ROZDZIAŁ DWUNASTY: **PROCEDURY: ZAAWANSOWANE TEMATY**

Ostatni rozdział omawiał jak tworzyć procedury, przekazywać parametry i alokować oraz uzyskać dostęp do zmiennych lokalnych. Ten rozdział omawia jak uzyskać dostęp w innych procedurach, przekazywanie procedur jako parametrów i implementacje jakichś zdefiniowanych przez użytkownika struktur sterujących.

12.0 WSTĘP

Rozdział ten całkowicie omawia procedury, parametry i zmienne lokalne rozpoczęte w poprzednim rozdziale. Rozdział ten opisuje jak języki o strukturze blokowej takie jak Pascal, Odula-2, Algol i Ada uzyskują dostęp do lokalnych i nielokalnych zmiennych. Omawia również jak zaimplementować zdefiniowane przez użytkownika strukturę sterującą, iterator. Większość materiału w tym rozdziale będzie interesująca dla piszących kompilatory i tych którzy chcą nauczyć się jak kompilator generuje kod dla pewnych typów konstrukcji programowych.. Kilka czystych programów języka asemblera będzie używać technik, które opisuje ten rozdział. Dlatego niewiele z tego materiału tego rozdziału nie jest szczególnie ważne dla tych ,którzy chcą tylko nauczyć się języka asemblera. Jednakże, jeśli mamy zamiar pisać kompilatory, lub chcemy zrozumieć jak kompilator generuje kod, żeby pisać wydajne programy w HLL'ach, będziemy musieli nauczyć się tego materiału wcześniej czy później.

Rozdział ten zaczyna się od omówienia pojęcia **zasięg** i jak HLL'e jak Pascal uzyskują dostęp do zmiennych w zagnieżdżonych procedurach. Pierwsza sekcja omawia koncepcję zagnieżdżenia leksykalnego i zastosowania wiązań statycznych i terminali do uzyskania dostępu do zmiennych nielokalnych. Następnie ten rozdział opisuje jak przekazać zmienne spod różnych poziomów leksykalnych jako parametry. Trzecia sekcja omawia jak przekazać parametry z jednej procedury jako parametry do innej procedury. Czwarta zasadniczy temat tego rozdziału opisuje przekazywanie procedur jako parametrów. Rozdział kończy się opisaniem iteratorów, zdefiniowanej przez użytkownika struktury danych.

Rozdział ten zakłada znajomość języków o strukturze blokowej takich jak Pascal czy Ada. Jeśli nasze doświadczenia z HLL'ami są związane z językami o strukturze nie blokowej takimi jak C, C++, BASIC czy FORTRAN, niektóre z koncepcji przedstawianych w tym rozdziale może być zupełnie nowych i możemy mieć problemy z ich zrozumieniem. Jakiś wstępny tekst o Pascalu lub Adzie bezie pomocny przy wyjaśnieniu niezrozumiałych koncepcji, które ten rozdział zakłada jako warunek wstępny .

12.1 ZAGNIEŻDŻENIA LEKSYKALNE, ŁĄCZENIA STATYCZNE I DISPLAY

W języku o strukturze blokowej, takim jak Pascal możliwe jest zagnieżdżanie procedur i funkcji. Zagnieżdżanie jednej procedury wewnątrz innej ogranicza dostęp do zagnieżdżonej procedury; nie możemy uzyskać dostępu do zagnieżdżonej procedury z zewnątrz otaczającej procedury. Podobnie zmienne zadeklarowane wewnątrz procedury są widoczne wewnątrz tej procedury i dla wszystkich procedur zagnieżdżonych wewnątrz tej procedury. Jest to standardowe pojęcie zakresu języka o strukturze blokowej, które powinno być dobrze znane, każdemu kto pisał programy w Pascalu lub Adzie.

Jest wiele złożoności ukrytej za koncepcją zakresu, lub leksykalnego zagnieżdżenia, w języku o strukturze blokowej. Podczas gdy dostęp do zmiennych lokalnych w bieżącym rekordzie aktywacji jest wydajny, dostęp do

zmiennych globalnych w języku o strukturze blokowej może być bardzo niewydajny. Sekcja ta będzie opisywać jak HLL'e jak Pascal zajmują się nielokalnymi identyfikatorami i jak uzyskują dostęp do zmiennych globalnych i wywołują nielokalne procedury i funkcje.

12.1.1 ZASIĘG

Zasięg w większości języków wysokiego poziomu jest pojęciem statycznym lub wykonywanym w czasie kompilacji. Zasięg jest pojęciem ,kiedy nazwa jest widzialna lub dostępna wewnątrz programu. Ta zdolność do ukrywania nazw jest użyteczna w programach ponieważ jest to często dogodne przy wielokrotnym używaniu pewnych (nie opisowych) nazw. Zmienna i stosowana do sterowania większości pętli for w językach wysokiego poziomu jest doskonałym przykładem. W całym rozdziale będziemy widzieli coś takiego jak xyz_i, xyz_j itp. Powód dla wyboru takich nazw jest taki ,że MASM nie wspiera koncepcji zasięgu nazw jak języki wysokiego poziomu. NA szczęście MASM 6.x i późniejsze wspierają zasięg nazw.

Domyślnie MASM 6.x traktuje etykiety instrukcji (te z dwukropkiem po nich) jako lokalne dla procedury. To znaczy, możemy tylko odnosić się do takich etykiet wewnątrz procedury w której są zadeklarowane. Jest to prawda nawet jeśli zagnieżdzimy jedną procedurę wewnątrz innej. Na szczęście, nie ma dobrego powodu aby chcieć zagnieżdżać procedury w programie MASM;

Posiadanie lokalnych etykiet wewnątrz procedury jest miłe. Pozwala to nam na ponowne użycie etykiety instrukcji (np. etykieta pętli) bez martwienia się o konflikt nazw z innymi procedurami. Czasami jednakże, możemy chcieć wyłączyć zasięg nazw w procedurze; dobrym przykładem jest to kiedy mamy instrukcję case, której tablica skoków pojawia się na zewnątrz procedury. Jeśli etykiety instrukcji case są lokalne w tej procedurze, nie będą widzialne na zewnątrz procedury i nie można ich użyć przy tablicy skoków instrukcji case. Są dwa sposoby w jaki możemy wyłączyć zasięg etykiet w MASM 6.x. Pierwszy sposób to zawarcie instrukcji w naszym programie:

```
option          nonscoped
```

To pozwoli wyłączyć zasięg zmiennych od tego punktu w przód w naszym pliku źródłowym programu. Możemy z powrotem włączyć zasięg poprzez instrukcję w postaci

```
option          scoped
```

Poprzez umieszczenie tych instrukcji wokół naszej procedury możemy wybiórczo sterować zasięgiem.

Inny sposób do sterowania zasięgiem pojedynczych nazw jest umieszczenie podwójnego dwukropka („::") po etykiecie. Informuje to asembler ,że ta szczególna nazwa powinna być globalna dla otaczającej procedury.

MASM podobnie jak język C, wspiera trzy poziomy zasięgu: publiczny, globalny (lub statyczny) i lokalny. Symbole lokalne są widoczne tylko wewnątrz procedury, w której są zdefiniowane. Symbole globalne są dostępne w całym pliku źródłowym, ale nie są widoczne w innych modułach programu. Symbole publiczne są widoczne w całym programie, w modułach. MASM używa następujących domyślnych zasad zasięgu:

*Domyślnie etykieta instrukcji pojawia się w procedurze jako lokalna dla tej procedury

*Domyślnie wszystkie nazwy procedur są publiczne

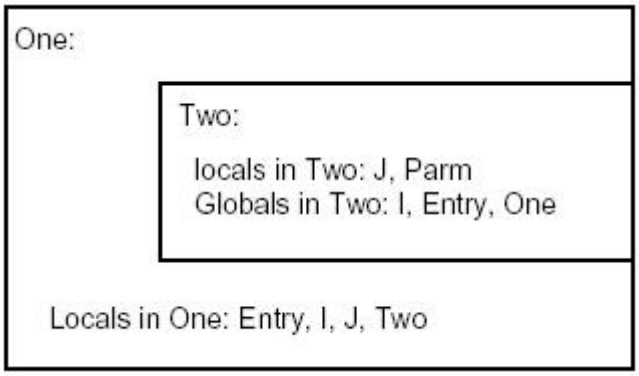
*Domyślnie większość innych symboli jest globalna

Zauważmy, że te zasady dotyczą tylko MASMa 6.x. Inne asemblery i wcześniejsze wersje MASMa korzystają z różnych zasad.

Przesłonięcie domyślności pierwszej z powyższych zasad jest łatwe - albo zastosujemy instrukcję option nonscoped albo zastosujemy podwójny dwukropek dla uczynienia globalnej etykiety. Powinniśmy być świadomi, że nie możemy uczynić lokalnej etykiety publicznej stosując dyrektyw public lub externdef. Możemy uczynić symbol globalnym (stosując obojętnie jaką technikę) przed uczynieniem ją publiczną.

Posiadanie wszystkich nazw procedur publicznymi domyślnie zazwyczaj nie jest dużym problemem. Jednakże może się okazać, że chcemy zastosować tą samą (lokalna) nazwę procedury w kilku różnych modułach. Jeśli MASM automatycznie uczyni takie nazwy publicznymi, linker da nam błąd ponieważ są to wielokrotne publiczne procedury o tej samej nazwie. Możemy włączyć lub wyłączyć tą domyślną akcję stsoując poniższe instrukcje:

```
option    proc:private                ;procedury są globalne
```



Rysunek 12. Identyfikacja zasięgu

Option proc:export ;procedury są publiczne

Zauważmy ,że jakieś debuggery tylko dostarczają informacji symbolicznych jeśli nazwa procedury jest publiczna. Jest tak dlatego, że MASM 6.x domyślnie ustawia je na nazwy publiczne. Problem ten nie istnieje w CodeView; więc możemy zastosować którykolwiek jest bardziej dogodny. Oczywiście, jeśli wybierzemy prywatne nazwy procedur (tylko globalne), wtedy będziemy musieli użyć dyrektyw public lub externdef dla uczynienia żądanej nazwy procedury publiczną.

To omówienie lokalnych, globalnych i publicznych symboli stosuje się głownie do instrukcji i etykiet procedur. Nie ma zastosowania do zmiennych zadeklarowanych w segmencie danych, przyrównań, makr typedefów lub większości innych symboli. Takie symbole są zawsze globalne bez względu na to gdzie je zdefiniujemy. Jedyny sposób uczynienia ich publicznymi jest wyszczególnienie ich nazw dyrektywami public lub externdef

Jest sposób deklaracji nazw parametrów i zmiennych lokalnych, alokowanych na stosie, taki ,że ich nazwy są lokalne dla danej procedury. Zobacz do dyrektywy proc w podręczniku do MASMa po szczegóły

Zakres nazwy ogranicza swoją widzialność wewnątrz programu. To znaczy, program ma dostęp do nazwy zmiennej tylko wewnątrz tego zasięgu nazw. Na zewnątrz zasięgu program nie ma dostępu do tej nazwy. Wiele języków programowania, takich jak Pascal i C++ pozwalają nam na ponowne użycie identyfikatorów jeśli zasięg tych wielokrotnych użyć nie zachodzi na siebie. Jak widzieliśmy MASM dostarcza minimalnych cech zasięgu etykiet instrukcji. Jest jednak inna kwestia związana z zasięgiem: powiązanie adresu i czas życia zmiennej. Powiązanie adresu jest to działanie kojarzenia adresu pamięci z nazwą zmiennej. Czas życia zmiennej jest tą częścią wykonywanego programu podczas którego komórka pamięci jest ograniczana do zmiennej. Rozważmy poniższą procedurę Pascalowską:

```
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;
```

Rysunek 12.1 pokazuje zasięg identyfikatorów One, Two, Entry, i , j i Param.
Lokalna zmienna j w Two maskuje identyfikator j w procedurze One wewnątrz Two

12.1.2 AKTYWACJA ELEMENTU, POWIĄZANIE ADRESU I CZAS ŻYCIA ZMIENNEJ

Aktywacja elementu jest procesem wywoływania procedury lub funkcji. Kombinacja rekordu aktywacji i jakiegoś kodu wykonywalnego jest uważane za przypadek podprogramu. Kiedy występuje aktywacja elementu podprogram wiąże adres maszynowy do swoich lokalnych zmiennych. Adres powiązany (dla zmiennych lokalnych) występuje wtedy kiedy podprogram modyfikuje wskaźnik stosu aby zrobić miejsce dla zmiennych lokalnych. Czas życia tych zmiennych jest od tego punktu do momentu kiedy podprogram niszczy rekord aktywacji eliminując pamięć dla zmiennych lokalnych.

Chociaż zasięg ogranicza widzialność nazw do pewnej części kodu i nie pozwala na powtarzanie nazw wewnątrz tego samego zasięgu, nie znaczy to, że jest tylko jeden adres graniczny dla nazwy. Jest całkiem możliwe, że ma kilka adresów granicznych dla tej samej nazwy w tym samym czasie. Rozważmy wywołanie rekurencyjne procedury. Przy każdej aktywacji procedura buduje nowy rekord aktywacji. Ponieważ poprzednia instancja jeszcze istnieje , teraz są dwa rekordy aktywacji na stosie zawierające zmienne lokalne dla tej procedury. Ponieważ dodatkowo wystąpiła rekurencyjna aktywacja, system buduje więcej rekordów aktywacji, każdy z adresem granicznym do tej samej nazwy. Rozwiązanie tej możliwej dwuznaczności (który adres jest dostępny kiedy działamy na zmiennej?), system zawsze manipuluje zmienną w ostatnim rekordzie aktywacji.

Zauważmy, że procedury One i Two w poprzedniej sekcji są pośrednio rekurencyjne. To znaczy, obie wywołują podprogramy które ,po kolei , wywołuje je same. Zakładając, że parametr One jest mniejszy niż 10 przy inicjalizacji wywołania, kod ten będzie generował wiele rekordów aktywacji (i dlatego też wiele kopii zmiennych lokalnych) na stosie. Na przykład, mamy wywołanie One(9),wtedy stos wygląda tak jak na rysunku 12.2 po pierwszym napotkaniu end związanego z procedurą Two.

Jak możemy zobaczyć, jest kilka kopii I i J na stosie w tym punkcie. Procedura Two (bieżący wykonywany podprogram) uzyskuje dostęp do J w ostatnim rekordzie aktywacji, który jest na samym dole rysunku 12.2. Poprzednia instancja Two uzyskiwała dostęp do zmiennej J tylko w swoim rekordzie aktywacji, kiedy bieżąca instancja wracała do One a potem cofała do Two.

Czas życia instancji zmiennych jest od punktu stworzenia rekordu aktywacji do punktu usunięcia rekordu aktywacji. Zauważmy ,że pierwsza instancja J ,powyżej, (na szczycie powyższego diagramu) ma najdłuższy czas życia ze wszystkich zachodzących na siebie instancji J.

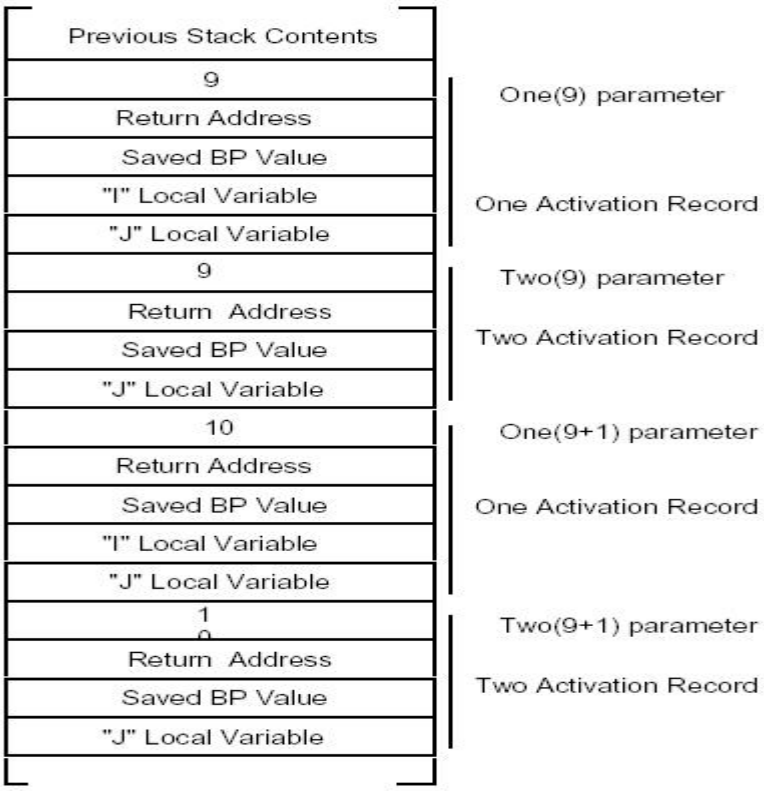
12.1.3 ŁĄCZENIA STATYCZNE

Pascal pozwala procedurze Two uzyskać dostęp do I w procedurze One. Jednakże, kiedy istnieje możliwość rekurencji może być kilka instancji I na stosie. Pascal, oczywiście, pozwoli tylko procedurze Two na dostęp do ostatniej instancji I. Na diagramie stosu na rysunku 12.2, odpowiada to wartości I w rekordzie aktywacji, który zaczyna się z „parametrem One(9+1)". Jedynym problemem jest to „jak się dowiedzieć gdzie znajduje się rekord aktywacji zawierający I"?

Szybką ale kiepską myślą, jest odpowiedź, że jest to po prostu indeks wsteczny do stosu. W końcu możemy łatwo zobaczyć na powyższym diagramie, że I jest offsetem osiem z rekordu aktywacji Two. Niestety, nie zawsze jest taki przypadek. Zakładamy, że procedura Three również wywołuje procedurę Two a poniższa instrukcja pojawia się wewnątrz procedury One:

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

Z tą instrukcją w tym miejscu jest całkiem możliwe jest posiadanie dwóch różnych ramek stosu na wejściu do Procedury Two: jeden z rekordem aktywacji dla procedury Three wciśnięty między rekordy aktywacji One i Two i jeden z rekordami aktywacji dla procedur One i Two przylegającymi jeden do drugiego. Najwyraźniej stały offset z rekordu aktywacji Two nie zawsze wskazuje na zmienną I w ostatnim rekordzie aktywacji.

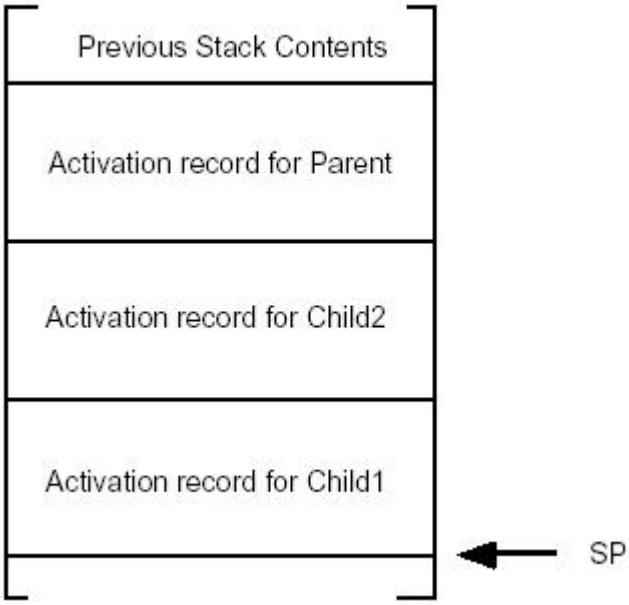


Rysunek 12.2 Pośrednia rekurencja

Przebiegły czytelnik może zauważyć, że zachowana wartość bp w rekordzie aktywacji Two wskazuje na wywołujący rekord aktywacji. Można pomyśleć, że możemy użyć tego jako wskaźnika do rekordu aktywacji One. Ale ten schemat jest zawodny z tego samego powodu że stały offset zawodzi technicznie. A stara wartość Bp, łączona dynamicznie, wskazuje na wywołujący rekord aktywacji. Ponieważ kod wywołujący nie jest konieczni otoczony procedurą, łączenie dynamiczne może nie wskazywać na otaczający rekord aktywacji procedury

Jaka jest rzeczywista potrzeba wskaźnika do otaczającego rekordu aktywacji procedury. Wiele kompilatorów w językach o strukturze blokowej tworzy takie wskaźniki, łączenie (linkowanie) statyczne. Rozważmy poniższy Pascalowski kod:

```
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};
```



Rysunek 12. 3 Rekord aktywacji po kilku zagnieżdżonych wywołaniach

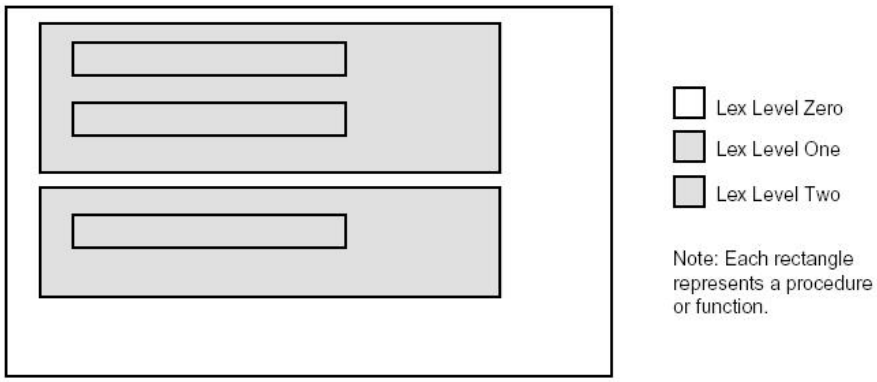
```
begin {Parent}
        Child2;
        Child2;
end;
```

Po wprowadzeniu po raz pierwszy Child1, stos będzie wyglądał jak na rysunku 12.3.Kiedy Child1 spróbuje uzyskać dostęp do zmiennej i w Parent, potrzebny będzie wskaźnik, statyczne łącze, do rekordu aktywacji Parent. Niestety nie ma sposobu dla Child1, na wejściu, ‚wykombinować gdzie leży w pamięci rekord aktywacji Parent. Byłoby to konieczne dla kodu wywołującego (Child2 w tym przykładzie) dla przekazania łącza statycznego do Child1. Ogólnie, kod wywoływany może traktować łącze statyczne jako inny parametr; zazwyczaj odkładany na stos bezpośrednio przed wykonaniem instrukcji call.

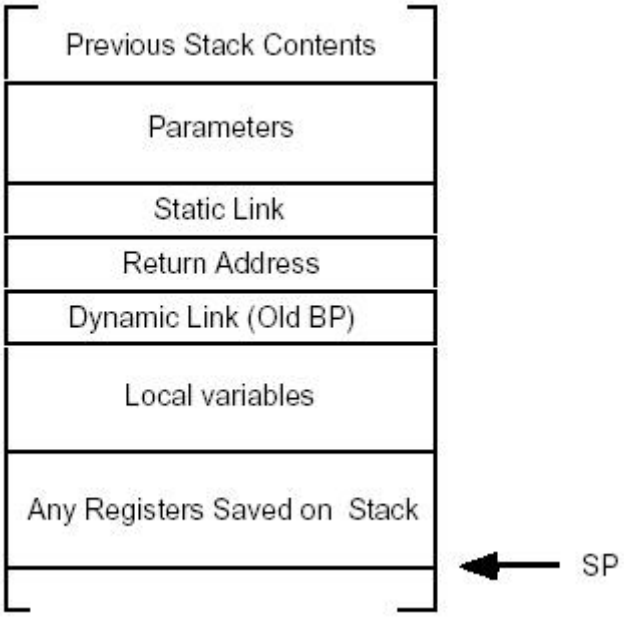
Aby w pełni zrozumieć jak przekazujemy łącze statyczne z wywołania do wywołania, musimy najpierw zrozumieć koncepcję poziomów leksykalnych. Poziomy leksykalne w Pascalu odpowiadają statycznemu zagnieżdżeniu poziomów procedur i funkcji. Wielu piszących kompilatory wyszczególnia lex poziom zero jako główny program. To znaczy wszystkie symbole ‚deklarowane w głównym programie istnieją jako lex poziom zero. Nazwy procedur i funkcji pojawiających się w programie głównym definiują lex poziom jeden, nie zależnie jak dużo procedur i funkcji pojawia się w programie głównym. Wszystkie one zaczynają nową kopię lex poziomu jeden. Dla każdego poziomu zagnieżdżenia Pascal wprowadza nowy lex poziom. Rysunek 12.4 pokazuje to.
Podczas wykonywania, program może tylko uzyskać dostęp do zmiennych na lex poziomie mniejszym lub równym poziomowi bieżącego podprogramu. Ponadto tylko jeden zbiór wartości na danym lex poziomie jest dostępny w jednym czasie a wartości te są zawsze w ostatnim rekordzie aktywacji przy tym lex poziomie.

Przed zamartwianiem się o to jak uzyskać dostęp do nie lokalnych zmiennych używając łącza statycznego musimy wykombinować jak przekazać łączę statyczne jako parametr. Kiedy przekazujemy łącze statyczne jako parametr do jednostki programowej (procedury lub funkcji), są trzy typu sekwencji wywołania:

- jednostka programowa wywołuje „potomka” procedury lub funkcji. Jeśli bieżący lex poziom jest n, wtedy procedura lub funkcja „potomna” jest na lex poziomie n+1 i jest lokalna



Rysunek 12.4 Schematyczne pokazanie poziomów leksykalnych procedury



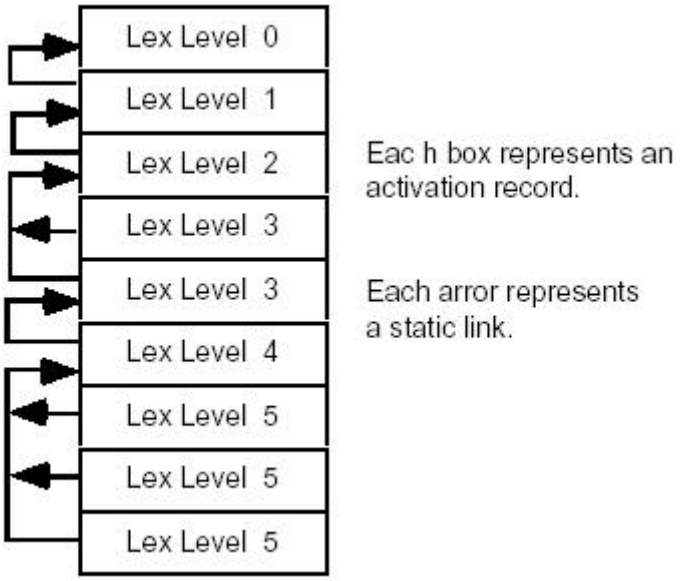
Rysunek 12.5 Ogólny rekord aktywacji

dla bieżącej jednostki programowej. Zauważmy, że większość języków o strukturze blokowej nie pozwala na wywoływanie procedur lub funkcji spod lex poziomów większych niż n+1

- Jednostka programowa wywołuje równorzędne procedury lub funkcje. Równorzędna procedura lub funkcja jest na tym samym poziomie leksykalnym jak bieżący kod wywołujący a pojedyncza jednostka programowa łączy obie jednostki programowe
- Jednostka programowa wywołuje przodka procedury lub funkcji. Jednostka przodka jest albo jednostką macierzystą, rodzicem jednostki przodka lub równorzędną jednostką przodka

Sekwencje wywoływania dla pierwszych dwóch typów powyższych wywołań są bardzo proste. Przez wzgląd na ten przykład założymy, że rekord aktywacji dla tej procedury przybiera ogólną postać jak na rysunku 12.5.

Kiedy procedura lub funkcja macierzysta wywołują jednostkę programową potomka, łącze statyczne jest niczym więcej jak tylko wartością w rejestrze bp bezpośrednio przed wywołaniem. Dlatego też, przekazanie łącza statycznego do jednostki potomnej, jedynie odkładając bp przed wykonaniem instrukcji call:



Rysunek 12.6: Łącze statyczne

```
<Odłożenie innych parametrów na stos>
        push bp
        call    ChildUnit
```

Oczywiście jednostka programowa potomka przetwarza łącze statyczne za stosem podobnie jak inne parametry. W tym przypadku tak statyczne jak i dynamiczne łącze są takie same. Ogólnie jednak nie jest to prawda.

Jeśli jednostka programowa wywołuje równorzędną procedurę lub funkcję, bieżąca wartość w bp nie jest łączem statycznym. Jest wskaźnikiem do kodu wywołującego zmienne lokalne a równorzędna procedura nie może uzyskać dostępu do tych zmiennych. Jednakże, kod wywołujący i wywoływany dzielą tą samą macierzystą jednostkę programową, wiec kod wywołujący może po prostu odłożyć kopię swojego łącza statycznego na stos przed wywołaniem równorzędnej procedury lub funkcji. Poniższy kod robi to, zakładając że wszystkie procedury i funkcję są bliskie.:

```
<Odłożenie innych parametrów na stos>
        push    [bp+4]          ;odłożenie łącza statycznego na stos
        call    PeerUnit
```

Jeśli procedura lub funkcja jest daleka, łącze statyczne będzie dwa bajty dalej na stosie, więc będzie trzeba zastosować poniższy kod:

```
<Odłożenie innych parametrów na stos>
        push    [bp+6]          ;odłożenie łącza statycznego na stos
        call    PeerUnit
```

Wywoływanie przodka jest trochę bardziej złożone. Jeśli jesteśmy obecnie na lex poziomie n i życzymy sobie wywołać przodka z lex poziomu m (m<n), będziemy musieli przejrzeć listę łącz statycznych aby znaleźć żądany rekord aktywacji. Łącza statyczne mają postać listy rekordów aktywacji. Poprzez przejście tego łańcucha rekordów aktywacji aż do jego końca, możemy kroczyć przez ostatnie rekordy aktywacji wszystkich otaczających procedur i funkcji poszczególnych jednostek programowych. Diagram stosu na rysunku 12.6 pokazuje statyczne łącza dla sekwencji wywołań procedur statycznie zagnieżdżonych na głębokość pięciu lex poziomów.

Jeśli jednostka programowa obecnie wykonująca piąty lex poziom życzy sobie wywołać procedurę spod poziomu trzeciego, musi odłożyć łącze statyczne do ostatniej aktywnej jednostki programowej spod lex poziomu dwa. Żeby znaleźć to statyczny łącze będziemy musieli przemierzyć łańcuch łączy statycznych. Jeśli jesteśmy na lex poziomie n i chcemy wywołać procedurę spod lex poziomu m będziemy musieli przejrzeć (n-m)+1 łączy statycznych. Osiągniemy to tym kodem:

```
;Bieżący lex poziom to 5. Kod ten lokuje łącze statyczne dla i potem wywołuje procedurę z lex poziomu 2
;Zakładamy, że wszystkie wywołania są bliskie:
        <Odłożenie koniecznych parametrów?
                mov     bx, [bp+4]                      ;Przejrzenie łączy statycznych do LL 4
```

```
        mov     bx, ss:[bx+4]           ;do Lex Poziomu 3
        mov     bx, ss:[bx+4]           ;do Lex Poziomu 2
        push    ss:[bx+4]
        call    ProcAtLL2
```

Zauważmy ,że prefiks ss: w powyższych instrukcjach. Pamiętamy ,że rekordy aktywacji wszystkie są w segmencie stosu a indeksy bx domyślnie w segmencie danych.

12.1.4 UZYSKANIE DOSTĘPU DO ZMIENNYCH NIE LOKALNYCH STOSUJĄC ŁACZA STATYCZNE

Żeby uzyskać dostęp do nie lokalnych zmiennych, musimy przeglądnąć łańcuch łącz statycznych aż do uzyskania wskaźnika do żądanego rekordu aktywacji. Operacja ta jest podobna do lokalizowania łącz statycznych dla wywoływania procedur naszkicowana w poprzedniej sekcji, z wyjątkiem tego, że przeglądamy tylko n-m łącz statycznych zamiast (n-m) + 1 łącz uzyskujących wskaźnik do odpowiedniego rekordu aktywacji. Rozważmy poniższy kod Pascalowski:

```
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}
```

Procedura Inner uzyskuje dostęp do globalnych zmiennych spod lex poziomu n-1 i n-2 (gdzie n jest lex poziomem procedury) Procedura Middle uzyskuje dostęp do pojedynczej zmiennej globalnej spod lex poziomu m-1 (gdzie m jest lex poziomem procedury Middle) Poniższy kod jeżyka asemblera implementuje te trzy procedury:

```
Outer           proc    near
                push    bp
                mov     bp, sp
                sub     sp, 2                   ;Make room for I.

                mov     word ptr [bp-2],1       ;Set I to one.
                push    bp                      ;Static link for Middle.
                call    Middle

                mov     sp, bp                  ;Remove local variables.
                pop     bp
                ret     2                       ;Remove static link on ret.
Outer           endp

Middle          proc    near
```

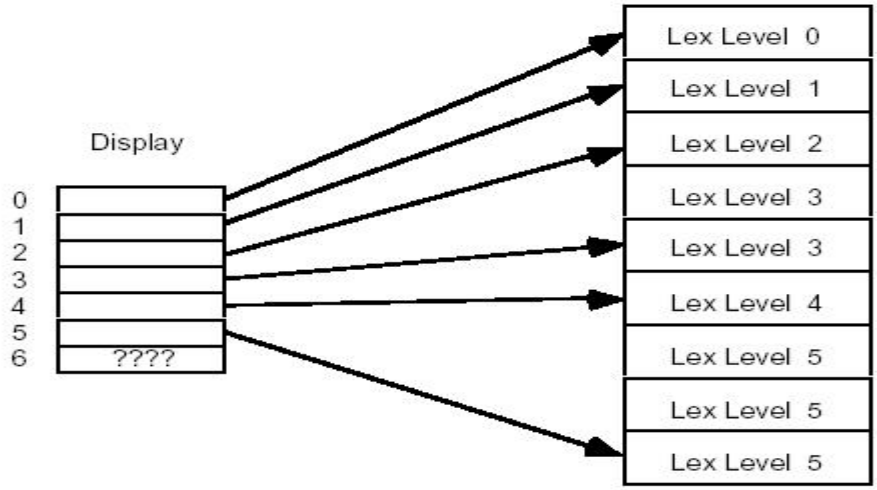
```
                push    bp                      ;Save dynamic link
                mov     bp, sp                  ;Set up activation record.
                sub     sp, 2                   ;Make room for J.

                mov     word ptr [bp-2],2       ;J := 2;
                mov     bx, [bp+4]              ;Get static link to prev LL.
                mov     ax, ss:[bx-2]           ;Get I's value.
                add     ax, [bp-2]              ;Add to J and then
                puti                            ; print the sum.
                putcr
                push    bp                      ;Static link for Inner.
                call    Inner

                mov     sp, bp
                pop     bp
                ret     2                       ;Remove static link on RET.
Middle          endp

Inner           proc    near
                push    bp                      ;Save dynamic link
                mov     bp, sp                  ;Set up activation record.
                sub     sp, 2                   ;Make room for K.

                mov     word ptr [bp-2],2       ;K := 3;
                mov     bx, [bp+4]              ;Get static link to prev LL.
                mov     ax, ss:[bx-2]           ;Get J's value.
                add     ax, [bp-2]              ;Add to K

                mov     bx, ss:[bx+4]           ;Get ptr to Outer's Act Rec.
                add     ax, ss:[bx-2]           ;Add in I's value and then
                puti                            ; print the sum.
                putcr

                mov     sp, bp
                pop     bp
                ret     2                       ;Remove static link on RET.
Inner           endp
```

Jak widzimy, dostęp do zmiennych globalnych może być bardzo nieefektywny.

Zauważmy, że ponieważ różnice pomiędzy rekordami aktywacji rosną, stają się mniej i mniej wydajne przy uzyskiwaniu dostępu do zmiennych globalnych. Uzyskanie dostępu do zmiennych globalnych w poprzednim rekordzie aktywacji wymagało tylko jednej dodatkowej instrukcji przy dostępie, przy dwóch lex poziomach potrzebujemy dwóch dodatkowych instrukcji, itd. Jeśli przeanalizujemy dużą liczbę programów pascalowskich, odkryjemy że większość z nich nie zagnieżdża procedur i funkcji a w tych gdzie są zagnieżdżone jednostki programowe, rzadko uzyskują dostęp do zmiennych globalnych. Jest jednak jeden ważny wyjątek. Chociaż procedury i funkcje Pascala rzadko uzyskują dostęp do lokalnych zmiennych wewnątrz innych procedur i funkcji, one często uzyskują dostęp do zmiennych globalnych zadeklarowanych w programie głównym. Ponieważ takie zmienne pojawiają się w lex poziomie zero, dostęp do takich zmiennych może być tak niewydajny jak możliwy kiedy używamy łącz statycznych. Rozwiązując ten drobny problem, większość opartych na 80x86 języków o strukturze blokowej alokuje zmienne na lex poziomie zero bezpośrednio w segmencie danych i uzyskując dostęp do nich bezpośrednio.

12.1.5 **Display (Wyświetlanie)**

Po przeczytaniu poprzedniej sekcji mogliśmy nabrać pewności, że nigdy nie powinniśmy stosować nie lokalnych zmiennych lub ograniczyć nielokalny dostęp do tych zmiennych zadeklarowanych na lex poziomie zero. W końcu jest często dosyć łatwo włożyć wszystkie dzielone zmienne na lex poziom zero. Jeśli jesteśmy projektantami języka programowania, możemy zaadoptować filozofię projektowania języka C i po prostu nie dostarczać struktur blokowych. Taki kompromis okazuje się być niepotrzebny. Jest struktura danych, **display (wyświetlanie)**, która dostarcza wydajnego dostępu do każdego zbioru nie lokalnych zmiennych.



Rysunek 12.7 Display

Display jest po prostu tablicą wskaźników do rekordów aktywacji. Display[0] zawiera wskaźnik do ostatniego rekordu aktywacji dla lex poziomu zero. Display[1] zawiera wskaźnik do ostatniego rekordu aktywacji dla poziomu jeden i tak dalej. Zakładając ,że zachowujemy tablicę Display w bieżącym segmencie danych (zawsze dobre miejsce do jej trzymania), to tylko dzięki dwóm instrukcjom uzyskujemy dostęp do nie lokalnych zmiennych. Display pracuje tak jak pokazano na rysunku 12.7

Zauważmy ,że wejście w display'u zawsze wskazuje ostatni rekord aktywacji dla procedury na danym lex poziomie. Jeśli nie ma żadnego aktywnego rekordu aktywacji dla poszczególnego lex poziomu (np. powyżej lex poziom sześć) wtedy wejście w display'u zawiera śmieci.

Maksymalne zagnieżdżenie poziomów leksykalnych w naszym programie określa jak dużo elementów musi być w display'u. Większość programów ma tylko trzy lub cztery zagnieżdżone procedury więc display jest zazwyczaj bardzo mały. Ogólnie, rzadko będziemy wymagali więcej niż 10 lub więcej elementów w display'u.

Inną zaletą zastosowania display'a jest to ,że każdy pojedyncza procedura może zachować informację display'a ,kod wywołujący nie musi być zawiły. Kiedy stosujemy łącza statyczne kod wywołujący musi obliczyć i przekazać właściwe łącze statyczne do procedury. To nie tylko jest wolne ale kod który to robi musi pojawiać się przed każdym wywołaniem. Jeśli nasz program używa stosuje display'a kod wywoływany zamiast wywołującego zachowuje display więc potrzebujemy jednej kopii kodu na procedurę. Co więcej, jak pokazują następne przykłady kod działający z display'em jest krótszy i szybszy.

Utrzymanie display'a jest bardzo proste. Na wejściu inicjalizującym do procedury musimy najpierw zapisać zawartość tablicy display spod bieżącego lex poziomu a potem przechować wskaźnik do bieżącego rekordu aktywacji do tego samego miejsca. Zakładając ,że nie lokalna zmienna wymaga tylko dwóch instrukcji, jedną do załadowania elementu display do rejestru i drugą dla uzyskania dostępu zmiennej. Poniższy kod implementuje procedury z przykładowymi łączami statycznymi.

```
; Assume Outer is at lex level 1, Middle is at lex level 2, and
; Inner is at lex level 3. Keep in mind that each entry in the
; display is two bytes. Presumably, the variable Display is defined
; in the data segment.

Outer           proc    near
                push    bp
                mov     bp, sp
                push    Display[2]          ;Save current Display Entry
                sub     sp, 2               ;Make room for I.
```

```
                mov     word ptr [bp-4],1       ;Set I to one.
                call    Middle

                add     sp, 2                   ;Remove local variables
                pop     Display[2]              ;Restore previous value.
                pop     bp
                ret
Outer           endp

Middle          proc    near
                push    bp                      ;Save dynamic link.
                mov     bp, sp                  ;Set up our activation
record.
                push    Display[4]              ;Save old Display value.
                sub     sp, 2                   ;Make room for J.

                mov     word ptr [bp-2],2       ;J := 2;
                mov     bx, Display[2]          ;Get static link to prev LL.
                mov     ax, ss:[bx-4]           ;Get I's value.
                add     ax, [bp-2]              ;Add to J and then
                puti                            ; print the sum.
                putcr
                call    Inner

                add     sp, 2                   ;Remnove local variable.
                pop     Display[4]              ;Restore old Display value.
                pop     bp
                ret
Middle          endp

Inner           proc    near
                push    bp                      ;Save dynamic link
                mov     bp, sp                  ;Set up activation record.
                push    Display[6]              ;Save old display value
                sub     sp, 2                   ;Make room for K.

                mov     word ptr [bp-2],2       ;K := 3;
                mov     bx, Display[4]          ;Get static link to prev LL.
                mov     ax, ss:[bx-4]           ;Get J's value.
                add     ax, [bp-2]              ;Add to K

                mov     bx, Display[2]          ;Get ptr to Outer's Act Rec.
                add     ax, ss:[bx-4]           ;Add in I's value and then
                puti                            ; print the sum.
                putcr

                add     sp, 2
                pop     Display [6]
                pop     bp
                ret
Inner           endp
```

Chociaż ten kod nie wygląda szczególnie lepiej niż dawny kod, zastosowanie display'a jest często dużo bardziej wydajne niż zastosowanie łącz statycznych

12.1.6 INSTRUKCJE ENTER I LEAVE 80286.

Kiedy projektowano 80286, projektanci CPU Intela zadecydowali, że dodadzą dwie instrukcje do pomocy przy utrzymaniu display'i. Niestety, chociaż ich praca jest bardzo ogólna i wymaga tylko danych w segmencie

stosu, jest bardzo powolna; dużo wolniejsza niż zastosowanie technik z poprzedniej sekcji. Chociaż wiele nie optymalizujących kompilatorów stosuje te instrukcje, najlepsze kompilatory unikają ich stosowania ,jeśli to możliwe.

Instrukcja leave jest bardzo łatwa do zrozumienia. Wykonuje takie same operacje jak dwie instrukcje:

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

Dlatego też możemy użyć tej instrukcji dla kodu standardowej procedury exit jeśli mamy procesor 80286 lub późniejszy. Na 80386 lub wcześniejszych procesorach instrukcja leave jest krótsza i szybsza niż odpowiadająca jej sekwencja move i pop. Jednak instrukcja leave jest wolniejsza na procesorach 80486 i późniejszych.

Instrukcja enter posiada dwa operandy. Pierwszy jest liczbą bajtów pamięci lokalnej wymaganą przez bieżącą procedurę, drugi jest lex poziomem bieżącej procedury. Instrukcja enter robi co następuje:

```
; ENTER  Locals, LexLevel
        push    bp              ;zachowanie łącza dynamicznego
        mov     tempreg, sp     ;zachowanie na później
        cmp     LexLevel, 0     ;zrobione jeśli jest to lex poziom zero
        je      Lex0
lp:     dec     LexLevel
        jz      Done            ;wyjście jeśli w końcu lex poziom
        sub     bp, 2           ;indeks do display w poprzednim rekordzie aktywacji
        push    [bp]            ;i odłożenie każdego elementu
        jmp     lp              ;powtórzenie dla każdego wejścia
Done:   push    tempreg         ;dodanie wejścia dla bieżącego lex poziomu
Lex0:   mov     bp, tempreg     ;wskaźnik do bieżącego rekordu aktywacji
        sub     sp, Locals      ;alokowanie pamięci lokalnej
```

Jak możemy zobaczyć w tym kodzie, instrukcja enter kopiuje display z rekordu aktywacji do rekordu aktywacji. Może to stać się bardzo kosztowne jeśli zagnieżdżamy procedury na dowolną głębokość. Większość HLLi ,jeśli używa instrukcji enter , zwykle określa poziom zagnieżdżenia zero unikając kopiowania display'a w całym stosie.

Instrukcja enter wkłada wartość dla wejścia display[n[pod lokację BP-(n*2). Instrukcja enter nie kopiuje wartości dla display[0] do każdej ramki stosu. Intel założył, że będziemy trzymać zmienne globalne programu głównego w segmencie danych. Oszczędza to czas i pamięć, nie przeszkadzają kopiowaniu wejścia display[0].

Instrukcja enter jest bardzo wolna, szczególnie na procesorach 80486 i późniejszych. Jeśli rzeczywiście chcemy skopiować display z rekordu aktywacji do rekordu aktywacji jest prawdopodobnie inny sposób ich odłożenia. Poniższy strzępek kodu pokazuje jak to zrobić:

```
; 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.
```

Jeśli wierzyć cyklom Intelowskim ,widzimy, że instrukcja enter nie jest prawie nigdy szybsza niż prosta sekwencja instrukcji, która wykonuje to samo zadanie. Jeśli interesuje nas oszczędność miejsca zamiast pisanie szybkiego kodu, instrukcja enter ogólnie jest lepszą alternatywą. Tak samo, ogólnie rzecz biorąc jest prawdą również dla instrukcji leave. To jest tylko jeden długi bajt, ale jest wolniejszy niż odpowiadające mu instrukcje mov bp, sp i pop bp.

12.2 PRZEKAZYWANIE ZMIENNYCH SPOD RÓŻNYCH LEX POZIOMÓW JAKO PARAMETRY

Uzyskanie dostępu spod różnych lex poziomów w programie o strukturze blokowej wprowadza do programu kilka złożoności. Poprzednia sekcja wprowadziła nas do złożoności przy dostępie do nie lokalnych zmiennych. Problem ten staje się nawet gorszy kiedy próbujemy przekazać takie zmienne jako parametry do innych jednostek programowych. Poniższa podsekcja omawia strategię dla każdego ważnego mechanizmu przekazywania parametrów.

Dla celów tego omówienia, poniższa sekcja będzie zakładała, ze „lokalna" odnosi się do zmiennych w bieżącym rekordzie aktywacji, „globalna" odnosi się do zmiennych w segmencie danych a „pośredni" odnosi się do zmiennych w jakimś rekordzie aktywacji innym niż bieżący rekord aktywacji. Zauważmy że poniższa sekcja nie zakłada, że ds. jest równe ss. Te sekcje również przekazują również wszystkie parametry na stos. Możemy łatwo zmodyfikować szczegóły przekazywania tych parametrów gdzie indziej.

12.2.1 PRZEKAZYWANIE PARAMETRÓW PRZEZ WARTOŚĆ W JĘZYKU O STRUKTURZE BLOKOWEJ

Przekazywanie wartości parametrów do jednostki programowej nie jest trudniejsze niż dostęp do odpowiadających zmiennych; wszystko co musimy zrobić to odłożyć wartość na stos przed wywołaniem skojarzonej procedury.

Przekazując zmienne globalne przez wartość do innej procedury możemy zastosować kod podobny do poniższego:

```
push    GlobalVar       ;zakładamy że GlobalVar jest  w DSEG
call    Procedure
```

Przekazując lokalną zmienną przez wartość do innej procedury możemy użyć takiego kodu:

```
push    [bp-2]          ;lokalna zmienna w bieżącym rekordzie aktywacji
call    Procedure
```

Przekazując zmienną pośrednią jako wartość parametru, musimy najpierw zlokalizować rekord aktywacji tej zmiennej pośredniej a potem odłożyć jego wartość na stos. Dokładny mechanizm jaki zastosujemy zależy od tego czy używamy łącz statycznych lub display'a do śledzenia rekordu aktywacji zmiennej pośredniej. Jeśli stosujemy łącza statyczne możemy zastosować kod podobny do poniższego do przekazania zmiennej z dwóch lex poziomów z bieżącej procedury:

```
mov     bx, [bp+4]      ;zakładamy, że S.L jest pod offsetem 4
mov     bx, ss:[bx+4]   ;przechodzimy dwa łącza statyczne
push    ss:[bx-2]       ;odkładamy wartość zmiennych
call    Procedure
```

Przekazywanie zmiennej pośredniej przez wartość kiedy stosujemy display jest nieco łatwiejsze. Możemy użyć kodu takiego jak poniższy dla przekazania zmiennej pośredniej z lex poziomu jeden:

```
mov     bx, Display[1*2]    ;pobranie wejścia Display[1]
push    ss:[bx-2]           ;odłożenie wartości zmiennej
```

```
call    Procedure
```

12.2.2 PRZEKAZYWANIE PARAMETRÓW PRZEZ REFERENCJĘ,WARTOŚĆ I WARTOŚĆ-WYNIK W JĘZYKU O STRUKTURZE BLOKOWEJ

Mechanizmy przekazywania przez referencję, wynik i wartość -wynik ogólnie przekazują adres parametru na stos. Jeśli globalna zmienna rezyduje w segmencie danych, wszystkie rekordy aktywacji istnieją w segmencie stosu a ds ≠ ss wtedy musimy przekazać daleki wskaźnik dla uzyskania dostępu do wszystkich możliwych zmiennych.

Przekazując daleki wskaźnik musimy odłożyć wartość segmentu występującą przed wartością offsetu na stos. Dla globalnych zmiennych, wartość segmentu znajduje się w rejestrze ds.; dla wartości nie globalnych, ss zawiera wartość segmentu Dla obliczenia offsetowej części adresu normalnie użylibyśmy instrukcji lea. Poniższa sekwencja kodu przekazuje zmienne globalne przez referencję:

```
push    ds.                 ;odkładamy najpierw adres segmentu
lea     ax, GlobalVar       ;obliczanie offsetu
push    ax                  ;odłożenie offsetu GlobalVar
```

Zmienne globalne są specjalnym przypadkiem ponieważ asembler może obliczyć ich czas wykonania podczas czasu asemblacji. Dlatego też, tylko dla skalarnych zmiennych globalnych, możemy skrócić powyższą sekwencję kodu do:

```
push    ds                  ;odłożenie adresu segmentu
push    offset GlobalVar    ;odłożenie części offsetu
call    Procedure
```

Dla przekazania zmiennej lokalnej poprzez referencję nasz kod musi najpierw odłożyć wartość ss na stos a potem odłożyć offset. Ten offset jest offsetem zmiennej wewnątrz segmentu stosu, nie offsetem wewnątrz rekordu aktywacji! Poniższy kod przekazuje adres lokalnej zmiennej poprzez referencję:

```
push    ss                  ;odłożenie adresu segmentowego
lea     ax, [bp-2]          ;obliczenie offsetu lokalnej zmiennej
push    ax                  ;i odłożenie go
call    Procedure
```

Dla przekazania zmiennej pośredniej poprzez referencję musimy najpierw zlokalizować rekord aktywacji zawierający zmienną żeby obliczyć adres efektywny w segmencie stosu. Kiedy stosujemy łącza statyczne, kod przekazujący adres parametru może wyglądać następująco:

```
push    ss                  ;odłożenie części segmentowej
mov     bx, [bp+4]          ;Zakładamy ,że S.L jest pod offsetem 4
mov     bx, ss:[bx+4]       ;przechodzimy dwa łącza statyczne
lea     ax, [bx-2]          ;obliczanie adresu efektywnego
push    ax                  ;odłożenie części offsetowej
call    Procedure
```

Kiedy stosujemy display, sekwencja wywołująca mogłaby wyglądać następująco:

```
push    ss                  ;odłożenie części segmentowej
mov     bx, Display[1*2]    ;pobranie wejścia Display[1]
lea     ax, [bx-2]          ;pobranie offsetu zmiennej
push    ax                  ;o odłożenie go
call    Procedure
```

Jak możemy sobie przypomnieć z poprzedniego rozdziału, są dwa sposoby przekazywania parametrów przez wartość-wynik. Możemy odłożyć wartość na stos a potem, kiedy wracamy z procedury, zdejmujemy tą wartość ze stosu i przekazujemy z powrotem do zmiennej. To jest właśnie specjalny przypadek mechanizmu przekazywania przez wartość opisanego w poprzedniej sekcji.

12.2.3 PRZEKAZYWANIE PARAMETRÓW PRZEZ NAZWE I LENIWE WARTOŚCIOWANIE W JĘZYKU O BLOKOWEJ STRUKTURZE

Ponieważ przekazujemy adres thunka kiedy przekazujemy parametry przez nazwę lub leniwe wartościowanie, obecność globalnych, pośrednich i lokalnych zmiennych nie wpływa na sekwencję wywołującą procedury. Zamiast tego, thunk musi zająć się rozróżnianiem lokacji tych zmiennych. Poniższe przykłady przedstawia thunki, które mogą łatwo zmodyfikować te thunki dla parametrów leniwego wartościowania.

Największy problem thunk ma z lokalizacją rekordu aktywacji zawierającego zmienną której adres zwraca. W ostatnim rozdziale nie było to zbyt dużym problemem ponieważ zmienne istniały albo w bieżącym rekordzie

aktywacji albo globalnej przestrzeni danych. W obecności zmiennych pośrednich zadanie to staje się co nieco bardziej złożone. Najłatwiejszym rozwiązaniem jest przekazanie dwóch wskaźników kiedy przekazujemy zmienną przez nazwę. Pierwszy wskaźnik powinien być adresem thunka, drugi wskaźnik powinien być offsetem rekordu aktywacji zawierającego zmienną do której thunk musi uzyskać dostęp. Kiedy procedura wywołuje thunka, musi przekazać ten offset rekordu aktywacji jako parametr do thunka. Rozpatrzmy poniższą procedurę Panacei:

```
TestThunk: procedure (name item: integer; var j : integer);
begin TestThunk;

        for j in 0..9 do item := 0;
end TestThunk;

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

Kod asemblerowy dla powyższego kodu mógłby wyglądać tak:

```
; TestThunk AR:
;
;       BP+10-    Address of thunk


;       BP+8-     Ptr to AR for Item and J parameters (must be in the same AR).
;       BP+4-     Far ptr to J.

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

                les     bx, [bp+4]              ;Get ptr to J.
                mov     word ptr es:[bx], 0     ;J := 0;
ForLoop:        cmp     word ptr es:[bx], 9     ;Is J > 9?
                ja      ForDone
                push    [bp+8]                  ;Push AR passed by caller.
                call    word ptr [bp+10]        ;Call the thunk.
                mov     word ptr ss:[bx], 0     ;Thunk returns adrs in BX.
                les     bx, [bp+4]              ;Get ptr to J.
                inc     word ptr es:[bx]        ;Add one to it.
                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          ;Make room for locals.
```

```
                jmp     OverThunk
Thunk           proc
                push    bp
                mov     bp, sp
                mov     bp, [bp+4]    ;Get AR address.
                mov     ax, [bp-22]   ;Get I's value.
                add     ax, ax        ;Double, since A is a word array.
                add     bx, -20       ;Offset to start of A
                add     bx, ax        ;Compute address of A[I] and
                pop     bp            ; return it in BX.
                ret     2             ;Remove parameter from stack.
Thunk           endp

OverThunk:      push    offset Thunk ;Push (near) address of thunk
                push    bp           ;Push ptr to A/I's AR for thunk
                push    ss           ;Push address of I onto stack.
                lea     ax, [bp-22]  ; Offset portion of I.
                push    ax
                call    TestThunk
                mov     sp, bp
                ret
CallThunk       endp
```

12.3 PRZEKAZYWANIE PARAMETRÓW JAKO PARAMETRÓW DO INNYCH PROCEDUR

Kiedy procedura przekazuje jeden ze swoich własnych parametrów jako parametr do innej procedury, rozwiną się pewne problemy, które nie istniały kiedy przekazywaliśmy zmienne jako parametry. Istotnie, w pewnych (rzadkich) przypadkach jest logicznie niemożliwe przekazanie jakiegoś typu parametru do innej procedury. Sekcja ta zajmuje się problemami przekazywania parametrów jednej procedury do innej procedury. Przekazywanie parametrów przez wartość zasadniczo niczym się nie różni niż zmiennych lokalnych . wszystkie techniki z poprzedniej sekcji mają zastosowanie do przekazywaniu parametrów przez wartość. Poniższa sekcja zajmuje się przypadkami gdzie wywoływana procedura przekazuje parametr przekazany do niej przez referencję, wartość-wynik, wynik i leniwe wartościowanie.

12.3.1 PRZEKAZANIE PARAMETRÓW REFERENCYJNYCH DO INNYCH PROCEDUR

Przekazywanie parametrów referencyjnych do innej procedury jest tam gdzie zaczyna się złożoność. Rozważmy następującą (pseudo) Pascalowski szkielet procedury:

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

        procedure ToProc (???? Parm:integer);
        begin
        -
        -
        -
        end;
begin {HasRef}
        -
        -
        -
        TpProc(refParm);
        -
        -
        -
end;
```

„????” w liście parametrów ToProc wskazuje, że wypełnimy we właściwym mechanizmie przekazywania parametrów jak gwarantuje omówienie

Jeśli ToProc wymaga przekazania parametrów przez wartość (???? Jest pustym ciągiem), wtedy HasRef musi pobrać wartość parametru refparm i przekazać tą wartość do ToProc. Wykonuje to poniższy kod:

```
        les     bx, [bp+4]          ;pobranie adresu refparm
        push    es:[bx]             ;odłożenie liczby całkowitej wskazującej na refparm
        call    ToProc
```

Przekazanie parametrów referencyjnych poprzez referencję, wartość-wynik lub wynik jest łatwe - wystarczy skopiować parametr wywołujący na stos. To znaczy, jeśli parametr parm w ToProc jest parametrem referencyjnym, parametrem wartość-wynik lub parametrem wyniku, zastosujemy poniższą sekwencję wywołującą:

```
        push    [bp+6]              ;odłożenie części segmentowej ref param
        push    [bp+4]              ;odłożenie części offsetowej ref parm
        call    ToProc
```

Przekazanie parametrów referencyjnych przez nazwę jest dosyć proste. Piszemy thunka, który chwyta adres parametru referencyjnego i zwraca tą wartość. W powyższym przykładzie wywołanie ToProc może wyglądać tak jak poniżej:

```
            jmp     SkipThunk
Thunk0      proc    near
            les     bx, [bp+4]          ;zakładamy, że BP wskazuje AR HasRef
            ret
Thunk0      endp

SkipTunk    push    offset Thunk0       ;adres thunka
            push    bp
            call    ToProc
```

Wewnątrz ToProc, referencja do parametru może wyglądać jak następuje:

```
        push    bp                  ;zachowanie wskaźnika do rekordu aktywacji
        mov     bp, [bp+4]          ;wskaźnik do rekordu aktywacji Parm
        call    near  ptr [bp+6]    ;wywołanie thunka
        pop     bp                  ;odzyskanie naszego wskaźnika do AR
        mov     ax, es:[bx]         ;dostęp do zmiennej
        -
        -
        -
```

Przekazanie parametru referencyjnego przez leniwe wartościowanie jest bardzo podobne do przekazania go przez nazwę. Jedyna różnica (w sekwencji wywołania ToProc) jest taka, że thunk musi zwrócić wartość zmiennej zamiast jej adresu. Możemy łatwo wykonać to z poniższym thunkiem:

```
Thunk1  proc    near
        push    es
        push    bx
        les     bx, [bp+4]          ;zakładamy, że BP wskazuje AR HasRef
        mov     ax, es:[bx]         ,zwracana wartość ref parm w ax
        pop     bx
        pop     es
        ret
Thunk1  endp
```

12.3.2 PRZEKAZANIE PARAMETRÓW WARTOŚĆ-WYNIK I WYNIK JAKO PARAMETRÓW

Zakładając ,że stworzyliśmy zmienną lokalną która przechowuje wartość parametru wartość-wynik lub wynik, przekazujemy jeden z tych parametrów do innej procedury nie różni się od przekazywania wartości parametru do innego kodu. Ponieważ procedura robi lokalną kopię parametru wartość-wynik lub alokuje pamięć dla parametru wyniku, możemy traktować zmienną podobnie jak parametr wartość lub zmienną lokalną pod względem przekazywania jej do innej procedury.

Oczywiście, nie ma sensu stosować wartości parametru wynik do przechowywania wartości w pamięci lokalnego

parametru. Dlatego też, uważajmy kiedy przekazujemy parametry wynikowe do innych procedur aby zainicjalizować parametr wyniku przed użyciem jego wartości.

12.3.3 PRZEKAZYWANIE PARAMETRÓW NAZW DO INNYCH PROCEDUR

Ponieważ thunk przekazywania parametrów przez nazwę zwraca adres parametru, przekazywanie parametrów nazw do innej procedury jest bardzo podobne do przekazywania parametrów referencyjnych do innej procedury. Podstawowa różnica występuje kiedy przekazujemy parametr jako parametr nazwy.

Kiedy przekazujemy parametr nazwy jako parametr wartości, po pierwsze wywołujemy thunk, wyłuskujemy adres jaki zwraca thunk a potem przekazujemy wartość do nowej procedury. Poniższy kod demonstruje takie wywołanie kiedy thunk zwraca adres zmiennej w es:bx (zakładając przekazanie parametru przez nazwę wskaźnik AR jest pod adresem bp+4 a wskaźnik do thunka jest pod adresem bp+6):

```
push    bp                      ;Save our AR ptr.
mov     bp, [bp+4]              ;Ptr to Parm's AR.
call    near ptr [bp+6]         ;Call the thunk.
push    word ptr es:[bx]        ;Push parameter's value.
pop     bp                      ;Retrieve our AR ptr.
call    ToProc                  ;Call the procedure.
  .
  .
  .
```

Przekazywanie parametru nazw do innej procedury przez referencję jest bardzo proste. Wszystko co musimy zrobić to odłożyć adres zwracanego thunka na stos. Poniższy kod, wykonujący to jest bardzo podobny do powyższego:

```
push    bp                      ;Save our AR ptr.
mov     bp, [bp+4]              ;Ptr to Parm's AR.
call    near ptr [bp+6]         ;Call the thunk.
pop     bp                      ;Retrieve our AR ptr.
push    es                      ;Push seg portion of adrs.
push    bx                      ;Push offset portion of adrs.
call    ToProc                  ;Call the procedure.
  .
  .
```

Przekazanie parametru nazw do innej procedury jako przekazanie przez parametr nazwy jest bardzo łatwe; wszystko co trzeba zrobić to przekazać thunk (i powiązane wskaźniki) do nowej procedury. Wykonuje to poniższy kod:

```
push    [bp+6]          ;przekazanie adresu thunka
push    [bp+4]          ;przekazanie  adresu AR thunka
call    ToProc
```

Aby przekazać parametr nazwy do innej procedury przez leniwe wartościowanie musimy stworzyć thunk dla parametru leniwego wartościowania, który wywołuje thunk przekazywania parametru przez nazwę, usuwając wskaźnik i potem zwracając tą wartość.

12.3.4 PRZEKAZYWANIE PARAMETRÓW LENIWEGO WARTOSCIOWANIA JAKO PARAMETRÓW

Parametry leniwego wartościowania typowo składają się z trzech części: adresu thunku, lokacji dla trzymania wartości powrotu thunka i zmiennej boolowskiej, która określa czy procedura musi wywołać thunka dla pobrania wartości parametru lub czy może po prostu użyć wartości wcześniej zwróconej przez thunk. Kiedy przekazujemy parametr przez leniwe wartościowanie do innej procedury, kod wywołujący musi najpierw sprawdzić zmienną boolowską aby zobaczyć czy wartość pola jest prawidłowa. Jeśli pole boolowskie jest prawdą, kod wywołujący może po prostu zastosować dane w polu wartości. W innym przypadku, ponieważ pole wartości ma dane, przekazanie tej danej do innej procedury nie różni się od przekazania zmiennej lokalnej lub parametru wartości do innej procedury.

12.3.5 PODSUMOWANIE PRZEKAZYWANIA PARAMETRÓW

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Value	Pass the value	Pass address of the value parameter	Pass address of the value parameter	Pass address of the value parameter	Create a thunk that returns the address of the value parameter	Create a thunk that returns the value
Reference	Dereference parameter and pass the value it points at	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Create a thunk that passes the address (value of the reference parameter)	Create a thunk that deferences the reference parameter and returns its value
Value-Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the value-result parameter	Create a thunk that returns the value in the local value of the value-result parameter
Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the result parameter	Create a thunk that returns the value in the local value of the result parameter

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Name	Call the thunk, dereference the pointer, and pass the value at the address the thunk returns	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Pass the address of the thunk and any other values associated with the name parameter	Write a thunk that calls the name parameter's thunk, dereferences the address it returns, and then returns the value at that address
Lazy Evaluation	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the local value as the value parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Create a thunk that returns the address of the Lazy Eval's value field	Create a thunk that checks the boolean field of the caller's Lazy Eval parameter. It should call the corresponding thunk if this variable is false. It should set the boolean field to true and then return the data in the value field

Tablica 48 Przekazywanie parametrów jako parametrów do innych procedur

12.4 PRZEKAZYWANIE PROCEDUR JAKO PARAMETRÓW

Wiele języków programowania pozwala nam przekazać nazwę procedury lub funkcji jako parametr, To pozwala kodowi wywołującemu przekazać różne działania dla wykonania wewnątrz procedury. Klasycznym przykładem jest procedura plot, która rysuje wykres jakiejś ogólnej funkcji matematycznej przekazywanej jako parametr do plot.

Pascal Standardowy pozwala nam przekazać procedury i funkcje poprzez deklarowanie ich jak następuje:

```
procedure DoCall (procedure x);
begin
        x;
end;
```

Instrukcja DoCall(x,y,z); wywołuje DoCall, która po kolei wywołuje procedurę xyz.

Przekazywanie procedury lub funkcji jako parametru może wydawać się łatwym zadaniem - przekazanie adresu funkcji lub procedury demonstruje poniższy przykład:

```
procedure PassMe;
begin
        writeln ('Pass me została wywołana');
end;
procedure CallPassMe (procedure x);
begin
        x;
end;
begin {main}
        CallPassMe(PassMe);
end.
```

Kod 8086 implementujący powyższe mógłby wyglądać tak:

```
PassMe          proc    near
                print
                byte    „PassMe została wywołana" ,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              ;przekazanie adresu PassMe do CallPassMe
                push    bx
                call    CallPassMe
                exitPgm
Main            endp
```

Dla przykładu tak prostego jak powyższy, ta technika pracuje dobrze. Jednakże nie zawsze pracuje poprawnie jeśli PassMe musi uzyskać dostęp do nie lokalnych zmiennych. Poniższy kod Pascalowski demonstruje problem, który mógłby wystąpić:

```
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 = 10 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(%, dummy);
end.
```

Kod ten tworzy poniższą sekwencję wywołań:

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 będzie drukowało wartość Recurse1 zmiennej i do standardowego wyjścia. Jednakże jest kilka rekordów aktywacji obecnych na stosie które podnoszą oczywiste pytanie ,"którą kopię i Print wyświetli?" Bez podania tego, po długim namyśle możemy dojść do wniosku, że powinniśmy drukować wartość „1" ponieważ Recurse2 wywołuje Print kiedy wartość Recurse1 dla i to jeden. Zauważmy, że kiedy Recurse2 przekazuje adres Print do Recurse1, wartość i to cztery. Pascal , podobnie jak większość języków o strukturze blokowej, będzie stosował wartość i w momencie przekazywania przez Recurse2 adresu Print do Recurse1.W związku z tym, powyższy kod powinien drukować wartość cztery a nie wartość jeden.

To stwarza różne implementacje problemu. W końcu Print nie może po prostu uzyskać dostępu do display'a aby skorzystać z dostępu do zmiennej globalnej i - wejście do display'a dla Recurse1 wskazuje ostatnią kopię rekordu aktywacji Recurse1, , nie wejścia zawierającego wartość cztery , która jest tym czego chcemy.

Większość popularnych rozwiązań w systemowym zastosowaniu display'a jest zrobienie lokalnej kopii każdego display'a kiedykolwiek wywołujemy procedurę lub funkcję. Kiedy przekazujemy procedurę lub funkcję jako parametr, kod wywołujący kopiuje display wraz z adresem procedury lub funkcji. Jest tak dlatego, że Intelowska instrukcja enter robi kopię display'a kiedy budujemy rekord aktywacji.

Jeśli przekazujemy funkcję albo procedurę jak parametry, możemy rozważyć zastosowanie łącza statycznego zamiast display'a. Kiedy stosujemy łącze statyczne musimy tylko przekazać pojedynczy wskaźnik (łącze statyczne) wraz z adresem podprogramu. Oczywiście jest więcej pracy przy dostępie do nie lokalnych zmiennych, ale musimy kopiować display'a przy każdym wywołaniu, co jest bardzo kosztowne.

Poniższy kod 80x86 dostarcza implementacji powyższego kodu przy zastosowaniu łącz statycznych:

```
wp              textequ   <word ptr>
Dummy           proc      near
                ret
Dummy           endp
; PrintIt; (Use the name PrintIt to avoid conflict).
;
;       stack:
;
;       bp+4:     static link.
PrintIt         proc      near
                push      bp
                mov       bp, sp
                mov       bx, [bp+4]              ;Get static link
                mov       ax, ss:[bx-10]          ;Get i's value.
                puti
                pop       bp
                ret       2
PrintIt         endp
; Recurse1(i:integer; procedure x);
;
;       stack:
;
;       bp+10:    i
;       bp+8:     x's static link
;       bp+6: x's address
```

```
Recurse1        proc    near
                push    bp
                mov     bp, sp
                push    wp [bp+10]          ;Push value of i onto stack.
                push    wp [bp+8]           ;Push x's static link.
                push    wp [bp+6]           ;Push x's address.
                push    bp                  ;Push Recurse1's static link.
                call    Recurse1
                pop     bp
                ret     6
Recurse1        endp

; Recurse2(i:integer; procedure y);
;
;       stack:
;
;       bp+10: j
;       bp+8:  y's static link.


;       bp+6:  y's address.
;       bp+4:  Recurse2's static link.

Recurse2        proc    near
                push    bp
                mov     bp, sp
                cmp     wp [bp+10], 1       ;Is j=1?
                jne     TryJeq5
                push    [bp+8]              ;y's static link.
                call    wp [bp+6]           ;Call y.
                jmp     R2Done

TryJeq5:        cmp     wp [bp+10], 5       ;Is j=5?
                jne     Call1
                mov     ax, [bp+10]
                dec     ax
                push    ax
                push    [bp+4]              ;Push static link to R1.
                lea     ax, PrintIt         ;Push address of print.
                push    ax
                call    Recurse1
                jmp     R2Done

Call1:          mov     ax, [bp+10]
                dec     ax
                push    ax
                push    [bp+8]              ;Pass along existing
                push    [bp+6]              ; address and link.
                call    Recurse1

R2Done:         pop     bp
                ret     6
Recurse1        endp
```

```
main            proc
                push    bp
                mov     bp, sp
                mov     ax, 5                   ;Push first parameter.
                push    ax
                push    bp                      ;Dummy static link.
                lea     ax, Dummy               ;Push address of dummy.
                push    ax
                call    Recurse1
                pop     bp
                ExitPgm
main            endp
```

Jest kilka sposobów poprawy tego kodu. Oczywiście, ten szczególny program w rzeczywistości nie potrzebuje pielęgnacji display'a lub łącza statycznego ponieważ tylko PrintIt uzyskuje dostęp do nie lokalnych zmiennych; jednakże zignorujemy ten fakt na razie i pominiemy to. Ponieważ wiemy, że tylko PrintIt uzyskuje dostęp do zmiennych przy określonym lex poziomie i tylko program wywołuje PrintIt pośrednio, możemy przekazać wskaźnik do właściwego rekordu aktywacji; to jest to co powyższy kod robi, chociaż zachowuje również pełne łącza statyczne. Kompilatory muszą zawsze zakładać najgorszy przypadek i często generować nieefektywny kod. Jeśli przestudiujemy swoje potrzeby, możemy poprawić efektywność naszego kodu poprzez unikanie dużych kosztów pielęgnowania łączy statycznych lub kopiowania display'a.

Zapamiętajmy, że thunki są specjalnymi przypadkami funkcji, które możemy wywołać pośrednio

Cierpimy na ten same problemy i wady przy procedurach i funkcjach jako parametrach w związku z dostępem do nie lokalnych zmiennych. Jeśli taki podprogram uzyskuje dostęp do nie lokalnych zmiennych (a thunki prawie zawsze) wtedy musimy zachować ostrożność kiedy wywołujemy taki podprogram. Na szczęście thunki nigdy nie powodują pośredniej rekurencji(która jest odpowiedzialna za problemy w przykładzie Recurse1 /Recurse1) więc możemy użyć display'a do uzyskania dostępu do nie lokalnych zmiennych pojawiających się wewnątrz thunka.

12.5 ITERATORY

Iterator jest czymś pomiędzy strukturą sterującą a funkcją. Chociaż popularne języki wysokiego poziomu niezbyt często wspierają iteratory, są one obecne w wielu językach wysokiego poziomu. Iteratory dostarczają połączenia mechanizmu stanu maszynowego / wywołania funkcji. Iteratory są również częścią pętli struktury sterującej, iterator dostarcza wartości dla zmiennej sterowania pętli na każdą iterację.

Aby zrozumieć co to jest iterator rozważmy poniższą pascalowską pętlę for:

```
for I := 1 to 10 do <jakieś instrukcje>
```

Kiedy uczyliście się Pascala, prawdopodobnie myśleliście, że ta instrukcja inicjalizuje i jedynką, porównuje i z 10 i wykonuje instrukcje jeśli i jest mniejsze niż lub równe 10. Po wykonaniu instrukcji, instrukcja for zwiększa i i porównuje go z 10 ponownie., powtarzając ten proces szereg razy dopóki I nie będzie większe niż 10.

Podczas gdy ten opis jest semantycznie poprawny ,i rzeczywiście jest to sposób w jaki większość kompilatorów Pascala implementuje pętlę for, nie jest to jedyny punkt widzenia, który opisuje jak działa pętla for. Przypuśćmy, zamiast tego , że potraktujemy słowo zastrzeżone „to" jako operator. Operator który oczekuje dwóch parametrów (jeden i dziesięć w tym przypadku) i zwraca zakres wartości po każdym wykonaniu. To znaczy, przy pierwszym wywołaniu operator „to" zwróciłby jeden, po drugim dwa i tak dalej. Po dziesiątym wywołaniu operator „to" będzie niepoprawny i zakończy pętlę. Jest to dokładnie opis iteratora

Generalnie rzecz biorąc, iterator steruje pętla Różne języki używają różnych nazw dla pętli sterowanych iteratorem, ten tekst będzie używał nazwy foreach jak następuje:

```
foreach  zmienna in iterator() do
        instrukcje;
endfor;
```

Zmienna jest zmienną której typ jest kompatybilny z typem zwracanym przez iterator. Iterator zwraca dwie wartości: boolowskie sukces lub porażka i wynik funkcji. Tak długo jak iterator zwraca sukces, instrukcja foreach przydziela inną wartość zwracaną do zmiennej i wykonuje instrukcje. Jeśli iterator zwraca porażka, pętla foreach się kończy i wykonuje następne instrukcje sekwencyjne po treści pętli foreach. W przypadku porażki, instrukcja foreach nie wpływa na wartość zmiennej.

Iteratory są dużo bardziej złożone niż normalne funkcje. Typowa funkcja wywołuje dwie podstawowe operacje: wywołanie i powrót. Wezwanie iteratora wywołuje cztery podstawowe operacje:

1) Pierwsze wywołanie iteratora
2) Dostarczenie wartości
3) Wznowienie iteratora
4) Zakończenie iteratora

Aby zrozumieć jak działa operator, rozważmy poniższy krótki przykład z języka programowania Panacea:

```
Range: iterator (start, stop : integer): integer;
begin range;
        while  (start <= stop) do
                yield start;
                start  := start +1;
        endwhile;

end Range;
```

W języku Panacea iterator wywołany może pojawić się w instrukcji foreach Za wyjątkiem powyższej instrukcji yield,, każda dobrze znana z Pascala lub C++ powinna móc decydować o podstawowej logice tego iteratora

Iterator w Panacei może wracać do swojego kodu wywołującego stosując jeden lub dwa oddzielne mechanizmy, może wracać do kodu wywołującego poprzez wyjście przez end Range; instrukcje lub może zwrócić wartość poprzez wykonanie instrukcji yield. Iterator powiedzie się jeśli wykonamy instrukcję yield, nie powiedzie się jeśli po prostu wróci do kodu wywołującego. Dlatego też, instrukcja foreach będzie tylko wykonywała odpowiednią instrukcję jeśli wyjdziemy z iteratora przez yield. Instrukcja foreach zakończy się jeśli po prostu wrócimy z iteratora. W powyższym przykładzie, iterator zwraca wartość start.. stop poprzez yield a potem iterator kończy się. Pętla

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

jest porównywalna do instrukcji pascalowskiej

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

Kiedy program Panacea wykonuje po raz pierwszy instrukcję foreach, czyni początkowe wywołanie iteratora. Iterator działa dopóki wykonuje yield lub zwraca. Jeśli wykonuje instrukcję yield zwraca wartość wyrażenia następującego po yield jako wynik iteratora i zawraca sukces. Jeśli po prostu zwraca, iterator zwraca porażkę i żadnego wyniku iteratora. W bieżącym przykładzie przy początkowym wywołaniu iterator zwraca sukces i wartość jeden.

Zakładając pomyślny powrót (jak w bieżącym przykładzie), instrukcja foreach przypisuje wartość powrotną iteratora do zmiennej sterującej pętli i wykonuje treść pętli foreach. Po wykonaniu treści pętli, instrukcja foreach wywołuje ponownie iterator. Jednakże, tym razem instrukcja foreach wznawia iterator zamiast czynić początkowe wywołanie. Iterator wznawia kontynuowanie z pierwszą instrukcją po ostatnim wykonanym yieldzie. W przykładzie range wykonywanie będzie kontynuowane od instrukcji start := start + 1; Przy pierwszym wznowieniu, iterator Range dodaje jeden do start, tworząc wartość dwa. Dwa jest mniejsze niż dziesięć (wartość stop) wiec pętla while będzie powtórzona a iterator dostarczy wartość dwa. Proces ten będzie powtarzany tak długo dopóki iterator dostarczy dziesięć. Po wznowieniu po dostarczeniu dziesięć, iterator zwiększy start do jedenastu i zwróci, zamiast dostarczyć, ponieważ ta nowa wartość nie jest mniejsza lub równa dziesięć. Kiedy iterator range zwróci (błąd), pętla foreach zakończy się.

12.5.1 IMPLEMENTOWANIE ITERATORÓW PRZY ZASTOSOWANIU ROZWINIĘCIA IN-LINE

Implementacja iteratora jest raczej złożona. Przede wszystkim rozważmy pierwszą próbę asemblerowej implementacji powyższej instrukcji foreach:

```
push    1               ;zakładamy 286 lub lepszy
push    10              ;a parametry przekazujemy na stos
```

```
                call    Range_Initial   ;robimy początkowe wywołanie iteratora
                jc      Failure         ;C=0, 1 znaczy sukces, błąd
ForLoop:        puti                    ;zakładamy, że wynik jest w AX
                call    Range_Resume    ;wznowienie iteratora
                jnc     ForLoop         ;wyzerowane przeniesienie to sukces!
Failure:
```

Chociaż wygląda to jak prosto zaimplementowany projekt, jest kilka kwestii do rozważenia. Po pierwsze, wywołanie Range_Resume wygląda dosyć prosto, ale nie ma stałego adresu, który adres wznowienia. Chociaż jest to prawdą, że ten przykład Range ma tylko jeden adres wznowienia, generalnie możemy mieć tak dużo instrukcji yield jak jest w iteratorze. Na przykład poniższy iterator zwraca wartości 1,2,3 i 4:

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

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

end OneToFour;
```

Początkowe wywołanie wykona instrukcję yield 1l Pierwsze wznowienie wykona instrukcje yield 2, drugie wznowienie wykona instrukcje yield 3; itd. Oczywiście nie ma pojedynczego adresu wznowienia, który może wyliczyć kod wywołujący.

Jest parę dodatkowych szczegółów do rozpatrzenia. Po pierwsze iterator może wywołać procedurę lub funkcję. Jeśli tak procedura lub funkcja wykonuje instrukcję yield wtedy wznowienie przez instrukcję foreach kontynuuje wykonywanie wewnątrz procedury lub funkcji która wykonywała yield. Po drugie, semantyka iteratora wymaga zachowania wartości wszystkich lokalnych zmiennych i parametrów aż do zakończenia iteratora. To znaczy, zwracanie wartości nie dealokuje zmiennych lokalnych i parametrów .Podobnie, każdy adres wznowienia opuszczający stos (na przykład wywołanie procedury lub funkcji, które wykonują instrukcję yield) nie może być zagubione kiedy kawałek kodu zwraca wartość a odpowiednia instrukcja foreach wznawia iterator. Generalnie, znaczy to ,że nie możemy zastosować standardowej sekwencji wywołania i powrotu do yield lub wznowienia iteratora ponieważ musimy zachować zawartość stosu.

Podczas gdy jest kilka sposobów implementacji iteratorów w asemblerze, być może najpraktyczniejszą metodą jest mieć wywołanie iteratorem sterowania pętlą poprzez iterator i powrót pętli do iteratora funkcją.

Niektóre języki wysokiego poziomu wspierają iteratory. Na przykład Metaware's Professional Pascal Compiler for PC wspiera iteratory. Stworzy on sekwencję kodu podobną do poniższej

```
iterator OneToFour: integer
begin
        yield 1;
        yield  2;
        yield 3;
        yield 4;
end;
```

i wywoła ją w programie głównym jak następuje:

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

Professional Pascal kompletnie przestawi nasz kod. Zamiast tego stworzymy funkcję asemblerową i wywołamy tą funkcję z wnętrza treści pętli, kod przekaże treść pętli for do funkcji, rozszerzając iterator in-line (podobnie jak makro) i wywołuje treść pętli for funkcji przy każdym yield. To znaczy Professional Pascal prawdopodobnie stworzy kod asemblerowy podobny do tego:

```
; The following procedure corresponds to the for loop body
; with a single parameter (I) corresponding to the loop
; control variable:

ForLoopCode     proc    near
                push    bp
                mov     bp, sp
                mov     ax, [bp+4]   ;Get loop control value and
                puti                 ; print it.
                putcr
                pop     bp
                ret     2            ;Pop loop control value off stk.
ForLoopCode     endp

; The follow code would be emitted in-line upon encountering the
; for loop in the main program, it corresponds to an in-line
; expansion of the iterator as though it were a macro,
; substituting a call for the yield instructions:

                push    1            ;On 286 and later processors only.
                call    ForLoopCode
                push    2
                call    ForLoopCode
                push    3
                call    ForLoopCode
                push    4
                call    ForLoopCode
```

Metoda dla implementacji iteratorów jest dogodna i tworzy stosunkowo wydajny (szybki) kod .Robi to, jednakże, jest parę wad. Po pierwsze ponieważ musimy poszerzyć iterator in-line gdziekolwiek go wywołujemy, bardziej niż makro, nasz program może zwiększyć się znacznie jeśli iterator nie jest krótki i stosujemy go często. Po drugie, metoda implementacji iteratora kompletnie skrywa odpowiednią logikę kodu i czyni nasz program trudnym do odczytu i zrozumienia.

12.5.2 IMPLEMENTACJA ITERATORÓW RAMKAMI WZNOWIEŃ

Poszerzanie in-line nie jest jedynym sposobem implementacji iteratorów. Jest inna metoda, która zachowuje strukturę naszego programu kosztem odrobinę bardziej złożonej implementacji. Kilka języków wysokiego poziomu, w tym Icon i CLU, używają tej implementacji.

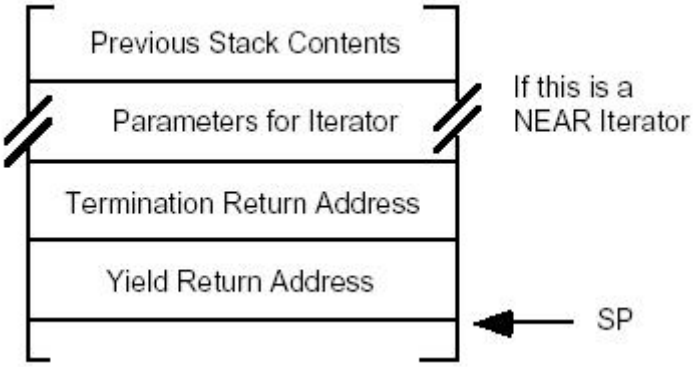
Po pierwsze potrzebujemy innej ramki stosu: ramki wznowień. Ramka wznowień zawiera dwa wejścia; adres powrotu yield (to znaczy adres następnej instrukcji po instrukcji yield) i łącze dynamiczne, które jest wskaźnikiem do rekordu aktywacji iteratora. Zazwyczaj łącze dynamiczne jest wartością rejestru bp w czasie wykonywania instrukcji yield. Wersja ta implementuje cztery części iteratora jak następuje:

1) Wywołanie instrukcji dla początkowego wywołania iteratora
2) Wywołanie instrukcji dla instrukcji yield
3) Instrukcja ret dla wznowienia działania, i
4) Instrukcja ret dla zakończenia iteratora

Przede wszystkim iterator wymaga dwóch adresów powrotu zamiast jednego, jaki jest normalnie oczekiwany. Pierwszy adres powrotny jaki pojawia się na stosie jest adresem powrotnym zakończenia. Drugi adres powrotny jest tam gdzie podprogram [przekazuje sterowanie do operacji yield. Kod wywołujący musi odłożyć te dwa adresy powrotne przy początkowym wywołaniu iteratora.. Stos, na początkowym wejściu do iteratora powinien wyglądać jak na rysunku 12.8

Jako przykład rozważmy iterator Range przedstawiany wcześniej. Iterator ten wymaga dwóch parametrów, wartości początkowej i wartości końcowej:

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



Rysunek 12.8: Rekord aktywacji iteratora

Kod który robi początkowe wywołanie iteratora Range, tworzący stos jak ten powyżej, może być następujący:

```
push    1                   ;odłożenie wartości parametru start
push    10                  ;odłożenie wartości parametru stop
push    offset ForDone      ;odłożenie adresu zakończenia
call    Range
```

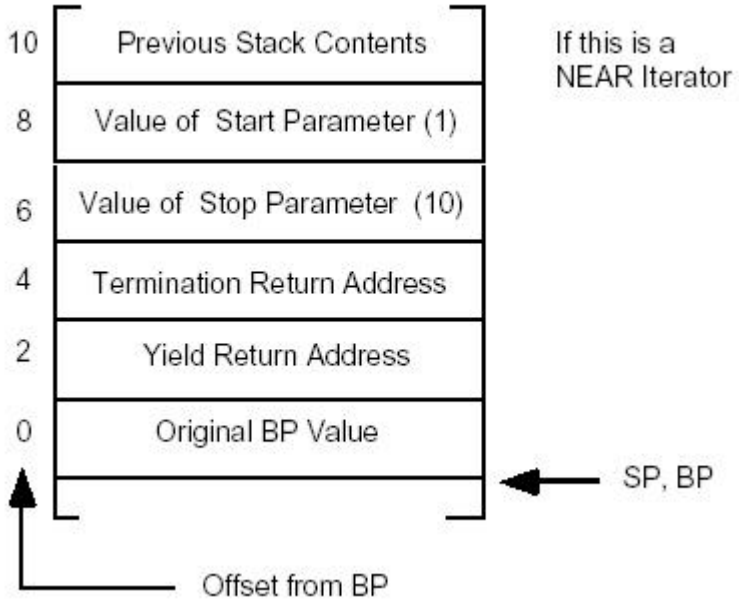
ForDone jest pierwszą instrukcją bezpośrednio następującą po pętli foreach, to znaczy, instrukcją do wykonania kiedy iterator zwróci porażkę. Treść pętli foreach musi zacząć się pierwszą instrukcją następującą po wywołaniu Range. Na końcu pętli foreach, zamiast skoku na początek pętli lub, lub wywołać ponownie iterator, kod ten powinien wykonać instrukcję ret. Powód stanie się jasny za chwilę. Więc implementacja powyższej instrukcji foreach może być następująca:

```
        push    1
        push    10
        push    offset Fordone
        call    Range
        mov     bp, [bp]
        puti
        putcr
        ret
ForDone:
```

Przyznajmy, nie wygląda to wcale jak pętla. Jednakże, stosując jakieś sztuczki ze stosem, zobaczymy, że ten kod rzeczywiście powtarza treść pętli (puti i putcr) jak zaplanowano.

Teraz rozważmy iterator Range,:

```
Range_Start     equ     word ptr <[bp+8]>       ;Address of Start parameter.
Range_Stop      equ     word ptr <[bp+6]>       ;Address of Stop parameter.
Range_Yield     equ     word ptr <[bp+2]>       ;Yield return address.

Range           proc    near
                push    bp
                mov     bp, sp
RangeLoop:      mov     ax, Range_Start         ;Get start parameter and
                cmp     ax, Range_Stop          ; compare against stop.
                ja      RangeDone               ;Terminate if start > stop

; Okay, build the resume frame:

                push    bp                      ;Save dynamic link.
                call    Range_Yield             ;Do YIELD operation.
                pop     bp                      ;Restore dynamic link.
                inc     Range_Start             ;Bump up start value
                jmp     RangeLoop               ;Repeat until start > stop.

RangeDone:      pop     bp                      ;Restore old BP
                add     sp, 2                   ;Pop YIELD return address
                ret     4                       ;Terminate iterator.
Range           endp
```



Rysunek 12.9 Rekord aktywacji Range

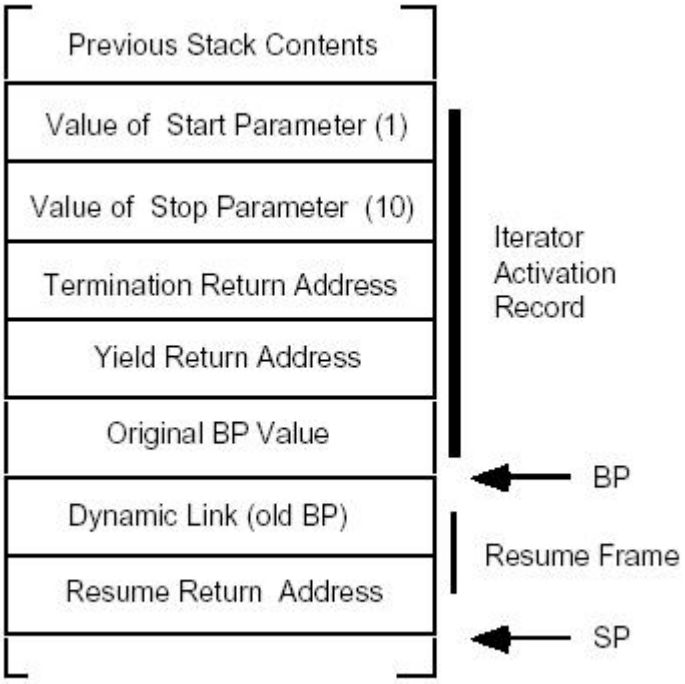
Chociaż ten podprogram jest raczej krótki, jest całkiem złożony. Najlepszy sposób opisania jak ten iterator działa, jest pobranie kilku instrukcji po kolei. Pierwsze dwie instrukcje są standardową sekwencją wejść do procedury. Po wykonaniu tych instrukcji stos wygląda jak na rysunku 12.9

Następne trzy instrukcje w iteratorze Range, przy etykiecie RangeLoop, implementują test zakończenia pętli while. Kiedy parametr Start zawiera wartość większą niż parametr Stop ,sterowanie jest przekazane do etykiety RangeDone, przy której wskazywany kod zdejmuje wartość bp ze stosu, zdejmuje adres powrotny yield ze stosu (ponieważ ten kod nie będzie wracał do treści pętli iteratora) a potem wraca przez adres powrotny zakończenia, który jest bezpośrednio powyżej adresu powrotnego yield na stosie. Instrukcja powrotu również zdejmuje ze stosu dwa parametry.

Rzeczywista praca iteratora ma miejsce w treści pętli while. Instrukcje push, call i pop implementują instrukcje yield. Instrukcje push i call budują ramkę wznowień a potem zwracają sterowanie do treści pętli foreach. Instrukcja call nie wywołuje podprogramu. To ,co tu się robi to wykańcza ramkę wznowień (przez przechowanie adresu powrotnego yield w ramce wznowień) a potem zwrócenie sterownia z powrotem do treści pętli foreach poprzez skok pośredni przez adres powrotny yield odłożony na stos przez początkowe wywołanie iteratora. Po wykonaniu tego wywołania, ramka stosu wygląda tak jak na rysunku 12.9.
Zauważmy również, że rejestr ax zawiera wartość powrotną dla iteratora. Ax jest dobrym miejscem do zwracania wyniku zwracanego iteratora.

Bezpośrednio po powrocie yield do pętli foreach, kod musi przeładować bp oryginalną wartością przed wywołaniem iteratora. Pozwala to kodowi wywołującemu poprawnie uzyskać dostęp do parametrów i zmiennych lokalnych w swoim własnym rekordzie aktywacji zamiast rekordzie aktywacji iteratora. Ponieważ bp tak się zdarzyło, wskazuje na oryginalną wartość bp dla kodu wywołującego, wykonanie instrukcji mov bp, [bp] przeładuje bp odpowiednio. Oczywiście, w tym przykładzie przeładowanie bp nie jest konieczne ponieważ treść pętli foreach nie odnosi się do komórek pamięci z rejestru bp, ale generalnie rzecz biorąc, będziemy musieli przywrócić bp.

Na końcu treści pętli foreach instrukcja ret wznawia iterator. Instrukcja ret zdejmuje adres powrotny ze stosu, który zwraca sterowanie do iteratora bezpośrednio po wywołaniu. Instrukcja w tym momencie zdejmuje bp ze stosu, zwiększa zmienną Start a potem powtarza całą pętlę.



Rysunek 12.10 Rekord wznowienia Range

Oczywiście, jest to dużo pracy aby stworzyć część kodu, która po prostu powtarza pętlę 10 razy. Prosta pętla for byłaby dużo łatwiejsza i całkiem bardziej wydajniejsza niż implementacja foreach opisaną w tej sekcji. Sekcja ta stosuje iterator Range ponieważ było łatwo pokazać jak działa iterator stosując Range, a nie dlatego, że w rzeczywistości implementacja Range jako iteratora jest dobrym pomysłem.

12.9 PODSUMOWANIE

Języki o strukturze blokowej ,takie jak Pascal dostarczają dostęp do nie lokalnych zmiennych spod różnych lex poziomów. Dostęp do nie lokalnych zmiennych jest złożonym zadaniem wymagającym specjalnych struktur danych takich jak łańcuch łącz statycznych lub display. Display jest prawdopodobnie najbardziej efektywnym sposobem dostępu do zmiennych nie lokalnych. 80286 i późniejsze procesory dostarczają specjalnych instrukcji enter i leave dla utrzymania listy display, ale instrukcje te są zbyt wolne dla większości powszechnych zastosowań.

Po szczegóły zajrzyj:

- „Leksykalne zagnieżdżenia, Łącza statyczne i Display'e
- „Zasięg"
- „Łącza statyczne"
- „Uzyskanie dostępu do zmiennych nie lokalnych stosując łącza statyczne"
- „Display"
- „Instrukcje ENTER i LEAVE 80286"
- „Przekazywanie zmiennych spod różnych Lex Poziomów jako Parametrów"
- „Przekazywanie parametrów jako parametrów do innych procedur"
- Przekazywanie procedur jako parametrów"

Iteratory są skrzyżowaniem funkcji i konstrukcji pętli. Są one bardzo potężną programistyczną konstrukcją dostępną w wielu językach wysokiego poziomu. Efektywna implementacja iteratorów wymaga ostrożnego manipulowania stosem w czasie wykonania .Aby zobaczyć jak implementować iteratory odczytaj poniższe sekcje:

- „Iteratory"
- „Implementacja Iteratorów stosując poszerzenie in-line'
- „Implementacja iteratorów z ramką wznowień"

12.10 PYTANIA

1) Co to jest iterator?
2) Co to jest ramka wznowień?
3) Jak iteratory w tym rozdziale implementują wynik sukces lub porażka?
4) Jak wygląda stos kiedy wykonujemy treść pętli sterowanej przez iterator?
5) Co to jest łącze statyczne?
6) Co to jest display?
7) Opisz jak uzyskujemy dostęp do zmiennych nie lokalnych kiedy używamy łącz statycznych.
8) Opisz jak uzyskujemy dostęp do zmiennych nie lokalnych kiedy używamy display'a
9) Jak uzyskamy dostęp do nie lokalnych zmiennych kiedy stosujemy display utworzony przez instrukcję ENTER 80286?
10) Namaluj obraz rekordu aktywacji dla procedury spod lex poziomu 4, który używa instrukcji ENTER dla zbudowania display'a
11) Wyjaśnij dlaczego łącza statyczne pracują lepiej niż display kiedy przekazujemy procedury i funkcje jako parametry.
12) Przypuśćmy, że chcemy przekazać pośrednią zmienną przez wartość-wynik używając techniki gdzie odkładamy wartość przed wywołaniem procedury a potem zdejmujemy wartość (przechowując z powrotem w zmiennej pośredniej) przy powrocie z procedury Dostarcz dwóch przykładów, jeden stosujący łącza statyczne i jeden stosujący display, które implementują przekazywanie wartość -wynik w ten sposób.
13) Skonwertuj poniższy (pseudo) kod pascalowski na język asemblera 80x86. Zakładamy, że Pascal wspiera przekazywanie przez nazwę i przekazywanie przez leniwe wartościowanie parametrów, jak wskazuje na to poniższy kod.