   es                  ;Sauve ES:DI de nouveau
            push   di

; Selon que c'était une voyelle ou non, nous devons maintenant
; passer au caractère suivant dans la chaîne. Incremente
; notre pointeur de chaîne de un et repète de nouveau
; le processus.

NotAVowel:  inc    wp GVStrPtr
            jmp    GVLoop
```

```
; Si nous avons atteint la fin de la chaîne, terminer
; l'itérateur ici. Nous devons restaurer la valeur originale
; de ES:DI, enlever les variables locales, extraire l'adresse
; de YIELD et ensuite retourner à l'adresse de terminaison.

GVDone:     pop    di                  ;Restaure ES:DI
            pop    es
            mov    sp, bp              ;Enlève les vars locales
            add    sp, 2               ;Extrait l'adresse de YIELD
            pop    bp
            ret
GetVowel    endi

Main        proc
            mov    ax, dseg
            mov    ds, ax
            mov    es, ax

            print
            byte   "Enter a chaîne: ",0
            lesi   InputStr
            gets                       ;Lit la ligne entrée.

; Ce qui suit est la boucle foreach. Notez que l'étiquette
; "FOREACH" est présente pour des raisons de documentation seulement.
; En fait, la boucle foreach commence toujours avec la première
; instruction après l'appel à GetVowel.
;
; Une autre note : ce code utilise des index basés sur zéro pour
; la chaîne. La version Pascal utilise des index qui commencent par 1
; pour les chaînes. Aussi les nombres réels affichés seront différents.
; Si vous voulez que les valeurs affichées soient identiques dans les
; deux programmes, enlevez le commentaire pour l'instruction INC ci-dessous.

            push   offset ForDone      ;Adresse de terminaison.
            call   GetVowel            ;Démarre l'itérateur
FOREACH:    mov    bx, ax
            print
            byte   "Vowel ",0
            mov    al, InputStr[bx]
            putc
            print
            byte   " at position ",0
            mov    ax, bx
;           inc    ax
            puti
            putcr
            ret                        ;reprise de l'itérateur.

ForDone:    printf
            byte   "# of A's: %d\n"
            byte   "# of E's: %d\n"
            byte   "# of I's: %d\n"
            byte   "# of O's: %d\n"
            byte   "# of U's: %d\n"
            byte   "# of W's: %d\n"
            byte   "# of Y's: %d\n",0
            dword  VowelCnt + ('A'*2)
            dword  VowelCnt + ('E'*2)
            dword  VowelCnt + ('I'*2)
            dword  VowelCnt + ('O'*2)
            dword  VowelCnt + ('U'*2)
            dword  VowelCnt + ('W'*2)
            dword  VowelCnt + ('Y'*2)

Quit:       ExitPgm                    ;Macro DOS pour quitter le programme.
Main        endp

cseg        ends
```

```
sseg        segment     para stack 'stack'
stk         byte        1024 dup ("stack ")
sseg        ends

zzzzzzseg   segment     para public 'zzzzzz'
LastBytes   db          16 dup (?)
zzzzzzseg   ends
            end   Main
```

12.6.1 Un autre exemple d'itérateur

Un problème avec les exemples d'itéreteur qu'on a vus jusqu'à maintenant est qu'ils n'accèdent à aucune variable globale ou intermédiaire. En outre, ces exemples ne fonctionnent pas si un itérateur est récursif ou appelle d'autres procédures qui produisent la valeur pour la boucle **foreach**. Le problème principal avec ces exemples est que le corps de boucle **foreach** était responsable du rechargement du registre **bp** avec un pointeur sur le bloc d'activation de la procédure de la boucle **foreach**. Malheureusement, le corps de la boucle **foreach** doit présumer que bp pointe couramment sur le bloc d'activation de l'itérateur pour pouvoir obtenir un pointeur sur son propre bloc d'activation à partir de celui-ci. Ce ne sera pas le cas si ce n'est pas celui qui est sur le dessus de la pile.

Pour rectifier ce problème, le code faisant l'opération yeld doit positionner le registre **bp** de sorte qu'il pointe sur le bloc d'activation de la procédure contenant la boucle foreach avant de revenir de nouveau à la boucle. C'est une opération quelque peu complexe. La macro suivante accomplit ceci depuis l'intérieur d'un itérateur :

```
Yield       macro
            mov   dx, [BP+2]          ;Endroit où il faut revenir.
            push  bp                  ;Sauve le lien de l'Iterateur
            mov   bp, [bp]            ;Obtient ptr sur le B.A. de l'appelant

            call  dx                  ;Pousse l'adresse de reprise et rtn.
            pop   bp                  ;Restaure ptr sur notre B.A.
endm
```

Notez un effet secondaire malheureux de ce code: il modifie le registre **dx**. Par conséquent, l'itérateur ne le préserve pas travers un appel à la fonction d'itérateur.

Le macro ci-dessus suppose que le registre **bp** pointe sur le bloc d'activation de l'itérateur. S'il ne le fait pas, alors vous devez exécuter quelques instructions supplémentaires pour suivre les liens statiques à reculons jusqu' au bloc d'activation de l'itérateur pour obtenir l'adresse du bloc d'activation de la procédure de la boucle **foreach**.

```
; ITERS.ASM
;
; Correspond vaguement à l'exemple dans le
; texte "Concepts de langage de programmation" de Ghezzi et Jazayeri
;
; Randall Hyde
;
;
; Ce programme démontre une implémentation de:
;
; l: = 0;
; foreach i in range(1,3) do
;    foreach j in iter2() do
;           writeln(i, ', ', j, ', ', l):
;
;
; iterator range(start, stop):integer;
; begin
;
;    while start < = stop do begin
;
```

```
  ;          yeld start;
  ;          start: = start+1;
  ; end;
  ; end;
  ;
  ; iterator iter2:integer;
  ; var k:integer;
  ; begin
  ;
  ;   foreach k in iter3 do
  ;          yeld k;
  ; end;
  ;
  ; iterator iter3:integer;
  ; begin
  ;
  ;   l: = l + 1;
  ;   yeld 1;
  ;   l: = l + 1;
  ;   yeld 2;
  ;   l: = l + 1;
  ;   yeld 0;
  ;   end;
  ;
  ;
  ; Ce code affiche :
  ;
  ;   1. 1, 1
  ;   1. 2, 2
  ;   1. 0, 3
  ;   2. 1, 4
  ;   2. 2, 5
  ;   2. 0, 6
  ;   3. 1, 7
  ;   3. 2, 8
  ;   3. 0, 9


              .xlist
              include      stdlib.a
              include      libstdlib.lib
              .list

              .286         ;Permet modes d'adrs supplémentaires.

dseg          segment      para stack 'data'

; Mettez la pile dans le segment de données pour pouvoir utiliser le modèle
; de mémoire small pour simplifier l'adressage :

stk           byte         1024 dup ('stack')
EndStk        word         0

dseg ends

cseg          segment      para public 'code'
              assume       cs:cseg, ds:dseg, ss:dseg

; Voici la structure d'un bloc de reprise. Notez que cette structure n'est pas
; réellement utilisé dans ce code. Elle est fournie seulement pour vous montrer
; quelles données sont sur la pile quand le yeld établit un bloc de reprise.

RsmFrm              struct
ResumeAdrs          word   ?
ItérateurLink       word   ?
RsmFrm              ends

; La macro suivante établit un bloc de reprise et les retours à  l'appelant
; d'un itérateur. Elle suppose que l'itérateur et quiconque a appelé
; l'itérateur ont le bloc d'activation standard défini ci-dessus et que nous
```

```
; construisons le bloc de reprise standard décrit ci-dessus
;
; Ce code efface le registre DX. Celui qui appelle l'itérateur ne peut pas
; être sûr que DX soit préservé, de même, l'itérateur ne peut pas être sûr que DX
; soit préservé à travers un yeld. Vraisemblablement, l'itérateur renvoie sa
; valeur dans ax.
;
ActRec              struct
DynamicLink         word   ?            ;Valeur sauvée de BP.
YieldAdrs           word   ?            ;Adrs de Retour pour la proc.
StaticLink          word   ?            ;Lien statique pour la proc.
ActRec              ends

AR                  equ    [bp].ActRec

Yield       macro
            mov    dx, AR.YieldAdrs    ;Endroit où il faut revenir.
            push   bp                  ;Sauve le lien de l'Itérateur
            mov    bp, AR.DynamicLink  ;Obtient ptr sur le bloc activation
                                       ; de l'appelant
            call   dx                  ;Pousse l'adresse de reprise et rtn.
            pop    bp                  ;Restaure ptr sur notre B.A..
            endm

; Range(start, stop) - Yields de start à stop et puis échoue.

; La structure suivante definit le bloc d'activation pour Range:

rngAR       struct
DynamicLink word   ?                   ;Valeur sauvée de BP.
YieldAdrs   word   ?                   ;Adrs Retour pour proc.
StaticLink  word   ?                   ;Lien Statique pour proc.
FailAdrs    word   ?                   ;Va ici quand on échoue
Stop        word   ?                   ;Paramètre Stop
Start       word   ?                   ;Paramètre Start
rngAR       ends

rAR         equ    [bp].rngAR

Range       proc
            push   bp
            mov    bp, sp

; While start <= stop, yield start:

WhlStartLEStop:    mov    ax, rAR.Start       ;Met aussi valeur de retour
                   cmp    ax, rAR.Stop        ; dans AX.
                   jnle   RangeFail

                   yield

                   inc    rAR.Start
                   jmp    WhlStartLEStop

RangeFail:         pop    bp                  ;Restaure le Lien Dynamique.
                   add    sp, 4               ;Saute adrs de ret et L.S.
                   ret    4                   ;Retourne via l'adresse d'échec.
Range              endp

; Iter2- Just calls iter3() et retourne toute valeur qu'il génère.
;
; Notez : Puisque iter2 and iter3 sont au même niveau lexical, le lien statique
; passé à iter3 doit être le même que le lien statique passé à iter2.
; C'est pourquoi l'instruction "push [bp]" apparait en-dessous (contrairement à
; l'instruction "push bp" qui apparait dans les appels à Range et iter2).
; Gardez à l'esprit, Range et iter2 ne sont appelés que depuis main et bp contient
; le lien statique à ce moment. Ceci n'est plus vrai quand iter2 appelle iter3.

iter2       proc
```

```
            push   bp

            mov    bp, sp
            push   offset i3Fail       ;Adresse d'échec.
            push   [bp]                ;le lien statique est un lien sur main.
            call   iter3
            yield                      ;valeur de retour retournée par iter3
            ret                        ;Reprise d'Iter3.
i3Fail:     pop    bp                  ;Restaure le lien dynamique.
            add    sp, 4               ;Saute adresse de retour & L.S.
            ret                        ;Retourne via adresse d'échec.
iter2       endp

; Iter3() yields les valeurs 1, 2 et 0:

iter3       proc
            push   bp
            mov    bp, sp

            mov    bx, AR.StaticLink   ;Fait pointer BX sur le B.A. de main.
            inc    word ptr [bx-6]     ;Incrémente L dans main.
            mov    ax, 1
            yield
            mov    bx, AR.StaticLink
            inc    word ptr [bx-6]
            mov    ax, 2
            yield
            mov    bx, AR.StaticLink
            inc    word ptr [bx-6]
            mov    ax, 0
            yield
            pop    bp                  ;Restaure le Lien Dynamique.
            add    sp, 4               ;Saute adresse de  retour & L.S.
            ret                        ;Retourne via adresse d'échec.
iter3       endp

; Les variables locales de main sont allouées sur la pile de manière à justifier
; l'utilisation des liens statiques.

i           equ    [bp-2]
j           equ    [bp-4]
l           equ    [bp-6]

Main        proc
            mov    ax, dseg
            mov    ds, ax
            mov    es, ax
            mov    ss, ax
            mov    sp, offset EndStk

; Alloue stockage pour i, j et l sur la pile:

            mov    bp, sp
            sub    sp, 6

            meminit

            mov    word ptr l, 0       ;Initialise l.

; foreach i in range(1,3) do:

            push   1                   ;Paramètres.
            push   3
            push   offset iFail        ;Adresse d'échec.
            push   bp                  ;Le lien statique pointe sur notre BA.
            call   Range

; Yield de range vient ici. L'étiquette est pour votre usage.

RangeYield: mov    i, ax               ;Sauve la valeur du contrôle de boucle.
```

```
; foreach j in iter2 do:

            push   offset jfail        ;Adresse d'échec.
            push   bp                  ;Le lien statique pointe sur notre BA.
            call   iter2

; Yield de iter2 vient ici:

iter2Yield: mov    j, ax

            mov    ax, i
            puti
            print
            byte   ", ",0
            mov    ax, j
            puti
            print
            byte   ", ",0
            mov    ax, l
            puti
            putcr

; Redémarre iter2:

            ret                        ;Reprend l'itérateur.

; Redémarre Range ici bas.

jFail:      ret                        ;Reprend l'itérateur.

; Fini!

iFail:      print
            byte   cr,lf,"All Done!",cr,lf,0

Quit:       ExitPgm                    ;Macro DOS pour quitter le programme.
Main        endp

cseg        ends

; zzzzzzseg doit être le dernier segment qui est chargé mémoire!
; Ici commence le tas (heap).

zzzzzzseg   segment      para public 'zzzzzz'
LastBytes   db           16 dup (?)
zzzzzzseg   ends
            end          Main
```

12.7 Exercices de laboratoire

Les exercices de laboratoire de ce chapitre consistent en trois composants. Dans le premier vous expérimenterez avec un ensemble d'itérateurs assez complexe. Dans le deuxième vous apprendrez comment les instructions du 80286 **enter** et **leave** fonctionnent. Dans le troisième, vous ferez quelques preuves sur les mécanismes de passage de paramètres.

12.7.1 Exercices sur les iterateurs

Dans cet exercice de laboratoire vous travaillerez avec un programme (Ex12_1.asm sur le CD-ROM d'accompagnement) utilisant quatre itérateurs. Les trois premiers exécutent quelques calculs assez simples, le quatrième retourne (successivement) des pointeurs sur le code des trois premiers itérateurs que le programme principal peut utiliser pour appeler ces itérateurs.

Pour votre rapport de laboratoire : étudiez le code suivant et expliquez comment il fonctionne. Lancez-le et expliquez ses sorties. Compilez le programme avec l'option "/Zi", puis depuis Code-View, placez un point d'arrêt sur la première instruction des quatre itérateurs. Exécutez-le jusqu'à ces points d'arrêt et affichez la mémoire commençant à la valeur du pointeur de pile en cours (ss:sp). Décrivez la signification des données sur la pile à chaque point d'arrêt. En outre, placez un point d'arrêt l'instruction **"call ax"**. Tracez dans la routine sur laquelle pointe ax à chaque point d'arrêt et décrivez quelle routine cette instruction appelle. Combien de fois cette instruction s'exécute-t-elle ?

```
; EX12_1.asm
;
; Programme destiné à supporter l'exercice de laboratoire du Chapitre 12.
;
; Ce programme combine des iterateurs, passant les paramètres comme paramètres,
; et les paramètres proceduraux tous dans le même programme.
;
;
; Ce programme implémente les iterateurs suivants(exemples écrits en panacea):
;
; programme EX12_1;
;
; fib:iterator(n:integer):integer;
; var
;      CurIndex:integer;
;      Fn1: integer;
;      Fn2: integer;
; endvar;
; begin fib;
;.
;      yield 1; (* Always have at least n=0 *)
;      if (n <> 0) then
;
;            yield 1; (* Have at least n=1 at this point *)
;
;            Fn1 := 1;
;            Fn2 := 1;
;            foreach CurIndex in 2..n do
;
;                  yield Fn1+Fn2;
;                  Fn2 = Fn1;
;                  Fn1 = CurIndex;
;
;            endfor;
;      endif;
;
; end fib;
;
;
;
; UpDown:iterator(n:integer):integer;
; var
;      CurIndex:integer;
; endvar;
; begin UpDown;
;
;      foreach CurIndex in 0..n do
;
;            yield CurIndex;
;
;      endfor;
;      foreach CurIndex in n-1..0 do
;
;            yield CurIndex;
;
;      endfor;
;
; end UpDown;
;
```

```
;
;
; SumToN:iterator(n:integer):integer;
; var
;      CurIndex:integer;
;      Sum: integer;
; endvar;
; begin SumToN;
;
;      Sum := 0;
;      foreach CurIndex in 0..n do
;
;            Sum := Sum + CurIndex;
;            yield Sum;
;
;      endfor;
;
; end SumToN;
;
;
; MultiIter retourne un pointeur sur un iterateur qui accepte
; un paramètre entier unique.
.
;
; MultiIter: iterator: [iterator(n:integer)];
; begin MultiIter;
;
;      yield @Fib;(* Retourne pointeurs sur les trois iterateurs ci-dessus *)
;      yield @UpDown;(* comme resultat ce cet iterateur.*)
;      yield @SumToN;
;
; end MultiIter;.
;
;
; var
;      i:integer;
;      n:integer;
;      iter:[iterator(n:integer)];
; endvar;
;
; begin EX12_1;
;
;      (* La boucle for suivante se répète six fois, passant son index de boucle *)
;      (* comme paramètre aux paramètres Fib, UpDown et SumToN.*)
;
;      foreach n in 0..5 do
;
;
;      (* L'itérateur suivant (un peu bizarres) se promène à travers *)
;      (* chacun des trois itérateurs: Fib, UpDown et SumToN. Il*)
;      (* retourne un pointeur comme valeur d'itérateur. La boucle *)
;      (* foreach la plus interne utilise ce pointeur pour appeler *)
;      (* l'itérateur approprié. *)
;
;            foreach iter in MultiIter do
;
;            (* OK, cette boucle for invoque tout ce que l'itérateur se trouvait*)
;            (* être retourné par l'iterator MultiIter ci-dessus. *)
;
;                  foreach i in [MultiIter](n) do
;
;                        write(i:3);
;
;                  endfor;
;                  writeln;
;
;            endfor;
;            writeln;
;
;      endfor;
```

```
;
; end EX12_1;

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

                .286                        ;Permet modes adrs suppl.

wp              textequ         <word ptr>
ofs             textequ         <offset>

dseg            segment         para public 'code'
dseg            ends

cseg            segment         para public 'code'
                assume          cs:cseg, ss:sseg

; La macro suivante construit un bloc de reprise et les retours à l'appelant
; d'un itérateur. Elle présume que l'itérateur et quiconque a appelé
; l'itérateur ont le bloc d'activation standard défini ci-dessus et que nous
; construisons le bloc de reprise standard décrit ci-dessus.
;
; Ce code efface le registre DX. Quiconque appelle l'itérateur ne peut pas
; compter sur DX, de même, l'itérateur ne peut pas compter sur DX
; après un yield. On suppose que l'itérateur retourne sa
; valeur dans AX.

Yield           macro
                mov     dx, [BP+2]          ;Endroit où on récupère le yield.
                push    bp                  ;Sauve le lien d'Iterator
                mov     bp, [bp]            ;Obtient ptr sur le B.A. de l'appelant.
                call    dx                  ;Pousse adresse de reprise et retourne.
                pop     bp                  ;Restaure ptr sur notre B.A.
endm

; Fib(n) -      Yield la séquence des nombres de fibonacci de F(0).à.F(n).
;               La séquence de fibonacci est définie comme:
;
;               F(0) et F(1) = 1.
;               F(n) = F(n-1) + F(n-2) for n > 1.

; La structure suivante définit le bloc d'activation pour Fib

CurIndex        textequ         <[bp-6]>    ;Valeur de la séquence en cours.
Fn1             textequ         <[bp-4]>    ;Valeur de F(n-1).
Fn2             textequ         <[bp-2]>    ;Valeur de F(n-2).
DynamicLink     textequ         <[bp]>      ;Valeur sauvée de BP.
YieldAdrs       textequ         <[bp+2]>    ;Retourne Adrs pour la proc.
FailAdrs        textequ         <[bp+4]>    ;Va ici quand on échoue
n               textequ         <[bp+6]>    ;Le paramètre initial

Fib             proc
                push    bp
                mov     bp, sp
                sub     sp, 6               ;Fait de la place pour les variables locales.

; On commenceras à récupérer des valeurs de yield à partir de F(0).
; Puisque F(0) et F(1) sont des cas speciaux, on yield leur valeurs ici.

                mov     ax, 1               ;Yield F(0) (on retourne toujours au moins
                yield                       ; F(0)).

                cmp     wp n, 1             ;Vérifie si l'utilisateur l'a appelé avec n=0.
                jb      FailFib
                mov     ax, 1
                yield

; OK, n >=1 alors on doit rentrer dans une boucle pour traiter les valeurs restantes.
```

```
; D'abord, on commence par initialiser Fn1 et Fn2 comme approprié.

            mov   wp Fn1, 1
            mov   wp Fn2, 1
            mov   wp CurIndex, 2

WhlLp:      mov   ax, CurIndex      ;Voir si CurIndex > n.
            cmp   ax, n
            ja    FailFib

            push  Fn1
            mov   ax, Fn1
            add   ax, Fn2
            pop   Fn2               ;Fn1 devient la nouvelle valeur de Fn2.
            mov   Fn1, ax           ;La val en cours devient la nvelle val de Fn1.
            yield                   ;Yield la valeur en cours.

            inc   wp CurIndex
            jmp   WhlLp

FailFib:    mov   sp, bp            ;Désalloue les vars locales.
            pop   bp                ;Restaure Lien Dynamique.
            add   sp, 2             ;Saute adrs de ret.
            ret   2                 ;Retourne via adresse d'échec.
Fib         endp

; UpDown-   Cette fonction yields la sequence 0, 1, 2, ..., n, n-1,
;           n-2, ..., 1, 0.

i           textequ     <[bp-2]>    ;Valeur de F(n-2).

UpDown      proc
            push  bp
            mov   bp, sp
            sub   sp, 2             ;Fait de la place pour i.

            mov   wp i, 0           ;Initialise notre variable index (i).
UptoN:      mov   ax, i
            cmp   ax, n
            jae   GoDown

            yield

            inc   wp i
            jmp   UpToN

GoDown:     mov   ax, i
            yield
            mov   ax, i
            cmp   ax, 0
            je    UpDownDone
            dec   wp i
            jmp   GoDown

UpDownDone: mov   sp, bp            ;Désalloue vars locales.
            pop   bp                ;Restaure Lien Dynamique.
            add   sp, 2             ;Saute adrs ret.
            ret   2                 ;Return via adresse d'échec.
UpDown      endp

; SumToN(n)- Cet itérateur retourne 1, 2, 3, 6, 10, ... somme(n) où
;           somme(n) = 1+2+3+4+...+n (p.ex., n(n+1)/2);

j           textequ     <[bp-2]>
k           textequ     <[bp-4]>

SumToN      proc
            push  bp
            mov   bp, sp
            sub   sp, 4             ;Fait de la place pour j et k.
```

```
            mov   wp j, 0           ;Initialise notre variable index (j).
            mov   wp k, 0           ;Initialise notre somme (k).

SumLp:      mov   ax, j
            cmp   ax, n
            ja    SumDone

            add   ax, k
            mov   k, ax

            yield

            inc   wp j
            jmp   SumLp

SumDone:    mov   sp, bp            ;Désalloue vars locales.
            pop   bp                ;Restaure Lien Dynamique.
            add   sp, 2             ;Saute adrs ret.
            ret   2                 ;Retourne via adresse d'échec

SumToN      endp

; MultiIter- Cet itérateur retourne un pointeur sur chacun des itérateurs ci-dessus.

MultiIter   proc
            push  bp
            mov   bp, sp

            mov   ax, ofs Fib
            yield
            mov   ax, ofs UpDown
            yield
            mov   ax, ofs SumToN
            yield

            pop   bp
            add   sp, 2
            ret
MultiIter   endp

Main        proc
            mov   ax, dseg
            mov   ds, ax
            mov   es, ax
            meminit

; foreach bx in 0..5 do

            mov   bx, 0             ;Variable de contrôle pour boucle externe.
WhlBXle5:

; foreach ax in MultiIter do

            push  ofs MultiDone     ;Adresse d'échec.
            call  MultiIter         ;Obtient l'itérateur à appeler.

; foreach i in [ax](bx) do

            push  bx                ;Pousse "n" (bx) sur la pile.
            push  ofs IterDone      ;Adresse d'échec
            call  ax                ;Appelle l'iterator sur lequel pointe
                                    ; la valeur de retour de MultiIter.
;
; write(ax:3);

            mov   cx, 3
            putisize
            ret
```

```
; endfor, writeln;

IterDone:    putcr                    ;Writeln;
             Ret

; endfor, writeln;

MultiDone:   putcr
             inc    bx
             cmp    bx, 5
             jbe    WhlBXle5

; endfor

Quit:        ExitPgm                  ;Macro DOS pour quitter lr programme

Main         endp

cseg         ends

sseg         segment      para stack 'stack'
stk          word         1024 dup (0)
sseg         ends

zzzzzzseg    segment      para public 'zzzzzz'
LastBytes    db           16 dup (?)
zzzzzzseg    ends
             end          Main
```

12.7.2 Les instructions enter et leave du 80x86

Le code suivant (Ex12_2.asm sur le CD-ROM d'accompagnement) utilise les instructions **enter** et **leave** du 80x86 pour maintenir une table lexicale dans un programme structuré par blocs. Compilez ce programme avec l'option "/Zi" et chargez-le dans CodeView. Placez les points d'arrêt sur les appels des procédures **Lex1**, **Lex2**, **Lex3** et **Lex4**. Exécutez le programme et quand vous rencontrez un point d'arrêt, utilisez la touche F8 pour tracer dans chaque procédure. Tracez au-delà de l'instruction **enter** (jusqu'au **nop** suivant). Notez les valeurs des registres **bp** et **sp** avant.and.après l'exécution de l'instruction enter.

Pour votre rapport de laboratoire : expliquez les valeurs dans les registres **bp** et **sp** après l'exécution de chaque instruction **enter**. Affichez la mémoire de ss:sp jusqu'à environ ss:sp+32 en utilisant une fenêtre de mémoire ou la commande **dw** dans la fenêtre de commande. Décrivez le contenu de la pile après l'exécution de chaque instruction **enter**.

Après avoir fini d'exécuter l'instruction **enter** de la procédure **Lex4**, placez un point d'arrêt sur chacune des instructions **leave**. Exécutez le programme à pleine vitesse (en utilisant la touche F5) jusqu'à ce que vous ayez rencontré chacune de ces instructions **leave**. Notez les valeurs des registres **bp** et **sp** avant.and.après l'exécution de chaque instruction **leave. Pour votre rapport de laboratoire :** incluez ces valeurs de **bp/sp** dans votre rapport de laboratoire et expliquez-les.

```
; EX12_2.asm
;
; Programme pour demontrer les instructions ENTER et LEAVE du Chapitre 12.
;
; Ce programme simule le code Pascal suivant:
;
; program EnterLeave;
; var i:integer;
;
;      procedure Lex1;
;      var j:integer;
;
;             procedure Lex2;
;             var k:integer;
```

```
;
;                   procedure Lex3;
;                   var m:integer;
;
;                         procedure Lex4;
;                         var n:integer;
;                         begin
;
;                               writeln('Lex4');
;                               for i:= 0 to 3 do
;                                     for j:= 0 to 2 do
;                                           write('(',i,',',j,') ');
;                                           writeln;
;                               for k:= 1 downto 0 do
;                                     for m:= 1 downto 0 do
;                                           for n := 0 to 1 do
;                                                 write('(',m,',',k,',',n,') ');
;                                                 writeln;
;                         end;
;
;                   begin {Lex3}
;
;                         writeln('Lex3');
;                         for i := 0 to 1 do
;                               for j := 0 to 1 do
;                                     for k := 0 to 1 do
;                                           for m := 0 to 1 do
;                                                 writeln(i,j,k,m);
;
;                         Lex4;
;
;                   end; {Lex3}
;
;     begin {Lex2}
;
;           writeln('Lex2');
;           for i := 1 downto 0 do
;                 for j := 0 to 1 do
;                       for k := 1 downto 0 do
;                             write(i,j,k,' ');
;           writeln;
;
;           Lex3;
;
;     end; {Lex2}
;
;     begin {Lex1}
;
;           writeln('Lex1');
;           Lex2;
;
;     end; {Lex1}
;
; begin {Main (lex0)}
;
;     writeln('Main Program');
;     Lex1;
;
; end.
            .xlist
            include     stdlib.a
            includelib  stdlib.lib
            .list

            .286                    ;Permet ENTER & LEAVE.

; Equates communs à toutes les procédures:

wp          textequ     <word ptr>
disp1       textequ     <word ptr [bp-2]>
```

```
disp2        textequ      <word ptr [bp-4]>
disp3        textequ      <word ptr [bp-6]>

; Note: le segment data et le segment de pile sont le même et unique dans ce
; programme. Ceci pour permettre l'utilisation du mode d'adressage [bx] quand
; on référence des variables locales et intermediaires sans qu'on ait à utiliser
; un préfixe de segment de pile.

sseg         segment      para stack 'stack'

i            word         ?            ;Variable du programme Main.
stk          word         2046 dup (0)

sseg         ends

cseg         segment      para public 'code'
             assume cs:cseg, ds:sseg, ss:sseg

; Le bloc d'activation de Main ressemble à ceci:
;.
; | adresse retour |<- SP, BP
; |----------------|

Main         proc
             mov    ax, ss             ;Rend SS=DS pour simplifier addressage
             mov    ds, ax             ; (on n'aura pas besoin de coller "SS:"
             mov    es, ax             ; devant les modes d'adressage comme
                                       ; "[bx]").

             print
             byte   "Main Program",cr,lf,0
             call   Lex1
Quit:        ExitPgm                   ;Macro DOS pour quitter le programme.
Main         endp

; Le bloc d'activation de Lex1 ressemble à ceci:
;
; | adresse retour |
; |----------------|
; | Lien Dynamique | <- BP
; |----------------|
; | Ptr sur BA Lex1| | Table lexicale
; |----------------|
; |  Var Locale J  | <- SP (BP-4)
; |----------------|

Lex1_J       textequ      <word ptr [bx-4]>

Lex1         proc   near
             Enter   2, 1              ;Une variable locale à 2 bytes au niveau lex 1.

             nop                       ;Instruction espace pour tracer pas à pas
             print
             byte   "Lex1",cr,lf,0
             call   Lex2
             leave
             ret
Lex1         endp




; Le bloc d'activation de Lex2 ressemble à ceci:
;
; | adresse retour |
; |----------------|
; | Lien Dynamique | <- BP
; |----------------|
```

```
; | Ptr sur BA Lex1| |
; |----------------| | Table lexicale
; | Ptr sur BA Lex2| |
; |----------------|
; |  Var Locale K  | <- SP (BP-6)
; |----------------|
;
;     writeln('Lex2');
;     for i := 1 downto 0 do
;           for j := 0 to 1 do
;                 for k := 1 downto 0 do
;                       write(i,j,k,' ');
;     writeln;
;
;     Lex3;

Lex2_k      textequ     <word ptr [bx-6]>
k           textequ     <word ptr [bp-6]>

Lex2        proc  near
            enter 2, 2              ;Une variable locale à 2 bytes au niveau lex 2.

            nop                     ;Instruction espace pour tracer pas à pas

            print
            byte  "Lex2",cr,lf,0

            mov   i, 1

ForLpI:     mov   bx, disp1         ;"J" est au niveau lex un.
            mov   Lex1_J, 0
ForLpJ:     mov   k, 1              ;"K" est locale.

ForLpK:     mov   ax, i
            puti
            mov   bx, disp1
            mov   ax, Lex1_J
            puti
            mov   ax, k
            puti
            mov   al, ' '
            putc

            dec   k                 ;Décrémente de 1->0 et quitte
            jns   ForLpK            ; si on atteint -1.

            mov   bx, disp1
            inc   Lex1_J
            cmp   Lex1_J, 2
            jb    ForLpJ

            dec   i
            jns   ForLpI

            putcr
            call Lex3

            leave
            ret
Lex2        endp

; Le bloc d'activation de Lex3 ressemble à ceci:
;
; | adresse retour |
; |----------------|
; | Lien Dynamique | <- BP
; |----------------|
; | Ptr sur BA Lex1| |
; |----------------| |
; | Ptr sur BA Lex2| | Table lexicale
```

```
; |----------------| |
; | Ptr sur BA Lex3| |
; |----------------|
; |  Var Locale M  | <- SP (BP-8)
; |----------------|
;
;                   writeln('Lex3');
;                   for i := 0 to 1 do
;                         for j := 0 to 1 do
;                               for k := 0 to 1 do
;                                     for m := 0 to 1 do
;                                           writeln(i,j,k,m);
;
;                   Lex4;

Lex3_M      textequ     <word ptr [bx-8]>
m           textequ     <word ptr [bp-8]>

Lex3        proc  near
            enter 2, 3              ;Variable locale de 2 bytes au niveau lex 3.

            nop                     ;Instruction espace pour tracer pas à pas

            print
            byte  "Lex3",cr,lf,0

            mov   i, 0
ForILp:     mov   bx, disp1
            mov   Lex1_J, 0
ForJlp:     mov   bx, disp2
            mov   Lex2_K, 0
ForKLp:     mov   m, 0
ForMLp:     mov   ax, i
            puti
            mov   bx, disp1
            mov   ax, Lex1_J
            puti
            mov   bx, disp2
            mov   ax, Lex2_k
            puti
            mov   ax, m
            puti
            putcr

            inc   m
            cmp   m, 2
            jb    ForMLp

            mov   bx, disp2
            inc   Lex2_K
            cmp   Lex2_K, 2
            jb    ForKLp

            mov   bx, disp1
            inc   Lex1_J
            cmp   Lex1_J, 2
            jb    ForJLp

            inc   i
            cmp   i, 2
            jb    ForILp

            call  Lex4

            leave
            ret
Lex3        endp

; Le bloc d'activation de Lex4 ressemble à ceci:
;
```

```
; | adresse retour |
; |----------------|
; | Lien Dynamique | <- BP
; |----------------|
; | Ptr sur BA Lex1 | |
; |----------------| |
; | Ptr sur BA Lex2| |
; |----------------| | Table lexicale
; | Ptr sur BA Lex3| |
; |----------------| |
; | Ptr sur BA Lex4| |
; |----------------|
; |  Var Locale N  | <- SP (BP-10)
; |----------------|
;
;
;     writeln('Lex4');
;     for i:= 0 to 3 do
;           for j:= 0 to 2 do
;                 write('(',i,',',j,') ');
;     writeln;
;     for k:= 1 downto 0 do
;           for m:= 1 downto 0 do
;                 for n := 0 to 1 do
;                       write('(',m,',',k,',',n,') ');
;     writeln;

n     textequ     <word ptr [bp-10]>

Lex4        proc  near
            enter 2, 4              ;Variable locale de 2 bytes au niveau lex 4.

            nop                     ;Instruction espace pour tracer pas à pas

            print
            byte  "Lex4",cr,lf,0

            mov i, 0
ForILp:     mov   bx, disp1
            mov   Lex1_J, 0
ForJLp:     mov   al, '('
            putc
            mov   ax, i
            puti
            mov   al, ','
            putc
            mov   ax, Lex1_J        ;Notez que BX contient toujours disp1.
            puti
            print
            byte  ") ",0
            inc   Lex1_J            ;BX contient toujours disp1.
            cmp   Lex1_J, 3
            jb    ForJLp

            inc   i
            cmp   i, 4
            jb    ForILp

            putcr

            mov   bx, disp2
            mov   Lex2_K, 1
ForKLp:     mov   bx, disp3
            mov   Lex3_M, 1
ForMLp:     mov   n, 0
ForNLp:     mov   al, '('
            putc

            mov   bx, disp3
            mov   ax, Lex3_M
```

```
                puti
                mov     al, ','
                putc
                mov     bx, disp2
                mov     ax, Lex2_K
                puti
                mov     al, ','
                putc
                mov     ax, n
                puti
                print
                byte ") ",0

                inc     n
                cmp     n, 2
                jb      ForNLp

                mov     bx, disp3
                dec     Lex3_M
                jns     ForMLp

                mov     bx, disp2
                dec     Lex2_K
                jns     ForKLp

                leave
                ret
Lex4            endp
cseg            ends

zzzzzzseg       segment         para public 'zzzzzz'
LastBytes       db              16 dup (?)
zzzzzzseg       ends
                end             Main.
```

12.7.3 Exercices de Passage de Paramètres

L'exercice suivant démontre des passages de paramètre simples. Ce programme passe des tableaux par la référence, des variables word par valeur et par référence et quelques fonctions et procédures par référence. Le programme lui-même trie deux tableaux en utilisant un algorithme de tri générique. L'algorithme de tri est générique parce que le programme principal lui passe une fonction de comparaison et une procédure pour échanger deux éléments si l'un est plus grand que l'autre.

```
; Ex12_3.asm
;
; Ce programme démonstre différentes méthodes de passage de paramètres.
; Il correspond au code (pseudo) Pascal suivant:
;
;
; program main;
; var  i:integer;
;      a:array[0..255] of integer;
;      b:array[0..255] of unsigned;
;
; function LTint(int1, int2:integer):boolean;
; begin
;      LTint := int1 < int2;
; end;
;
; procedure SwapInt(var int1, int2:integer);
; var temp:integer;
; begin
;      temp := int1;
;      int1 := int2;
;      int2 := temp;
; end;
```

```
;
; function LTunsigned(uns1, uns2:unsigned):boolean;
; begin
;      LTunsigned := uns1 < uns2;
; end;
;
; procedure SwapUnsigned(uns1, uns2:unsigned);
; var temp:unsigned;
; begin
;      temp := uns1;
;      uns1 := uns2;
;      uns2 := temp;
; end;
;
; (* Ce qui suit est un tri simple de type Bulle qui triera des tableaux contenant *)
; (* des types de données arbitraires. *)
;
; procedure sort(data:array; elements:integer; function LT:boolean; procedure swap);
; var i,j:integer;
; begin
;
;      for i := 0 to elements-1 do
;            for j := i+1 to elements do
;                  if (LT(data[j], data[i])) then swap(data[i], data[j]);
; end;
;
;
; begin
;
;      for i := 0 to 255 do A[i] := 128-i;
;      for i := 0 to 255 do B[i] := 255-i;
;      sort(A, 256, LTint, SwapInt);
;      sort(B, 256, LTunsigned, SwapUnsigned);
;
;      for i := 0 to 255 do
;      begin
;            if (i mod 8) = 0 then writeln;
;            write(A[i]:5);
;      end;
;
;      for i := 0 to 255 do
;      begin
;            if (i mod 8) = 0 then writeln;
;            write(B[i]:5);
;      end;
;
; end;

              .xlist
              include      stdlib.a
              includelib   stdlib.lib
              .list

              .386
              option       segment:use16

wp            textequ      <word ptr>

dseg          segment      para public 'data'
A             word         256 dup (?)
B             word         256 dup (?)
dseg          ends

cseg          segment      para public 'code'
              assume       cs:cseg, ds:dseg, ss:sseg

; function LTint(int1, int2:integer):boolean;
; begin
;      LTint := int1 < int2;
; end;
```

```
;
; Le bloc d'activation de LTint ressemble à ceci:
;
; |----------------|
; |     int1       |
; |----------------|
; |     int2       |
; |----------------|
; | adresse retour |
; |----------------|
; |   ancien BP    |<- SP, BP
; |----------------|

int1         textequ <word ptr [bp+6]>
int2         textequ <word ptr [bp+4]>

LTint        proc  near
             push  bp
             mov   bp, sp

             mov   ax, int1           ;Compare les deux paramètres
             cmp   ax, int2           ; et retourne vrai si int1<int2.
             setl  al                 ;Comparaison signée ici.
             mov   ah, 0              ;S'assurer d'effacer le byte de H.O..
             pop   bp
             ret   4
LTint        endp

; Le bloc d'activation de Swap ressemble à ceci:
;
; |----------------|
; |    Adresse     |
; |---    de    ---|
; |      int1      |
; |----------------|
; |    Adresse     |
; |---    de    ---|
; |      int2      |
; |----------------|
; | adresse retour |
; |----------------|
; |   ancien BP    |<- SP, BP
; |----------------|
;
; La variable temporaire est gardée dans un registre.
;
; Notez qu'échanger des entiers ou des entiers non signés peut se faire
; avec le même code puisque les opérations sont identiques pour
; les deux types.
;
; procedure SwapInt(var int1, int2:integer);
; var temp:integer;
; begin
;     temp := int1;
;     int1 := int2;
;     int2 := temp;
; end;
;
; procedure SwapUnsigned(uns1, uns2:unsigned);
; var temp:unsigned;
; begin
;     temp := uns1;
;     uns1 := uns2;
;     uns2 := temp;
; end;
;

int1         textequ      <dword ptr [bp+8]>
int2         textequ      <dword ptr [bp+4]>
```

```
SwapInt      proc   near
             push   bp
             mov    bp, sp
             push   es
             push   bx

             les    bx, int1           ;Obtient l'adresse de la variable int1.
             mov    ax, es:[bx]        ;Obtient la valeur d'int1.
             les    bx, int2           ;Obtient l'adresse de la variable int2.
             xchg   ax, es:[bx]        ;Échange la valeur d'int1 avec celle d'int2

             les    bx, int1           ;Obtient l'adresse d'int1 et
             mov    es:[bx], ax        ; y stocke la valeur d'int2.

             pop    bx
             pop    es
             pop    bp
             ret    8
SwapInt      endp

; Le bloc d'activation de LTunsigned ressemble à ceci:
;
; |----------------|
; |      uns1      |
; |----------------|
; |      uns2      |
; |----------------|
; | adresse retour |
; |----------------|
; |    ancien BP   |<- SP, BP
; |----------------|
;
; function LTunsigned(uns1, uns2:unsigned):boolean;
; begin
;     LTunsigned := uns1 < uns2;
; end;

uns1         textequ      <word ptr [bp+6]>
uns2         textequ      <word ptr [bp+4]>

LTunsigned   proc   near
             push   bp
             mov    bp, sp

             mov    ax, uns1           ;Compare uns1 avec uns2 et
             cmp    ax, uns2           ; retourne vrai si uns1<uns2.
             setb   al                 ;Comparaison non-signée.
             mov    ah, 0              ;Retourne un booléen de 16-bit.

             pop bp
             ret 4
LTunsigned   endp

; Le bloc d'activation de Sort ressemble à ceci:
;
; |----------------|
; |    Adresse     |
; |---          ---|
; |  des Données   |
; |----------------|
; |    Eléments    |
; |----------------|
; |    Adresse     |
; |---          ---|
; |     de LT      |
; |----------------|
; |    Adresse     |
; |---          ---|
; |    de Swap     |
; |----------------|
```

```
; | adresse retour |
; |----------------|
; |   ancien BP    |<- SP, BP
; |----------------|
;
; procedure sort(data:array; elements:integer; function LT:boolean; procedure swap);
; var i,j:integer;
; begin
;
;     for i := 0 to elements-1 do
;           for j := i+1 to elements do
;                 if (LT(data[j], data[i])) then swap(data[i], data[j]);
; end;

data        textequ     <dword ptr [bp+10]>
elements    textequ     <word ptr [bp+8]>
funcLT      textequ     <word ptr [bp+6]>
procSwap    textequ     <word ptr [bp+4]>
i           textequ     <word ptr [bp-2]>
j           textequ     <word ptr [bp-4]>

sort        proc  near
            push  bp
            mov   bp, sp
            sub   sp, 4
            push  es
            push  bx

            mov   i, 0
ForILp:     mov   ax, i
            inc   i
            cmp   ax, Elements
            jae   Idone

            mov   j, ax
ForJLp:     mov   ax, j
            cmp   ax, Elements
            ja    Jdone

            les   bx, data          ;Pousse la valeur de
            mov   si, j             ; data[j] sur la
            add   si, si            ; pile
            push  es:[bx+si]

            les   bx, data          ;Pousse la valeur de
            mov   si, i             ; data[i] sur la
            add   si, si            ; pile.
            push  es:[bx+si]

            call  FuncLT            ;Voit si data[i] < data[j]
            cmp   ax, 0             ;Teste le résultat booléen.
            je    NextJ

            push  wp data+2         ;Passe data[i] par référence.
            mov   ax, i
            add   ax, ax
            add   ax, wp data
            push  ax

            push  wp data+2         ;Passe data[j] par référence.
            mov   ax, j
            add   ax, ax
            add   ax, wp data
            push  ax

            call  ProcSwap

NextJ:      inc   j
            jmp   ForJLp
```

```
JDone:      inc   i
            jmp   ForILp

IDone:      pop   bx
            pop   es
            mov   sp, bp
            pop   bp
            ret   10
sort        endp

; Le bloc d'activation de Main ressemble à ceci:
;
; | adresse retour |<- SP, BP
; |----------------|
;
; begin
;
;     for i := 0 to 255 do A[i] := 128-i;
;     for i := 0 to 255 do B[i] := 33000-i;
;     sort(A, 256, LTint, SwapInt);
;     sort(B, 256, LTunsigned, SwapUnsigned);
;
;     for i := 0 to 255 do
;     begin
;           if (i mod 8) = 0 then writeln;
;           write(A[i]:5);
;     end;
;
;     for i := 0 to 255 do
;     begin
;           if (i mod 8) = 0 then writeln;
;           write(B[i]:5);
;     end;
;
; end;

Main        proc
            mov   ax, dseg          ;Initialise les registres de segment.
            mov   ds, ax
            mov   es, ax

; Notez que le code suivant agglomère les deux boucles for d'initialisation
; en une seule boucle.

            mov   ax, 128
            mov   bx, 0
            mov   cx, 33000
ForILp:     mov   A[bx], ax
            mov   B[bx], cx
            add   bx, 2
            dec   ax
            dec   cx
            cmp   bx, 256*2
            jb    ForILp

            push  ds                ;Adresse Seg de A
            push  offset A          ;Offset de A
            push  256               ;# d'éléments dans A
            push  offset LTint      ;Adresse de la routine de comparaison
            push  offset SwapInt    ;Adresse de la routine d'échange
            call  Sort

            push  ds                ;Adresse Seg de B
            push  offset B          ;Offset de B
            push  256               ;# d'éléments dans A
            push  offset LTunsigned ;Adresse de la routine de comparaison
            push  offset SwapInt    ;Adresse de la routine d'échange
            call  Sort

; Affiche les valeurs dans A.
```

```
                mov     bx, 0
ForILp2:        test    bx, 0Fh             ;Voit si (I mod 8) = 0
                jnz     NotMod              ; note: BX mod 16 = I mod 8.
                putcr
NotMod:         mov     ax, A[bx]
                mov     cx, 5
                putisize
                add     bx, 2
                cmp     bx, 256*2
                jb      ForILp2

; Affiche les valeurs dans B.
                mov     bx, 0
ForILp3:        test    bx, 0Fh             ;Voit si (I mod 8) = 0
                jnz     NotMod2             ; note: BX mod 16 = I mod 8.
                putcr
NotMod2:        mov     ax, B[bx]
                mov     cx, 5
                putusize
                add     bx, 2
                cmp     bx, 256*2
                jb      ForILp3

Quit:           ExitPgm                     ;Macro DOS pour quitter le programme.
Main            endp
cseg            ends

sseg            segment         para stack 'stack'
stk             word            256 dup (0)
sseg            ends

zzzzzzseg       segment         para public 'zzzzzz'
LastBytes       db              16 dup (?)
zzzzzzseg       ends
                end             Main
```

12.8 Projets de programmation

1) Écrivez un itérateur auquel vous passez un tableau de caractères par référence. L'itérateur devrait renvoyer un index dans le tableau qui pointe sur un caractère blanc (tout code ASCII inférieur ou égal à un espace) qu'il trouve. À chaque appel, l'itérateur devrait renvoyer l'index du caractère blanc suivant. Il échouera s'il rencontre un byte contenir la valeur zéro. Utilisez des variables locales pour toutes les valeurs dont l'itérateur a besoin.

2) Ecrivez une routine récursive qui fait ce qui suit :

```
function recursive(i:integer):integer;
var j,k:integer;
begin
        j := i;
        k := i*i;
        if (i >= 0) then writeln('AR Address =', Recursive(i-1));
        writeln(i,' ',j,' ',k);
        recursive := Value in BP Register;
end;
```

   Depuis votre programme principal, appelez cette procédure et passez-lui la valeur 10 sur la pile. Vérifiez que vous obtenez des résultats corrects en retour. Expliquez les résultats.

3) Écrivez un programme qui contient une procédure à laquelle vous passez quatre paramètres sur la pile. Ceux-ci devraient être passés par valeur, référence, valeur-résultat et résultat, respectivement (pour le paramètre par valeur-résultat, passez l'adresse de l'objet sur la pile). À l'intérieur de cette procédure, vous devrez appeler trois autres procédures qui prennent également quatre paramètres (chacune). Cependant, la première procédure devrait utiliser le passage par valeur pour chacun des quatre paramètres; la deuxième procédure devrait utiliser le passage par référence pour chacun des quatre paramètres; et la troisième devrait utiliser le passage par valeur-résultat pour chacun des quatre paramètres. Passez les quatre

paramètres de la procédure enfermante comme paramètres à chacune de ces trois procédures enfants. Passez les quatre paramètres de la procédure enveloppante comme paramètres à chacune de ces trois procédures filles. Dans ces procédures, affichez les valeurs des paramètres et modifiez leurs résultats. Immédiatement après le retour de chacune de ces procédures filles, imprimez les valeurs des paramètres. Écrivez un programme principal qui passe quatre variables locales (au programme principal) que vous avez initialisées avec différentes valeurs à la première procédure ci-dessus. Exécutez le programme et vérifiez qu'il fonctionne correctement et qu'il passe correctement les paramètres à chacune de ces procédures.

4) Écrivez un programme qui implémente le code Pascal suivant en assembleur. Supposez que toutes les variables (y compris globales) sont allouées dans des blocs d'activation sur la pile.

```
program nest3;
var    i:integer;
       procedure A(k:integer);

              procedure B(procedure c);
              var m:integer;
              begin

                     for m:= 0 to 4 do c(m);

              end; {B}

              procedure D(n:integer);
              begin

                     for i:= 0 to n-1 do writeln(i);

              end; {D}

              procedure E;
              begin

                     writeln('A stuff:');
                     B(A);
                     writeln('D stuff:');
                     B(D);

              end; {E}

       begin {A}

              B(D);
              writeln;
              if k < 2 then E;

       end; {A}

begin {nest3}

       A(0);

end; {nest3}
```

5) Le programme de la section 12.7.2 (Ex12_2.asm sur le CD-ROM d'accompagnement) utilise les instructions 80286 **enter** et **leave** pour maintenir la table lexicale dans chaque bloc d'activation. Comme précisé dans la section 12.1.6, ces instructions sont tout à fait lentes, particulièrement sur le 80486 et les processeurs postérieurs. Récrivez ce code en remplaçant les instructions **enter** et **leave** avec le code assembleur direct qui fait le même travail. Dans CodeView, tracez pas à pas dans le programme comme pour le deuxième exercice de laboratoire (section 12.7.2) pour vérifier que vos cadres de pile sont identiques à ceux que les instructions **enter** et **leave** produisent.

6) Le programme générique de Tri par Bulle de la section 12.7.3 fonctionne seulement avec des objets de données qui font deux bytes. C'est parce que la procédure de tri passe les valeurs Data[I] et Data[J] sur la pile aux routines de comparaison (LTint et LTunsigned) et parce que la routine de tri multiplie les index de I et de j par deux quand elle indexe dans le tableau de données. C'est une limitation sévère pour cette routine générique de tri. Réécrivez le programme pour le rendre vraiment générique. ceci en écrivant une routine "CompareAndSwap" qui remplacera les appels à LT et Swap. Vous devrez passer à CompareAndSwap le tableau (par référence) et les deux index de tableau (I et j) pour comparer et, si besoin, permuter. Écrivez deux versions de la routine CompareAndSwap, une pour des nombres entiers non signés et une pour des nombres entiers signés. Exécutez ce programme et vérifiez que votre implémentation fonctionne correctement.

12.9 Résumé

Les langages structurés par bloc, comme le Pascal, permettent d'accéder aux variables non-locales à différents niveaux lexicaux. L'accès à des variables non-locales est une tâche complexe exigeant des structures de données spéciales telles qu'une chaîne de liens statiques ou une table lexicale. La table lexicale est probablement la manière la plus efficace pour accéder à des variables non-locales. Les 80286 et les processeurs postérieurs fournissent des instructions spéciales, **enter** et **leave** pour maintenir une liste de tables lexicales, mais ces instructions sont trop lentes pour les usages les plus communs. Pour les détails additionnels, voyez

- "Imbrication lexicale, liens statiques et tables lexicales" à la Section 12.1
- "Portée" à la Section 12.1.1
- "Liens statiques" à la Section 12.1.3
- "Accès à des variables non-locales en utilisant des liens statiques" à la Section 12.1.4
- "La table lexicale" à la Section 12.1.5
- "Les instructions ENTER et LEAVE du 80286 à la Section 12.1.6
- "Passage de variables à différents niveaux lexicaux comme paramètres" à la Section 12.2
- "Passage de paramètres comme paramètres à une autre procédure" à la Section 12.3
- "Passage de procédures comme paramètres" à la Section 12.4

Les itérateurs sont un croisement entre une fonction et une structure de boucle. Ils sont une construction de programmation très puissante, disponible dans beaucoup de langages de niveau très élevé. L'implémentation efficace des itérateurs implique la manipulation soigneuse de la lors de l'exécution. Pour voir comment implémenter des itérateurs, lisez les sections suivantes:

- "Les Itérateurs" à la la Section 12.5
- "Implémentation d'itérateurs en utilisant l'expansion en-ligne" à la la Section 12.5.1
- "Implémentation d'itérateurs avec des blocs de reprise" à la la Section 12.5.2
- "Un exemple d'itérateur" à la la Section 12.6.1
- "Un autre exemple d'itérateur" à la la Section 12.6.2

12.10 Questions

1) Qu'est-ce qu'un itérateur ?
2) Qu'est-ce qu'un bloc de reprise ?
3) Comment les itérateurs de ce chapitre implémentent-ils les résultats de succès et d'échec ?
4) À quoi ressemble la pile lorsqu'on exécute le corps d'une boucle contrôlée par un itérateur ?
5) Qu'est-ce qu'un lien statique ?
6) Qu'est-ce qu'une table lexicale ?
7) Décrivez comment accéder à une variable non-locale en utilisant des liens statiques.
8) Décrivez comment accéder à une variable non-locale en utilisant une table lexicale.

9) Comment accéderiez-vous à une variable non-locale en utilisant la table lexicale établie par l'instruction ENTER du 80286 ?

10) Dessinez une image du bloc d'activation pour une procédure au niveau lexical 4 qui utilise l'instruction ENTER pour établir la table lexicale.

11) Expliquez pourquoi les liens statiques fonctionnent mieux qu'une table lexicale quand on passe des procédures et des fonctions comme paramètres.

12) Supposez que vous vouliez passer une variable intermédiaire par valeur-résultat en utilisant la technique où vous poussez la valeur avant d'appeler la procédure et puis extrayez la valeur de la pile (en la stockant de nouveau dans la variable intermédiaire) au retour de la procédure. Fournissez deux exemples, un utilisant les liens statiques et un utilisant une table lexicale, qui implémente le passage par valeur-résultat de cette manière.

13) Convertissez le (pseudo) code Pascal suivant en assembleur 80x86. Supposez que le Pascal supporte le passage de paramètres par nom et par évaluation-paresseuse comme suggérés par le code suivant.

```
program main;
var k:integer;

procedure one(LazyEval i:integer);
begin
       writeln(i);
end;

procedure two(name j:integer);
begin
       one(j);
end;

begin {main}
       k := 2;
       two(k);
end;
```