Styl programowania w asemblerze

Wstęp

Większość ludzi uważa programy asemblerowe za trudne do odczytania. Chociaż istnieje wiele powodów dla których ludzie tak uważają, podstawowym powodem jest to ,że języka asembler dla samych programistów stanowi problem w pisaniu czytelnych programów. Nie oznacza to, że jest niemożliwe napisanie czytelnych programów, tylko ,że zabierze to trochę dodatkowego wysiłku ze strony programistów asemblerowych aby stworzyć czytelny kod. Przedstawię tu kilka problemów z programami asemblerowymi. Każdy przykład demonstruje oddzielny problem.

ADDHEX.ASM

```
%TITLE "Suma dwóch liczb szesnastkowych"
IDEAL
DOSSEG
MODEL small
STACK 256
DATASEG
exitCode db 0
prompt1 db 'Podaj wartosc 1: ', 0
prompt2 db 'Podaj wartosc 2: ', 0
string db 20 DUP (?)
CODESEG
EXTRN StrLength:proc
EXTRN StrWrite:proc, StrRead:proc, NewLine:proc
EXTRN AscToBin:proc, BinToAscHex:proc
Start:
mov ax,@data
mov ds,ax
mov es,ax
mov di, offset prompt1
call GetValue
push ax
mov di, offset prompt2
call GetValue
pop bx
add ax,bx
mov cx,4
mov di, offset string
call BinToAscHex
call StrWrite
Exit:
mov ah,04Ch
mov al,[exitCode]
int 21h
PROC GetValue
call StrWrite
mov di, offset string
mov cl,4
call StrRead
call NewLine
call StrLength
mov bx,cx
mov [word bx + di], 'h'
call AscToBin
ret
ENDP GetValue
END Start
```

Cóż, największy problem z tym programem powinien być czywisty – nie ma absolutnie żadnych komentarzy z wyjątkiem tytułu programu. Innym problemem jest fakt ,że łańcuchy które monitują użytkownika pojawiają się w części programu a wywołania które wyświetlają te łańcuchy pojawiają się w innej. Choć jest to typowe programowanie w asemblerze, czyni to program trudniejszym do odczytania. Innym , stosunkowo niewielkim problemem jest to ,że używa TASM'a niż składni IDEALNEJ. Program te używa również "uproszczonych" dyrektyw segmentowych MASM/.TASM. Jak zwykle Microsoft nazywa funkcje tak ,że dodają złożoności do "uproszczonego" wyniku. Okazuje się ,że programy , które używają standardowych dyrektyw segmentacji, będą dużo łatwiejsze do odczytania. Zanim pójdziemy dalej, warto zwrócić uwagę na dwie dobre funkcje tego programu (z szasunkiem dla czytelności). Po pierwsze, programista wybrał rozsądny zbiór nazw dla procedur i zmiennych używanych w tym programie (Zakładam ,że autor tego segmentu kodu jest równiez autorem podprogramów bibliotecznych jakie wywołuje) Innym pozytywnym aspektem tego programu jest to ,że pola mnemoników i operandów są pięknie wyrównane. OK, po skrytykowaniu trudności w odczytaniu tego programu, jak można z niego zrobić wersję bardziej czytelną? Poniższy program będzie prawdopodobnie bardziej czytelny niż wersja powyższa. Prawdopodobnie, ponieważ ta wersja używa UCR Standard Library v2.0 i zakłada ,że czytelnik poznał funkcje tej określonej biblioteki.

```
;************************************************
;
; AddHex-
;
; Ten prosty program odczytuje dwie wartości
; całkowite od użytkownika, oblicza ich sumę
; i wyświetla wynik na ekrani
;
; Ten przykład używa "UCR Standard Library for
; 80x86 Assembly Language Programmers v2.0"
;
;
; 12/11/11
;************************************************

            title AddHex
            .xlist
            include ucrlib.a
            includelib ucrlib.lib
            .list
cseg        segment para public 'code'
            assume cs:cseg
; GetInt-
;
; Ta funkcja odczytuje wartości całkowitej z klawiatury
; i zwraca tą wartość w rejestrze AX.
;
; Ten podprogram sprawdza wartości niepoprawne (albo zbyr duże albo
; niewłaściwe cyfry) i prosi użytkownika o ponowne wpisnie wartości

GetInt          textequ <call GetInt_p>
GetInt_p        proc
                push dx                 ;DX -kod błędu .

GetIntLoop:     mov dx, false           ;Zakładamy brak błędu.
                try                     ;Pułapki błedu.

                FlushGetc               ;Wymuszenie wejścia z nowej linii.
                geti                    ;Odczyt liczby całkowitej.

                except $Conversion      ;Pułapka jeśli zły znak.
                print "Niepoprawna konwersja numeryczna,wpisz ponownie", nl
```

```
                mov dx, true

                except $Overflow        ;Pułapka jeśli # zbyt duże.
                print "Wartość poza zakresem,wpisz ponownie.",nl
                mov dx, true

                endtry
                cmp dx, true
                je GetIntLoop
                pop dx
                ret
                GetInt_p endp

Main            proc
                InitExcept
                print 'Wpisz wartość 1: '
                GetInt
                mov bx, ax
                print 'Wpisz wartość 2: '
                GetInt
                print cr, lf, 'Suma wynosi: '
                add ax, bx
                puti
                putcr

Quit:           CleanUpEx
                ExitPgm                 ;Macro DOS opuszcza program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db    256 dup (?)
sseg            ends

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

Warto zauważyć ,że program ten robi nieco więcej niż oryginalny program AddHex .W szczególności potwierdza dane wejściowe użytkownika; cos czego nie robi program oryginalny. Gdyby dokładnie zasymulować program oryginalny, można by go uprościć do postaci:

```
print       nl, 'Wpisz wartość 1: '
Geti
mov         bx, ax

print       nl, 'Wpisz wartość 2: '
Geti
add         ax, bx
putcr
puti
putcr
```

W tym przykładzie dwie rzeczy poprawiły czytelność programu, dodanie komentarzy i trochę lepsze formatowanie programu, i zastosowanie funkcji wysokopoziomowych UCR Standard Library dla uproszczenia kodowania.

PRZYKŁAD GRAFICZNY

Poniższy segment programu pochodzi z dużo większego programu "MODEX.ASM". Zajmuje się

Ustawianiem koloru grafiki wyświetlacza.

```
;=================================
;SET_POINT (Xpos%, Ypos%, ColorNum%)
;=================================
;
; Kreśli pojedynczy piksel na aktywnym wyświetlaczu
;
; ENTRY:    Xpos = Pozycja X kreślonego piksela
;           Ypos = Pozycja Y kreślonego piksela
;           ColorNum = Kolor nanoszonego piksela
;
; EXIT: Brak znaczących zwracanych wartości
;
SP_STACK STRUC
                  DW ?,?      ; BP, DI
                  DD ?        ; Wywołujący
      SETP_Color  DB ?,?      ; Kolor kreślonego punktu
      SETP_Ypos   DW ?        ; Pozycja Y kreślonego punktu
      SETP_Xpos   DW ?        ; Pozycja X kreślonego punktu
SP_STACK ENDS

      PUBLIC SET_POINT

SET_POINT   PROC FAR

PUSHx       BP, DI                ; zachowanie rejestrów
MOV         BP, SP                ; ustawienia ramki stosu

LES         DI, d CURRENT_PAGE    ; Punkt na aktywnej stornie VGA

MOV         AX, [BP].SETP_Ypos    ; Pobranie linii # piksela
MUL         SCREEN_WIDTH          ; Pobranie Offsetu startu linii

MOV         BX, [BP].SETP_Xpos    ; Pobranie Xpos
MOV         CX, BX                ; Kopiowanie wyodrębninego płaszczyzny #
SHR         BX, 2                 ; Offset X (Bajtów) = Xpos/4
ADD         BX, AX                ; Offset = Width*Ypos + Xpos/4

MOV         AX, MAP_MASK_PLANE1   ; Mapa Maski i wybieranie rejestru
AND         CL, PLANE_BITS        ; Pobranie bitów płaszczyzny
SHL         AH, CL                ; Pobranie wartości wybranej płaszczyzny
OUT_16      SC_Index, AX          ; Wybranie płaszczyzny

MOV         AL,[BP].SETP_Color    ; Pobranie koloru piksela
MOV         ES:[DI+BX], AL        ; Rysowanie piksela
POPx        DI, BP                ; Przywrócenie zapisanych rejestrów
RET         6                     ; Wyjście i wyczyszczenie stosu
SET_POINT   ENDP
```

W przeciwieństwie do poprzedniego przykładu, ten ma wiele komentarzy. Rzeczywiście komentarze nie są złe. Jedna kten określony podprogram ma swoje własne problemy. Po pierwsze większość instrukcji, nazw rejestrów i identyfikatorów jest zapisanych dużymi literami. Duże litery są dużo trudniejsze do odczytania niż małe litery. Nie wpsominając o dodatkowej pracy związanej z wpisywaniem dużych liter do komputera, to prawdziwy wstyd mieć taki błąd w programie. Inny duży problem z tym kodem to to ,że autor nie wyrównał pól etykiet, pól mnemoników i pól operandów (nie jest to strasznem ale źle wpływa na czytelność programu. Oto poprawiona wersja tego programu:

```
;==================================
;
;SetPoint (Xpos%, Ypos%, ColorNum%)
;
;
; Kreśli pojedynczy piksel na aktywnym wyświetlaczu
;
; ENTRY:    Xpos = Pozycja X kreślonego piksela
;           Ypos = Pozycja Y kreślonego piksela
;           ColorNum = Kolor kreślonego piksela
;
;           ES:DI = Adres bazowy ekranu
;
; EXIT: Brak znaczących zwracanych wartości
;
dp                textequ <dword ptr>

Color             textequ <[bp+6]>
YPos              textequ <[bp+8]>
XPos              textequ <[bp+10]>

                  public SetPoint
SetPoint          proc far
                  push bp
                  mov bp, sp
                  push di
                  les di, dp CurrentPage ;Wskazuje aktywną stronę VGA

                  mov ax, YPos           ;Pobiera linię # piksela
                  mul ScreenWidth        ;Pobranie offsetu startu linii

                  mov bx, XPos           ;Pobranie offsetu do linii
                  mov cx, bx             ;Zapis do obliczenia płaszczyzny
                  shr bx, 2              ;Offset X offset (bajty)= XPos/4
                  add bx, ax             ;Offset=Width*YPos + Xpos/4

                  mov ax, MapMaskPlane1  ;Mapa maski i wybranie rejestru
                                         ;płaszczyzny
                  and cl, PlaneBits      ;Pobranie bitów płaszczyzny
                  shl ah, cl             ;Pobranie wartości wybranej płaszczyzny
                  out_16 SCIndex, ax     ;Wybór płaszczyzny

                  mov al, Color          ;Pobranie koloru piksela
                  mov es:[di+bx], al     ;Rysowanie piksela

                  pop di
                  pop bp
                  ret 6
SetPoint          endp
```

Większość zmian był czysto mechaniczna: redukcja dużych liter w programie, lepsze odstępy w programie, wyrównanie komentarzy. Niemniej jednak , te małe , drobne zmiany mają duży wpływ ta jak ten kod jest odczytywany.

Przykład S.COM

Ten kod pochodzi z programu nazwanego "S.COM"

```
;Pobiera wszystkie nazwy plików pasujaće do filespec i ustawia tablicę
GetFileRecords:
      mov dx, OFFSET DTA               ;Ustawienie DTA
      mov ah, 1Ah
      int 21h
```

```
      mov dx, FILESPEC                  ;Pobranie pierwszej nazwy pliku
      mov cl, 37h
      mov ah, 4Eh
      int 21h
      jnc FileFound                     ;Żadnych plików.Spróbuj inny filespec.
      mov si, OFFSET NoFilesMsg
      call Error
      jmp NewFilespec
FileFound:
      mov di, OFFSET fileRecords        ;DI -> przechowanie nazw plików
      mov bx, OFFSET files              ;BX -> tablica plików
      sub bx, 2
StoreFileName:
      add bx, 2                         ;Dla wszystkich które będą pasowały,
      cmp bx, (OFFSET files) + NFILES*2
      jb @@L1
      sub bx, 2
      mov [last], bx
      mov si, OFFSET tooManyMsg
      jmp DoError
@@L1:
      mov [bx], di                      ;Przechowuje wskaźnik do stanu/nazwy
                                        ;pliku w plikach]
      mov al, [DTA_ATTRIB]              ;Przechowanie stanu bajtu
      and al, 3Fh                       ;Szczytowy bit używany do wskazania
                                        ;oznaczonego pliku
      stosb
      mov si, OFFSET DTA_NAME           ;Kopiowanie nazwy liku z DTA do
                                        ;magazynu nazw plików
      call CopyString
      inc di
      mov si, OFFSET DTA_TIME           ;Kopiowanie czasu , daty i rozmiaru
      mov cx, 4
      rep movsw
      mov ah, 4Fh                       ;Kolejna nazwa pliku
      int 21h
      jnc StoreFileName
      mov [last], bx                    ;Zapisz do wskaźnika do ostatniego
                                        ;wejścia pliku
      mov al, [keepSorted]              ;Czy wracając z  EXEC musimy posortować
                                        ;pliki?
      or al, al
      jz DisplayFiles
      jmp Sort0
```

Podstawowym problemem z tym programem jest formatowanie. Pola etykiet nakładają się na pola mnemoników (w prawie każdej instancji), pola operandów różnych instrukcji nie są wyrównane, jest kilka pustych linii dla organizowania kodu, programista nadużywa "lokalnych" nazw etykiet i chociaż nie powszechnie, jest kilka pozycji , które są pisane z dużej litery(pamiętaj ,że duże litery są trudne do odczytania). Ten program również umożliwia użycie "magicznych liczb", szczególnie w odniesieniu do opkodów przenoszonych do DOS. Innym subtelnym problemem z tym programem jest sposób organizacji kontroli przepływu. W paru punktach kod ten sprawdza czy istnieje warunek błędu (plik nie znaleziony i zbyt wiele przetwarzanych plików) Jeśli błąd istnieje, powyższy kod rozgałęzia się do jakiegoś obsługi błędu, który utor umieścił w środku podprogramu. Niestety, to przerywa przebieg tego programu. Większość czytelników bedzie chciało zobaczyć prostą wersję typowych działań programu bez zamartwiania się o szczegóły dotyczące warunków błędu. Niestety , organizacja tego kodu jest taka ,że użytkownik musi przeskakiwać rzadko używany kod aby śledzić co dzieje się dalej. Oto ciut poprawiona wersja powyższego programu:

```
;Uzyskiwanie wszystkich nazw plików dopasowanych do flespec i ustawienie tablicy

GetFileRecords    mov   dx, offset DTA    ;Ustawienie DTA
                  DOS   SetDTA
; Uzyskanie pierwszego pliku, który pasuje do określonej nazwy pliku (który może
; zawierać znaki wielozanczne). Jeśli żaden plik nie pasuje, wtedy
; mamy błąd.


                  mov dx, FileSpec
                  mov cl, 37h
                  DOS FindFirstFile
                  jc FileNotFound

; Tak długo jak żadne pliki nie pasują do file spec(który zawiera
; znaki wieloznaczne), uzyskuje informacje o pliku i umieszcza je w tablicy
; "plików". Za każdym razem przez pętlę "StoreFileName" uzyskamy
; nową nazwę pliku przez wywołanie funkcji DOS'owych FindNextFile (FindFirstFile
; dla pierwszej iteracji). Przechowujemy info o pliku i przechodzimy do
; kolejnego pliku.

                  mov di, offset fileRecords    ;DI -> magazynuje nazwypliku
                  mov bx, offset files          ;BX -> tablica plików
                  sub bx, 2                     ;Specialny przypadek dla iteracji
StoreFileName:    add bx, 2
                  cmp bx, (offset files) + NFILES*2
                  jae TooManyFiles
; Przechowanie wskaźnika do stanu/nazwy pliku w tablicy files[]
; Zwróć uwagę ,że bit H.O.stanu wskazuje ,że plik jest oznaczony.


                  mov [bx], di                  ;Przechowanie wskaźnika w files[]
                  mov al, [DTAattrib]           ;Przechowanie stanu bajtu
                  and al, 3Fh                   ;Czyszczenie pliku z oznaczonum
                                                ;bitem
                  stosb

;Kopiowanie nazwy pliku z obszaru przechowywania DTA do przestrzenie uchylonej.


                  mov si, offset DTAname
                  call CopyString
                  inc di                        ;Przeskocz bajt zero (???).

                  mov si, offset DTAtime        ;Kopiowanie czasu,daty i rozmiaru
                  mov cx, 4
      rep         movsw


;Przejście do kolejnego plikui ponowna próba.

                  DOS FindNextFile
                  jnc StoreFileName

; Po przetwaorzeniu ostatniego wejścia pliku, wyzeruj.
; (1) Zapisz wskaźnik do wejścia ostatniego pliku.
; (2) Jeśli wracamy z EXEC, możemy potrzebować przesortować i wyświetlić pliki


                  mov [last], bx
                  mov al, [keepSorted]
                  or al, al
```

```
                jz DisplayFiles
                jmp Sort0


;Skok w dół jeśli nie było plików do przetworzenia.

FileNotFound:   mov si, offset NoFilesMsg
                call Error
                jmp NewFilespec

;Skocz jeśli było zbyt wiele plików do przetworzenia

TooManyFiles:   sub bx, 2
                mov [last], bx
                mov si, offset tooManyMsg
                jmp DoError
```

Ta porawiona wersja rezygnuje ze zmiennych lokalnych, lepszy format kodu przez wyrównanie wszystkich pól instrukcji i wstawia puste linie w kod. Eliminuje również wiele dużych liter pojawiających się w porzedniej wersji. Inną poprawką jest to ,że kod ten przenosi kod obsługi błędów poza główny strumień tego segmentu kodu, pozwalając czytelnikowi na śledzenie typowego wykonania w bardziej liniowy sposób.

GRUPA DOCELOWA

Oczywiście program w asemblerze będzie nieczytelny dla kogoś kto nie zna języka asemblera. Dotyczy to prawie każdego języka programowania. W powyższym przykładzie, wątpliwe jest ,że "poprawione" wersje są rzeczywiście bardziej czytelne niż wersje pierwotne, jeśli nie zna się języka programowania asembler 80x86. Być może wersje poprawione są raczej bardziej estetyczne w sensie ogólnym, ale jeśli nie znasz asemblera 80x86, wątpliwe jest ,że zrozumiesz więcej z drugiej wersji niż pierwszej.

W związku z powyższym,zasadne jest określenie "grupy docelowej", która będzie czytała nasz programa aseblerowy. Taka osoba powinna:

- Być w miarę kompetentna jeśli chodzi o programowanie w asemblerze 80x86
- Być w miarę zapoznana z problemem jaki próbuje rozwiązać programem asemblerowym
- Biegle czytać po angielsku
- Dobrze znać pojęcia z języka wysokiego poziomu
- Dysponują odpowiednią wiedzą kogoś pracującego w dziedzinie informatyki (tj. rozumieć standard algorytmów i struktur danych, rozumieć podstawową architekturę komputera i rozumie podstawy matematyki dyskretnej.

MIERZALNOŚĆ CZYTELNOŚCI

Jest jedno pytanie "Co sprawia ,że jeden program jest bardziej czytelny niż inny?" Innymi słowy, jak zmierzyć "czytelność" programu? Użyć porównania ,"Wiem ,że to dobrze napisany program bo widziałem inny" jest niewłaściwe; większość ludzi tłumaczy to sobie tak "Jeśli twój program wygląda jak mój najlepszy program wtedy są one czytelne, w przeciwnym razie nie". Oczywiście, takie porównanie jest trochę wartością ponieważ zmienia się wraz z każdą osobą. Aby stworzyć porównanie dla zmierzenia czytelności programu asemblerowego, pierwszą rzecz jaką musimy zrobić to zadać sobie pytanie "Dlaczego czytelność jest ważna?" To pytanie ma prostą (chociaż nieco nonszalancką odpowiedź): Czytelność jest ważna ponieważ programy są czytane (ponadto linię kodu zazwyczaj czyta się dziesięć razy częściej niż się ją pisze). Przyjmijmy fakt ,że większość programów jest czytana przez innych programistów. Większa czytelność twoich

programów sprawia ,że spędzają oni mniej czasu odgadując jak ten program działa Zamiast tego, ogą skoncentrować się na dodawaniu funkcji lub korygowaniu błędów w kodzie. Na użytek dalszego tekstu przyjmijmy takie założenie: Program "czytelny" to taki, który kompetentny programista (który zna się na problemie jaki rozwiązuje ten program) może przyswoić bez wcześniejszego jego oglądania, i w pełni pojąć cały program w minimalną ilość czasu. To jest trudna sprawa! Ta definicja nie brzmi zbyt trudno do uzyskania, ale kilka nietrywialnych programów zawsze osiąga tenb status. Definicja ta sugeruje ,że odpowiedni programista (tj ten który zapoznany jest z problemem jaki próbuje rozwiązać program) może wybrać program, odczytać go w normalnym tempoie i w pełni zrozumieć ten program Wszystko inne nie jest "czytelnym" programem. Oczywiście w praktyce, definicja ta jest niezdatna do użytku ponieważ bardzo niewiele programów osiąga ten cel. Częścią problemu jest to,że programy wydają się zbyt długie i niewielu ludzi jest zdolnych do zarządzania dużą liczbą szczegółów w głowach w jednym czasie. Ponadto, nie ważne jak dobrze może być napisany program, "kompetentny programista" nie sugeruje,że IQ programisty jest tak wysokie ,że może odczytywać instukcje w pełni rozumiejąc ich znaczenie bez większego trudu. Dlatego musimy zdefiniować "czytelność", nie jako jednostkę logiczną, ale jako skalę. Chociaż naprawdę istnieją nieczytelne programy, jest wiele "czytelnych" programów, które są mniej czytelne niż pozostałe. Dlatego ta definicja wydaje się być bardziej realistyczna:

> Czytelny program to taki który składa się z jednego lub więcej modułów. Kompetentny program powinien móc wybrać dany moduł w tym programie i osiągnąć 80% poziom zrozumienia przez wydatkowanie mniej niż jednej minuty na każdą instrukcję w programie.

80% poziom zrozumienia oznacza ,że programista może poprawiać bugi w programie i dodawać nowe funkcje do programu bez popełniania błędów ze względu na nieporozumienia z kodem

JAK UZYSKAĆ CZYTELNOŚĆ

Miara "Poznam jeden kiedy zobaczę" dla czytelnych programów dostarcza dużo podpowiedzi odnośnie tego jak powinno się pisać programy które są czytelne. Jak pisałem wcześniej, sugeruje to ,że indywidualnie rozpatrywany program będzie czytelny jeśli jest bardzo podobny (dobry) do programów, jakie ta dana osoba napisała. To sugeruje ważną cechę, że czytelne programy muszą posiadać : spójność. Jeśli wszyscy programiścipisaliby przy użyciu spójnego stylu. Odkrywali by programy napisane przez innych, podobne do swoich, i dlatego łatwiejsze do odczytania.Ten prosty cel jest głównym zadaniem tego tekstu – zaproponować spójny standard dla wszystkich. Oczywiście sama spójność nie jest wystarczająco dobra. Konsekwentnie złe programy nie są szczególnie proste do czytania. Dlatego, należy ostrożnie rozpatrzyć wytyczne do używania kiedy definiujemy wszechogarniający standard. Celem tegpo tekstu jest stworzenie takiego standardu. Jednak nie należy odosić wrażenia ,że materiały jakie się tu pojawiają pojawiają się bo to brzmi dobrze w danym czasie albo ,że są osobistymi preferencjami, Koncetruje się on na wielu mechanicznych i psychologicznych sprawach, które wpływają na czytelność programu. Na przykład, duże litery są tudniejsze do odczytu niż małe. U człowieka trwa długo rozpoznanie dużych znaków, średnio czlowiekowi zajmuje więcej czasu odczytanie tekstu napisanego dużymi literami. Dlatego należy unikać używania sekwencji dużych liter w programie. Wiele innych tematów tu się pojawiających jest podobnych; sugerują drobne zmiany w sposobie w jakim możesz pisać swoje programy, co czyni łatwiejszym dla kogoś innego rozpoznanie jakichś wzorców w kodzie, zapoczątkowując w ten sposób zrozumienie.

ORGANZACJA TEGO TEKSTU

Tekst podąża od ogółu do szczegółu w dyskusji o czytelności. Zaczyna się od koncepcji programu. Potem omawia moduły. Przechodzimy do procedur. Później omówimy pojedyncze instrukcje. Poza tym pomówimy o komponentach które tworzą instrukcję (tzn instrukcje, nazwy i operatory) W

końcu przejdziemy do omawiania zagadnień ortogonalnych. Część II omawia ogólnie program.To przede wszystkim omówienie dokumentacji, która musi towarzyszyć program i
organizacji plików źródłowych. Omawia również w skrócie zarządzanie konfigurcją i temat kontroli kodu źródłowego. Pamiętaj ,że zrozumienie jak zbudowany jest program (tworzenie, asemblacja, linkowanie, testowanie, debug itd) jest ważne. Jeśli czytelnik w pełni zrozumie algorytm "heapsort" jakiego użyłeś, ale nie może stworzyć modułu wykonywalnego do uruchomienia, nie w pełni jeszcze zrozumiał twój program. Część III omawia organizację modułów w programie w logiczny sposób. Czyni to łatwiejszym dla innych ulokowanie sekcji kodu i organizację powiązanych sekcji kodu razem, gdzie każdy może znaleźć ważnu kod i zignorować nieważny lub nie powiązany kod podczas prób zrozumienia tego oc robi twój program. Część IV omawia użycie procedur wewnątrz programu.Jest to kontynuacja tego tematu w sekcji trzeciej, choć na niższym, bardziej szczegółowym poziomie. Część V omawia program na poziomie instrukcji . Większość zasad tego tesktu pojawia się w tej części. CzęśćVI omawia te tematy, któere wpływają na instrukcje (etykiety, nazwy, instrukcje, operandy itp) . Jest to duża sekcja zawierająca wiele zasad jakimi powinniśmy się kierować pisząc swoje własne, czytelne programy. W tej sekcji omawiamy konwencje nazewnicze, odpowiedniość operatorów i tak dalej. Część VII omawia typy danych i podobne pojęcia. Część VIII zajmuje się sprawami, które nie zostały ujęte w poprzednich sekcjach.

WSKAZÓWKI, ZASADY, WYMUSZONE ZASADY I WYJĄTKI

Nie wszystkie zasady są równie ważne.Na przykład, zasada sparwdzania pisowni wszystkich słów w komentarzach jest prawdopodobnie mniej ważna niż sugestia ,że wszystkie komentarze będą po polsku. Dlatego będę używał trzech określeń dla zachowania tych rzeczy w prostocie: Wskazówki, Reguły i Wymuszone Reguły. Wskazówka jest sugestią. Jest to reguła którą powinieneś stosować ,chyba ,że możesz werbalnie obronić tezę dlaczego powinieneś złamać tę regułę. Tak długo jak jest dobry, do obronienia powód, nie powinieneś czuć obaw przed naruszeniem wskazówki. Wskazówki istnieją aby poprawić spójność w obszarach nie ma dobrych powodów dla wyboru jednej metodologii przed inną. Nie powinieneś naginać Wskazówki tylko dlatego ,że jej nie lubisz – tak robiąc uczynisz programy niespójnymi w stosunku do innych programów, które postępują wedle Wskazówek (i dlatego trudniejsze do oczytu – jednak nie powinieneś cierpieć na bezsenność z powodu złamania Wskazówki). Reguły są dużo ważniejsze niż Wskazówki. Nigdy nie powinieneś łamać reguł chyba ,że jest jakiś wyjątkowy powód zrobienia tego (np. Tworzenie wywołań do podprogramu bibliotecznego zmusza cię do użycia złej konwencji nazewniczej). Kiedy czujesz ,że musisz naruszyć zasadę, należy sprawdzić czy jest to zasadne .Ponadto powinieneś wyjaśnić w komentarzach programu dlaczego było konieczne naruszenie reguły. Reguły są takie – regułami postępowania. Jednak ,są pewne sytuacje gdzie może zajść konieczność naruszenia reguły aby spełnić zewnętrzne wymagania lub nawet uczynić program bardziej czytelnym. Wymuszone Reguły są najtrudniejsze. Nigdy nie powinieneś naruszać wymuszonych reguł. Jeśli jest nawet prawdziwa konieczność zrobienia tego , należy rozważyć zdegradowanie Wymuszonej Reguły do prostej Reguły zamiast traktować naruszenie jako rozsądnej alternatywy. Wyjątek, jest to dokładnie to , znany przykład gdzie naruszamy powszechną Wskazówkę, Regułę lub (bardzo rzadko) Wymuszoną Regułę. Chociaż Wyjątki są rzadkie, stare powiedzenie "Każda reguła ma swój wyjątek.." z pewnością stosuje się do tego tekstu. Wyjątki wskazują pewne powszechne naruszenia jakich można się spodziewać. Oczywiście podział na Wskazówki, Reguły i Wymuszone Reguły to poinia jednego człowieka . W innej organizacji, podział ten może wymagać zmian, w zależności od potrzev organizacji.

JĘZYK ŹRÓDŁOWY

W tym tekście zakładam, że cały prgoram jest pisany w języku asemblera 80x86. Chociaż ta organizacja jest rzadka w komrcyjnych aplikacjach, niech będzie to założenie, które w żaden sposób nie wpłynie na iepoprawność wskazówek. Różne wskazówki istnieją dla różnych języków

wysokopoziomowych. Należy przyjąć rozsądny zbiór wskazówek dla innego języka jakiego używasz i zastosuj te wskzówki do modułów języka asemblera 80x86 w tym programie

CZĘŚĆ II: ORGANIZACJA PROGRAMU

Źródło programu generalnie składa się z jednego lub więcej źródeł, obiektu i plików bibliotecznych. Jeśli projekt rozrasta się i wzrasta liczba plików, trudno śledzić te pliki w projekcie. Jest to szczególnie prawdziwe jeśli liczba różnych projektów współdzieli wspólny zbiór modułów źródlowych.

2.1 Funkcje biblioteczne

Biblioteka,ze swojej natury, sugeruje stabilność. Ignorując możliwość błedów oprogramowania, rzadko można oczekiwać ,że liczba funkcji lub podprogramów w bibliotece będzie się różnić od projektu do projektu. Doprym przykładem jest "Biblioteka Standardowa UCR dla Programistów Języka Assemblera" Można oczekiwać,że "printf" będzie się zachowywać identycznie w dówch różnych programach, które używają Biblioteki Standardowej. Porónajmy dwa programy, każdy z własną implementacją wersji printf. Nie można było rozsądnie założyć ,że oba programy mają identyczną implementację. Prowadzi to do następujących reguł:

Reguła: Funkcje biblioteczne są tymi podprogramami przeznaczonymi dla wspólnego ponownego użycia w wielu różnych programach asemblerowych. Wszystkie biblioteki języka asemblera w systemie powinny istnieć jako pliki ".lib" i powinny pojawić się w podkatalogu "/lib" lub "/asmlib"

Wskazówka: "/asmlib" jest prawdopodobnie lepszym wyborem jeśli używasz wielu języków ponieważ te inne języki mogą potrzebować wstawić pliki w katalog "/lib"

Wyjątek: Jest to prawdopodobnie rozsdądne umieścić plik "stdlib.lib" Biblioteki Standardowej UCR w katalogu "/stdlib/lib" ponieważ więszość ludzi oczekuje go tam

Powyższa zasada zapewnia ,że pliki biblioteczne ,wszystkie są w jednej lokacji, więc są łatwe do znalezienia, modyfikacji i przeglądania. Przez wstawienie wszystkich modułów biblioteki do pojedynczego katalogu, unikniesz problemu zarządzania konfiguracją, takich jak przestarzała wersja biblioteki łączonej w jednym programem i wersja zaktualizowana łączona z innymi programami.

2.2 Wspólne moduły obiektów

Definiujemy bibliotekę jako zbiór modułów obiektu, które mają szerokie zastosowanie w wielu różnych programach. Biblioteka Standardowa UCR jest typowym przykładem takiej bibkioteki. Niektóre moduły obiektów nie są tak uniwersalne, ale jeszcze można znaleźć aplikację w dwóch lub więcej różnych programów. Istnieją problemy zarządzania konfiguracją w tej sytuacji: (1) upewnienie się ,że plik ".obj" jest aktualny kiedy łączysz go z programem. (2) Wiedząc jakie moduły używają tego modułu można sprawdzić czy zmiany w module nie złamią istniejącego kodu.

Poniższe zasady mają tu zastosowanie:

Reguły: Jeśli dwa różne programy współdzielą moduł obiektu, wtedy powiązane źródło, obiekt i makefiles dla tego modułu powinny się pojawić w podkatalogu, który jest określony dla tego modułu (tj żadnych innych plików w tym podkatalogu) Nazwa podkatalogu powinna być taka sama jak nazwa modułu. Jeśli to możliwe, powinieneś

stworzyć zbiór link/alias/skrót do tego podkatalogu i umieścić te linki w katalogu głównym każdego projektu używającego tego modułu. Jeśli link nie jest konieczny, powinieneś umieścić podkatalog modułu w podkatalogu "/common"

Wymuszone Reguła: Każdy podkatalog zawierjący jeden lub więcej modułów, powinien mieć plik make, który automatycznie będzie generował właściwe, aktualne pliki ".obj" pojedynycz plik wsadowy lub inny plik make powien móc automatycznie generuje nowy moduł obiektu (jeśli to konieczne) przez proste wykonanie programu make.

Wskazówki: Microsoft używa programu nmake. Przynajmniej użyj nmake z akceptowalną składnią w twoim makefile.

Inny problem stwierdzony przy projektach, które używają danego modułu jest dużo trudniejszy. Oczywiste rozwiązanie, komentowanie kodu źródłowego powiązanego z tym, modułem mówi ace czytelnikowi jakie programy używają tego modułu jest niepraktyczne. Utrzymywanie tych komentarzy jest zbyt podatne na błędy a komentarze bardzo szybko stają się gorzej niż nieużyteczne – stają się niepoprawne. Lepszym rozwiązaniem jest stworzenie pliku dummy używającego nazwy modułu z przyrostkiem ".elw" (elswhere) i umieszczenie tego pliku w głównym podkatalogu każdego programu który łaczy ten moduł. Teraz używając jednego z czcigodnych programów "whereis" możesz łatwo zlokalizować wszystkie projekty używające tego modułu

Wskazówka: Jeśli projekt używa modułu który nie jest umieszczony w podkatalogu projektu, stwórz plik dummy (używając "TOUCH" lub porównywalnego programu), który używa głównego modułu z przyrostkiem ".elw". Pozwoli to łatwo wyszukać wszystkie projekty, które używają wspólnych obiektów modułu przez użycie programu "whereis"

2.3 Moduły lokalne

Moduły lokalne są to te, jakich używma pojedynczy program / projekt. Zazwyczaj , kod źródłowy i obiektowy dla każdego modułu pojawia się w tym samym katalogu jaki odrębny plik powiązany z tym projektem. Jest to uzasadnione podejście, dopóki liczba plików nie zwiększy się do tego stopnia ,że trudno znaleźć jakiś plik w katalogu. W tym miejscu, większość programistów zaczyna reorganizację swoich katalogów przez tworzenie podkatalogów do przetrzymywania wielu tych modułów źródłowych. Jednak, położenia, nazwa i zawartość tych nowych podkatalogów może mieć duży wpływ na czytelność programu. Pierwszą rzeczą do rozpatrzenia kest zawartość tych nowych podkatalogów. Ponieważ programiści będą grzebać w twoim kodzie w przyszłości, będzie trzeba ulokować pliki źródłowe w projekcie, jest to ważne abyś tak zorganizował te nowe podkatalogi aby można bbyło w nich znaleźć pliki źródłowe jakie do nich przeniesiono. Najlepszym rozwiązaniem jest wstawnienie każdego modułu źródłowego (lub małej grupy silnie powiązanych modułów) do jednego wspólnego podkatalogu. Ten podkatalog powinien nosić nazwę modułu źródłwoego minus przyrostek (lub głównego modułu jeśli jest więcej niż jedne obecny w podkatalogu) Jeśli umieścisz dwa lub więcej pliki źródłowe w tym samym katalogu, upewnij się ,że jesty to zbiór spójny (w znaczeniu ,że pliki źródłowe zawierają kod który rozwiązuje dany pojedynczy problem)

Reguła: Jeśli katalog projektu zawiera zwbyt dużo plików, spróbuj przenieść jakieś moduły do podkatalogu wewnątrz katalogu projektu; nadaj podkatalogowi taką samą nazwę jak plikowi źródłowemu bez przyrostka. To znacznie zredukuje liczbę plików. Jeśli ta redukcja jest niewystarczająca, spróbuj skategoryzować moduły źródłowe (np PlikiIO, Grafika, Dźwięk itd) i przenieś te moduły do podkatalogu noszącego nazwę danej kategorii.

Wymuszona Rreguła: Każdy nowy podkatalog jaki stworzysz powinien mieć swój własny plik make, który będzie automatycznie tworzył asemblację całego modułu źródłowego wewnątrz tego podkatalogu w razie potrzeby

Wymuszina Reguła: Każdy nowy podkatalog jaki tworzysz dla tych modułów źródłowych powinien pojawić się wewnątrz katalogu zawierającego projekt. Jedynym wyjątkiem są te moduły, które są , lub będą, współdzielone z innymi projektami.

Samodzielne programy w asemblerze generalnie zawierają procedurę "główną" – pierwszą jednostką programu, która wykonuje się kiedy system operacyjny ładuje program do pamięci. Dla każdego programisty, w nowym projekcie, ta procedura jest kotwicą, gdzie zacznie odczytywanie kodu i punktem do którego stale będzie wracał czytający. Dlatego też, czytelnik powinien moc łatwo zlokalizować ten plik źródłowy. Poniższe zasady będą pomocne w takim przypadku;

Reguła: Moduł źródłowy zawierający program główny powinien mieć taką samą nazwę jak wykonywalny (oczywiście przyrostek powinien być inny). Na przykład, jeśli nazwą pliku wykonywalnego "Simulate 886" jest "Sim886.exe" wtedy powinieneś znaleźć program główny w pliku źródłowym "Sim886.asm"

Znajdowanie pliku źródlowego który zawiera program główny to jedna rzecz. Znajdowanie samego programu głównego może być tak samo trudne. Język asembler pozwala ci nadawać programowi głównemu nazwę jaką chcesz. Jednak, aby uczynić procedurę łatwą do znalezienia (zarówno w kodzie źródłowym jak i na poziomie OS), powinieneś nazwać ten program "main".

Reguła: Nazwa głównej procedury w programie asemblerowym powinna być "main"

2.4 Program Make File

Każdy projekt, nawet jeśli zawiera tylko pojedynczy moduł źródłowy, powinien mieć powiązany plik make. Jeśli ktoś chce zasemblować twój program, powinien nie martwić się jakiego programu używa (np. MASM), jakich opcji wiersza poleceń użyć, jakich użyć bibliotek itd. Powinien wpisać "nmake" i skończyć z programem wykonywalnym. Nawet jeśli asemblacja programu składa się nie więcej niż wpisania nazwy asemblera i pliku źródłowego, powinieneś mieć plik make.

Wymuszona Reguła: Katalog projektu głównego powinien zawierać plik make, który automatycznie genruje plik wykonywalny (lub inny oczekiwany moduł obiektu) w odpowiedzi na proste polecenie make/nmake

Reguła: Jeśli projekt używa modułów obiektu które nie są w tym samym podkatalogu co moduł programu głównego, powinieneś przetestować pliki ".obj" dla tych modułów i wykonać odpowiednie pliki make w ich katalogach jeśli kod obiektowy jest przestarzały. Zakładamy ,że biblioteki są zaktualizowane

Wskazówka: Unikaj fanatzyjnych funkcji "make". Większość programistów uczy się tylko podstaw o make i nie będą mogli zrozumieć co robi twój plik make jeśli w pełni wykorzystasz język make. Szczególnie unikaj stosowania domyślnych reguł ponieważ może to tworzyć spustoszenie jeśli ktoś arbitralnie doda lub usunie pliki z katalogu zawierającego plik make.

CZĘŚĆ III: ORGANIZACJA MODUŁU

Moduł jest zbiorem obiektów, które są logicznie powiązane. Obiekty te mogą zwierać stałe, typy danych, zmienne i jednostki programu (tzn funkcje, procedury itp). Zwróć uwagę, że w module nie muszą być fizycznie połączone. Na przykład, jest całkiem możliwe skonstruowanie modułu używającego kilku różnych plików źródłowych. Podobnie , jest całkiem możliwe posiadanie kliku różnych modułów w tym samym pliku źródłowym .Jednak, najlepsze moduły są fizycznie połączone jak również połączone logicznie; to znaczy, wszystkie obiekty powiązane z modułem istniejąw pojedynczym pliku źródłowym (lub katalogu jeśli plik źródłowy będzie zbyt długi) i nic innego nie ma. Moduły zawierają kilka różnych obiektów obejmujących stałe, typy, zmienne i jednostki programu (podprogramy) Moduły współdzielą wiele z tych atrybutów z podprogramami; to znaczy nie ma nic dziwnego ponieważ podprogramy są głównymi koponentami typowego modułu. Jednak, moduły mają pewne dodatkowe atrybuty, swoje własne. Poniżej opisane są atrybuty dobrze napisanych modułów
Uwaga: Jednostka (unit) i pakiet (package) , obie nazwy są synonimem dla terminu moduł.

3.1 Atrybuty modułu

Moduł to ogólny termin opisujący zbiór powiązanych obiektów programu (jednostki programu jak również dane i typy obiektów), które są jakoś połączone. Dobry moduł wspóldzieli te same atrybuty jak dobra jednostka programowa jak również możliwości ukrywania pewnych szczegółów z kodu zewnętrznego modułu

3.1.1 Spójność modułu

Moduły wykazują następujące rodzaje spójności (od dobrych do złych):

- Funkcjonalna lub logiczna spójność istnieje jeśli moduł wykonuje dokładnie jedno (proste) zadanie
- Skewencyjna lub potokowa spójność istnieje kiedy moduł wykonuje kilka sekwencyjnych operacji, które muszą być wykonywane w pewnej kolejności z danymi z jednej operacji będącej wprowadzanej do kolejnej w modzie "jak filtr"
- Globalna lub komunikacyjna spójność istnieje wtedy kiedy moduł wykonuje zbiór działań , które korzystają ze wspólnego zbioru danych, ale w inny sposób nie powiązanych
- Czasowa spójność istnieje keidy moduł wykonuje zbiór działań , które muszą wykonane w tym samym czasie (choć nie koniecznie w tej samej kolejności). Typowy moduł inicjalizujący jest przykładem takiego kodu
- Spójność proceduralna istnieje ,kiedy moduł wykonuje sekwencję działań w określonym porządku, ale jedynie rzeczy, które łączą je razem w porządku w jakim muszą być zrobione. W prezeciwieństwie do spójności sekwencyjnej, operacje nie współdzielą danych
- Spójność stanu występuje kiedy kilka różnych (niepowiązanych) operacji poajwia się w tym samy module a stan zmiennej (np parametr) wybiera operację do wykonania. Zazwyczja takie moduły zawierają instrukcje case (switch) i if...elseif...elseif.
- Brak spójności istnieje, jeśli działania w module nie mają widocznych związków z pozostałymi

Pierwsze trzy spójnośći są generalnie akceptowalne w programie. Czwarta (czasowa) jest prawdopodobnie OK, ale powinieneś jej używać rzadko. Ostatnie trzy formy nie powinny prawie nigdy pojawiać się w programie.

Wskazówka: Projektuj dobre moduły! Dobry moduł wykazują silną spójność. To znaczy, moduł powinien oferować (małą) grupę usług, które są logicznie powiązane. Na przykład, moduł "printer" może dostarczać wszystkich usług związanych z drukowaniem. Pojedyncze podprogramy wewnątrz modułu dostarczają pojedynczych usług

3.1.2 Sprzęganie modułu

Sprzęganie, odnosi się do sposobu w jaki dwa moduły komunikują się między sobą. Jest kilka

kryterii ,któere definiują poziom sprzężenia między dwoma modułami.

- Kardynalność – liczba obiektów przekazywanych między dwoma modułami. Im mniej tym lepiej (mnniej parametrów)
- Intymność – jak "prywatna" jest komunikacja? Lista parametrów jest najbardziej prywatną formą; pola danych prywatnych w klasie lu obiekcie są kolejnym poziomem; pola danych publicznych w klasie lub oviekcie są kolejne, globalne zmienne są najmniej intymne a przekazywanie danych w polu lub bazie danychjest najmniej intymnym połączeniem. Dobrze napisany moduł wykazuje najwyższy stopień intymności
- Widoczność – jest to coś podobnego do powyższej intymności. Odnosi się do tego jak dane są widoczne w całym systemie, które są przekazywane między dwoma modułami. Na przykład, przekazanie danych w liście parametrów jest bezpośrednie i bardzo widoczne (zawsze widzisz dane jakie wywołujący przekazuje do wywoływanego podprogramu); przekazywanie danych w zmiennych globalnych tworzy transfer mniej widoczny (możesz mieć ustawione zmienne globalne na długo przed wywołaniem podprogramu). Inny przykład to przekazywanie prostych (skalarnych) zmiennych zamiast ładowania kliku wartości do struktury/rekordu i przekazywanie tej struktury/rekordu podprogramu wywołującego.
- Elastyczność – odnosi się to tego jak łatwo jest tworzyć połączenia między dwoma podprogramami, które nie miały być pierwotnie przeznaczone do połączenia między sobą.. Na przykład przypuśćmy,że przekazujesz struktruę zawierającą trzy pola do funkcji. Jeśli chcesz wywołać tą funkcję ale masz tylko trzy obiekty danych, nie strukturę, musisz stworzyć prostą strukturę, skopiować te trzy wartości do pól tej struktury a potem wywołać funkcję. Z drugiej strony, masz po prostu przekazać te trzy wartości jako oddzielne parametry, możesz jeszcze przekazać w strukturze (przez określenie każdego pola) jak również wywołanie funckji z oddzielnymi wartościami. Moduł zawierający funkcje jest bardziej elastyczny.

Moduł jest luźno powiązany jeśli jego funkcje wykazują niską kardynalność, wysoką intymność, wysoką widoczność i wysoką elastyczność. Często te funkcje są w konflikcie z innymi (np. Zwiększenie elastyczności przez wyranie tych pól ze struktury [dobra rzecz] również zwiększy kardynalność [złą rzecz]) Jest to tradycyjny cel każego inżyniera wybierającego właściwe kompromisy dla każdego indywidualnego warunku; dlatego też musisz ostrożnie balansować kadym z tych czterech atrybutów.
Moduł który używa luźnych powiązań generalnie zawiera kilka błedów na KLOC (tysiące linii kodu).Co więcej, moduły które wykazują luźne powiązania są łatwiejsze do ponownego wykorzystania (w obecnych i przyszłych projektach).

Wskazówka: Projektuj dobre moduły! Dobry moduł wykazuje luźne powiązania. To znaczy, jest tylko kilka dobrze zdefiniowanych (widzialnych) interfejsów między modułem a światem zewnętrznym. Większość danych jest prywatnych, dostępnych tylko przez funkcje dostępowe .Ponadto, interfejs powinien być elstyczny.

Wskazówka: Projektuj dobre moduły! Dobre moduły wykazują ukrywanie informacji. Kod na zewnątrz modułu powinien tlko mieć dostęp do modułu przez mały zbiór publicznych podprogramów. Wszystkie dane powinny być prywatne w tym module. Moduł powinien implementować abstrakcyjny typ danych. Cały interfejs do modułu powinien być dobrze zdefiniowanym zbiorem operacji.

3.1.3 Fizyczna organizacja modułów

Wiele jezyków programowania dostarcza bezpośredniego wsparcia dla modułów(tj pakietów w Ada, modułów w Modula-2, i unitów w Delphi / Pascal). Niektóre języki dostarczają tylko pośredniego wsparcia dla modułów (plik źródłowe w C/C++) Inne jak BASIC,nie obsługują

modułów, więc musisz zasymulować je przez fizyczne pogrupowanie obiektów razem i przećwiczenie jakiejś dyscypliny. Język asemblera podpada pod grupę środkową. Podstawowy mechanizm dla ukrywania nazw z innych modułów jest implementowanym modułem jako indywidualny plik źródłowy i publikowanie tych nazw, które są częścią interfejsu modułu dla świata zewnętrznego

Reguła: Każdy moduł powinien całkowicie znajdować się w pojedynczym pliku źródłowym. Jeśli rozmiar go przekracza, wszystkie pliki źródłowe dla danego modułu powinny rezydować w podkatalogu szczególnie zaprojektowanego dla tego modułu.

Niektórzy ludzie mają szalony pomysł ,że modularyzacja oznacza wstawianie każdej funkcji do oddzielnego pliku źródłowego. Taka fizyczna modularyzacja generalnie szkodzi czytelności programu bardziej niż pomaga. Zmiast na logicznej modulryzacji ,która jest definiowaniem modułu przez jego działanie zamiast składnię kodu źródłowego.

3.4 Interfejs modułu

W każdym języku system który obsługuje moduły, są dwa podstawowe komponenty modułu: komponent interfejsu , który upublicznia widzialne nazwy modułu i komponent implementacji, który składa się z aktualnego kodu, danych i prywatnych obiektów. MASM (i większość asemblerów) używa schematu który jest bardzo podobny do używanego w C/C++ Są dyrektywy, które pozwalają na importowanie i eksportowanie nazw. Podobnie jak C/C++ możesz umieszczać te dyrektywy bezpośrednio w powiązanych modułach źródłowych. Jednak taki kod jest trudny do zarządzania (ponieważ musisz zmienić dyrektywy w każdym pliku kiedy chcesz zmodyfikować publiczną nazwę).Rozwiązaniem p[rzyjętym z C/C++ jest użycie plików nagłówkowych. Pliki nagłówkowe zawierają wszystkie publiczne definicje i eksporty (jak również definicje wspólnego typu danych i definicji stałych) Plik nagłówkowy dostarcza interfejsu do innych modułów, które chcą użyć kodu obecnego w implementowanym module. Dyrektywa externdef MASM 6.x jest doskonała dla tworzenia plików interfejsu. Kiedy używasz externdef wewnątrz modułu źródłowego ,który definiuje symbol, externdef zachowuje się jak dyrektywa public, eksportując nazwę do innych modułów. Kiedy używasz externdef wewnątrz modułów źródłowych, które odnoszą się do zewnętrznych nazw, externdef zachowuje się jak dyrektwywa extern (lub extrn). Pozwala to umieścić dyrektywę externdef w pojedynczym pliku i zawiera ten plik w modułach które importują i eksportują nazwy publiczne. Jeśli używasz asemblera , który nie wspiera externdef, prawdopodobnie powinieneś rozważyć przejście na MASM 6.x ,.Jeśli przejście na lepszy asembler (wspierający externdef) nie jest wykonalne, ostatnia rzecz jaką chcesz robić to zarządzanie interfejsem informacji w kilku oddzielnych plikach. Zamiast tego, użyj dyrektywy assemblera ifdef warunkowej asemblacji dla asemblacji zbioru w pliku nagłówkowym jeśli symbol z nazwą modułu jest zdefiniowana przed zawarciem pliku nagłówkowego. W przeciwnym razie oPowinno to zasemblować zbiór instrukcji extrn. Chociaż musisz jeszcze zarządzać publicznymi i zewnętrznymi informacjami w dwóch miejscach(w sekcji ifdef true i false), są w tym samym pliku i umieszczone blisko jedna drugiej.

Reguła: Trzymaj wszystkie dyrektywy interfejsu modułów (public, extrn, extern i externdef) w pojedynczym pliku nagłówkowym dla danego modułu. Umieść inne definicje wspólnych typów danych i definicje stałych również w tym pliku nagłówkowym.

Wskazówka: Powinien być tylko jeden plik nagłówkowy powiązany z dowolnym jednym modułem (nawet jeśli moduł ma wiele plików źródłowych z nim związanych). Jeśli , z tego samego powodu, czujesz ,że jest konieczne posiadanie wielu plików nagłówkowych związanych z modułem, powinieneś stworzyć pojedynczy plik, który zawierać będzie wszystkie inne interfejsy plików. W ten sposób program jakiego chcesz użyć wszystkich plików nagłówkowych potrzebuje tylko pojedynczego pliku.

Kiedy projektujesz pliki nagłówkowe, upewnij się ,że możesz zawrzeć plik więcej niż raz bez złego wpływu (tzn duplikowania symboli błędów). Tradycyjny sposób zrobienia tego to wstawienie instrukcji IFDEF pośród wszystkich innstrukcji w pliku nagłówkowym:

```
;  Module: MyHeader.a

                indef   MyHeader_A

MyHeader_A      =       0
                .
                .                       ;instrukcje w tym pliku nagłówkowym
                .
                endif
```

Za pierwszym razem, plik źródłowy zawiera "MyHeader.a" symbol "MyHeader_A" nie jest zdefiniowany. Dlatego też, asembler będzie przetwarzał wszystkie instrukcje w tym pliku nagłówkowym. W kolejnych operacjach zawierania (podczas samej asemblacji) symbol "MyHeader_A" jest już zdefiniowany, więc asembler ignoruje ciało pliku nagłówkowego. Czy można dołączać plik dwukrotnie? Łatwo. Niektóre pliki nagłówkowe mogą zawierać inne pliki nagłówkowe. Przez dołączenie pliku "YourHeader.a" moduł również może dołączać "MyHeader.a" (zakładając ,że "YourHeader.a" zawiera właściwą dyrektywę dołączania) Program główny, który zawiera "YourHeader.a" może również potrzebować "MyHeader.a" ,aby wyraźnie dołączyć ten plik nie zdając sobie z tego sprawy. "YourHeader.a" już przetworzył "MyHeader.a" powodując tym samym redefiniowanie symbolu.

Reguła: Zawsze wstawiaj właściwą instrukcję IFNDEF wokół wszystkich definicji w pliku nagłówkowym aby pozwolić na dołączenie wielu plików nagłówkowych bez efektów ubocznych .

Wskazówka: Użyj przyrostka ".a" dla plików nagłówkowych/interfejsu języka asemblera
Reguła: Pliki dołączane dla funkcji bibliotecznych w systemie powinny istnieć jako pliki ".a" i powinny się pojawić podkatalogu "/include" lub "/asminc"
Wskazówka: "/asminc" jest prawdopodobnie lepszym wyborem jeśli używasz wielu języków ponieważ te inne języki mogą potrzebować wstawiać pliki w katalogu "/include"

Wyjątek: Jest rozsądne pozostawić plik "stdlib.a" z Biblioteki Standardowej UCR w katalogu "/stdlib/include" ponieważ większość ludzi oczekuje go tam.

CZĘŚĆ IV: ORGANIZACJA JEDNOSTKI PROGRAMU

4.0 Organizacja jednostki programu

Jednostka programu jest dowolną procedurą, funkcją, współprogramem, iteratorem, podprogramem, wtyczką, lub innym terminem który opisuje sekcję kodu, który streszcza zbiór wpólnych działań na komputerze.Dla opisu tych pojęć będę używał terminu procedura lub podprogram. Podrogramy są ściśle związane z modułami, ponieważ zwykle są one głównymi składnikami modułu (wraz z danymim stałymi i typami) Zatem wiele atrybutów jakie stosuje się do modułów, również stosuje się do podprogramów.

4.1 Spójność podprogramu

Podprogramy wykazują następujące rodzaje spójności (od dobrych do złych):

- Funkcjonalna lub logiczna spójność istnieje jeśli podprogram wykonuje dokładnie jedno proste zadanie
- Sekwencyjna lub potokowa spójność istnieje kiedy podprogram wykonuje kilka sekwencyjnych działań, które muszą być wykonywane w pewnej kolejności z danymi z jednego działania będącego przenoszonej do kolejnej w modzie "filtra"
- Globalna lub komunikacyjna spójność istnieje kiedy podprogram wykonuje zbiór działań , które używają wspólnego zbioru danych, ale są niepowiązane
- Spójność czasow istnieje kiedy podprogram wykonuje zbiór działań, które muszą być wykonane w tym samym czasie (chociaż nie koniecznie w tym samym porządku). Typowym przykładem takiego kodu jest podprogram inicjalizujący.
- Spójność proceduralna istnieje kiedy podprogram wykonuje sekwencję działań w określonej kolejności, ale jedyna rzecza jaka je łączy razem jest porządek w jakim muszą być wykonane. W przeciwieństwie do spójności sekwencyjnej,m działania nie współdzielą danych.
- Spójność stanu występuje kiedy kilka różnych (niepowiązanych) działań pojawia się w tym samym module a stan zmiennej (np. parametr) wybiera działanie do wykonania. Taki typowy podprogram zawiera instrukcje case (switch) lub if...elseif...elseif
- Brak spójności istnieje jeśli działania w podprogramie nie mają wyraźnego związku ze sobą

Pierwsze trzy formy spójności są generalnie akceptowalne przez program,. Czwarta (czasowa) jest OK ale powinieneś jej używać rzadko. Ostatnie trzy formy prawie nigdy nie powinny pojawiać się w programie.

Wskazówka: Wszystkie podprogramy wykzują dobrą spójność. Spójność funkcjonalna jest najlepsza, po niej jest spójność sekwencyjna i globalna. Spójność czasowa jest OK,, okazyjnie. Powinieneś unikać innych form.

4.1.1 Sprzężenie podprogramu

Sprzężenie odnosi się do sposobu w jaki dwa podprogramy komunikują się jeden z drugim. Jest kilka kryterii, które definiują poziom sprzężenia między dwoma podprogramami:

- Kardynalność – liczba obiektów komunikujących się między dwoma podprogramami. Mniej obiektów tym lepiej
- Intymność – jaka jest komunikacja "prywatna"? Listy parametrów są najabrdziej prywatną formą; pola danych prywatnych w klasie lub obiekcie są kolejnym z poziomów; pola danych publicznych w klasie lub obiekcie są kolejne, zmienne globalne są mniej intymen a przekazywanie danych w pliku lub bazie danych jest najmniej intymnym połączeniem. Dobrze napisany podprogram wykazuje wysoki stopień intymności.
- Widzialność – jest to coś podobnego do intymności. Odnosi się to do widzialności danych w całym systemie, które są przekazywane między dwoma podprogramami. Na przykład, przekazywanie danych w liście parametrów jest bezpośrednie i bardzo widoczne; przekazywanie danych w zmiennych globalnych czyni transfer mniej widoczny. Inny przykład to przekazywanie prostych (skalarnych) zmiennych zamiast ładowania wartości do struktury/ rekordu i przekazanie tej struktury / rekordu do programu wywołujacego.
- Elastyczność – odnosi się do tego jak łatwo uczynić połączenie między dwoma podprogramami, które nie mogą być pierwotnie przeznaczone do wywołania jednego przez drugi. Przypuśćmy ,że przekazujesz strukturę zawierającą trzy pola w funkcji, Jeśli chcesz wywołać tą funkcję ale masz tylko trzy obiekty danych, nie strukturę, musisz stworzyć prostą strukturę, skopiuj te trzy wartości do tych pól tej struktury, a potem wywołaj ten podprogram. Z drugiej strony, po prostu przekazując te trzy wartości jako oddzielne parametry możesz jeszcze przekazać w strukturze (przez określenie każdego pola) jak

również wywołania podprogramu z określonymi wartościami.

Funkcja jest luźno sprzężona jeśli wykazuje niską kardynalność, wysoką intymność, wysoką widzialność i wysoką elastyczność. Często te funkcje są w konflikcie jedna z drugą (np zwiększenie elastyczności przez wyjęcie pól ze struktury [dobra rzecz] zwiększy również kardynalność[zła rzecz]). Zadaniem każdego inżyniera jest wybór właściwego kompromisu dla każego indywidualnego warunku; dlatego też musisz ostrożnie balansować przy każdym z czterech atrybutów. Program który używa luźnej spójności generalnie zawiera kilka błędów na KLOC (tysiące linii kodu). Podprogramy które używają luźnego sprzężenia są łatwiejsze do ponownego użycia

Wskazówka: Sprzężenie między podprogramami w kodzie źródłowym powinno być luźne

4.1.2 Rozmiar podprogramu

Gdzieś w 1960 roku , ktoś postanowił ,że programista może patrzeć tylko na jedną stronę listingu w jednym czasie, dlatego podprogramy powinny być maksymalnie długie na jedną stronę (66 lini jednocześni) W 1970 roku, kiedy obliczanie interaktywne stało się popularne, sprowadziło się to do 24 linii – rozmiar ekranu terminala. W rzeczywistości jest mało empirycznych doświadczeń sugerujących ,że mały rozmiar podprogramu jest dobrym pomysłem. W rzeczywistości kilka badań nad kodem zawerających sztuczne ograniczenia wielkości rozmiaru podprogramu wskazuje wręcz odwortnie – krótszy podprogram zawiera więcej błędów na KLOC. Podprogram , który wykazuje spójność funkcjonalną jest poprawnego rozmiaru, prawie bez względu na liczbę linii kodu jakie zawiera. Nie powinieneś sztucznie podzielić podprogram na dwa lub więcej podprogramy, ponieważ czujesz ,że podprogram staje się zbyt długi. Po pierwsze, zweryfikuj czy podprogram wykazuje silną spójność i luźne sprzężenie. Jeśli jest to ten przypadek, podprogram nie jest zbyt długi. Zapamiętaj jednak ,że długi podprogram jest prawdopodobnie dobrym wskazaniem ,że jest wykonywane kilka działań i dlatego nie wykazuje silnej spójności. Większość badań na tene temat wskazuje ,że podprogramy powyżej 150-200 linii kodu moze zawierać więcej błędów i są bardziej kosztowne w naprawie niż krótkie podprogramy. Odnotuj, przy okazji, że nie liczysz pustych linii lub linii zawierających tylko komentarze kiedy zliczasz linie kodu w programie. Zauważ również, że większość badań dotyczących rozmiaru podprogramu działało z HLL'ami. Podprogram porównywalnego języka asemblera będzie zawierał więcej linii kodu niż odpowiedni podprogram HLL. Dlatego możesz się spodziewać ,że twoje programy asemblerowe mogą być trochę dłuższe.

Wskazówka: Nie pozwalaj aby sztuczne ograniczenia wpływały na rozmiar podprogramów. Jeśli podprogram przekracza 200-250 lini kodu, upewnij się ,że podprogram wykazuje funkcjonalną lub sekwencyją spójność Sprawdź ponadto czy nie ma jakichś ogólnych podsekwencji w kodzie, które można włączyć do samodzielnych podprogramów.

Regułą: Nigdy nie skracaj podprogramu przez dzielenie go na n części , które zawsze wywołuje go we właściwej sekwencji jako sposób skracania oryginalnego podprogramu

4.2 Umieszczanie głównej procedury i danych

Jak napisano wcześniej, powinieneś nazwać procedurę główną main i umieścić ją w pliku źródłowym o tej samej nazwie co plik wykonywalny. Jeśli ten moduł jest dość długi, może jeszcze to utrudnić umieszczenie programu głównego. Dobrym rozwiązaniem jest umieszczenie zawsze procedury głównej w tym samym miejscu w pliku źródłowym. Zgodnie z konwencją , większość programistów czyni swój program główny pierwszą lub ostatnią procedurą w programie języka asemblerowego. Każda pozycja jest dobra. Wstawienie programu głównego gdziekolwiek czyni go trudnym do znalezienia

Reguła: Zawsze czyń procedurę główną pierwszą lub ostatnią procedurą w pliku źródłowym

MASM, ponieważ jest asemblerem wielofazowym, nie wymga aby definiować symbol przed jego użyciem. Jest to konieczne ponieważ wiele instrukcji (jak JMP) musi odnosić się do symboli znajdujących się dalej w programie. W podobny sposób, MASM rzeczywiście nie martwi się gdzie zdefiniowałeś swoje dane – przed lub po ich zastosowaniu. Jednak większość programistów "dorastała" z językami wysokopoziomowymi, które wymagają definiowania symboli przed ich pierwszym zastosowaniem. W wyniku tego , oczekują możliwości znajdowania deklaracji zmiennych przez przeglądanie wsteczne w pliku źródłowym. Ponieważ każdy tego oczekuje, dobrą tradycją jest kontynuowanie tego w programie asemblerowym

Reguła : Powinieneś deklarować wszystkie zmienne, stałe i makra przed ich zastosowaniem w programie asemblerowym

Reguła: Powinieneś zdefiniować wszystkie zmienne statyczne (te deklarowane w segmencie) na początku modułu źródłowego.

CZĘŚĆ V: ORGANIZACJA INSTRUKCJI

W programie asemblerowym, autor musi się sporo napracować aby uczynić program czytelnym. W związku z liczną liczbą zasad można tworzyć program , który jest czytelny. Jednak przez złamanie jednej zasady, nie ważne wdług ilu innych postępujesz, może uczynić program nieczytelnym. Nigdzie nie jest to bardziej prawdziwe niż przy organizacji instrukcji wewnątrz programu.
Microsoft Macro Assembler jest asemblerem darmowym. Różne pola instrukcji asemblerowych mogą pojawiać się w dowolnych kolumnach (tak długo jak pojawiają się we własciwej kolejności) Ilość spacji i tabulatorów może oddzielać różne pola w instrukcji. Dla asemblera poniższe dwie sekwencje kodów są identyczne:

```
                mov ax, 0
                mov bx, ax
                add ax, dx
                mov cx, ax

mov             ax,             0
        mov bx,                 ax
    add             ax, dx
                      mov               cx, ax
```

Pierwszy kod jest dużo łatwiejszy do przeczytania niż drugi.Chociaż jest to skrajny przykład, zauważ ,że tylko kilka błędów ma duży wpływ na czytelność programu Rozważmy taki przykład

```
GetFileRecords:
        mov dx, OFFSET DTA              ;Ustawienie  DTA
        mov ah, 1Ah
        int 21h
        mov dx, FILESPEC                ;Pobranie nazwy pierwszego pliku
        mov cl, 37h
        mov ah, 4Eh
        int 21h
```

```
        jnc FileFound                           ;Brak plików. Spróbuj innego filespec.
        mov si, OFFSET NoFilesMsg
        call Error
j       mp NewFilespec
FileFound:
        mov di, OFFSET fileRecords              ;DI -> przechowanie nazw plikó
        mov bx, OFFSET files                    ;BX -> tablica plików
        sub bx, 2
```

Wersja poprawiona:

```
GetFileRecords: mov     dx, offset DTA          ;Ustawienie DTA
                DOS     SetDTA

                mov     dx, FileSpec
                mov     cl, 37h
                DOS     FindFirstFile
                jc      FileNotFound


                mov     di, offset fileRecords  ;DI -> przechowanie nazw plików
                mov     bx, offset files        ;BX -> tablica plików
                sub     bx, 2                   ;Specjalny przypadek 1 iteracji
```

Instrukcja asemblerowa składa się z czterech możliwych pól: pola etykiety, pola mnemonika, pola operandu i pola komentarza. Pole mnemonika i komentarza są zawsze opcjonalne. Pole etykiety jest generalnie opcjonalne chociaż pewne instrukcje (mnemoniki) nie pozwalają na etykiety podczas gdy inne ich wymagają. Obecność pola operandu jest związana z polem mnemonika. Dla większości instrukcji aktualny mnemonik określa czy pole operandu jest wymagane. Wolność układania tych kolumn w dowolny sposób jest jedną z podstawowych przyczyn trudności w odczytywaniu programów asemblerowych.Chociaż MASM pozwala ci pisać programy w swobodnej formie, nie ma absolutnie żadnego powodu abyś nie mógł przyjąć stałęgo formatu pól , zawsze zaczynając każde pole w tej samej kolumnie. Oto zasady jakich powinieneś się trzymać:

Reguła:	Jeśli identyfikator jest obecny w polu etykiety, zawsze zaczynaj ten identyfikator w kolumnie jeden linii źródłowej
Reguła:	Wszystkie mnemoniki powinny zaczynać się w tej samej kolumnie. Generalnie, powinna to być kolumna 17, lub inna konwencjonalna pozycja
Regułą:	Wszystkie operandy powinny zaczynać się w tej samej kolumnie. Powinna to być kolumna 25 lub inna konwencjonalna pozycja.
Wyjątek:	Jeśli mnemonik (zazwyczja makro) jest dłuższy niż siedem znaków i wymaga operandu, nie masz wyboru ale zaczniesze pole operandu za kolumną25 (to jest wyjątek zakładający ,że wybrałeś kolumny 17 i 25 dla pól mnemonika i operandu , odpowiednio)
Wskazówka:	Próbuj zawsze zaczynać pole komentarza w sąsiednich liniach źródła tej samej kolumny (zwróć uwagę ,że jest to niepraktyczne zawsze zaczynać pole komentarza w tej samej kolumnie przez cały program)

Większość ludzi uczy się języka wysokopoziomowego przed nauczeniem sięasemblera. Nauczyli się ,że czytelny (HLL) program ma swoje struktury sterujące właściwie wcięte aby pokazać strukturę programu. Wcięcia działają świetnie kiedy masz język z blokami strukturalnymi. Asembler jednak, jest to oryginalny niestrukturyzowany język i zasady wcięć dla

strukturyzowanego języka programowania po prostu nie mają zastosowania. Choć jest ważne aby móc oznaczyć ,że pewne sekwencje instrukcji są specjalne (no należą do części "then" instrukcji if..then..else..endif) wcięcia nie są właściwym sposobem uzyskiwania tego w asemblerze. Jeśli musisz ustawić sekwencję instrukcji dla kodu otaczającego, najlepszą rzeczą jaką możesz zrobić jest użycie pustych linii w kodzie źródłowym. Odzielenie jednego wyliczenia od innego ,na przykład pojedynczą pustą linią , jest wystarczające. To rzeczywiście pokazuje, że jedna sekcja kodu jest szczególna, używa dwóch, trzech lub nawet czterech pustych linii do odzielenia jednego bloku instrukcji od otaczającego kodu. Oddzielenie dwóch całkowicie niepowiązanych sekcji kodu, możesz użyć kilku pustych linii i wiersza z linią gwaizdek do oddzielenia tych instrukcji np.

```
        mov dx, FileSpec
        mov cl, 37h
        DOS FindFirstFile
        jc FileNotFound
;  ****************************************

        mov di, offset fileRecords          ;DI ->przechowanie nazw plików
        mov bx, offset files                ;BX -> tablica plików
        sub bx, 2                           ;Specjalny przypadke dla 1 iteracji
```

Wskazówka: Użyj pustych linii do odzielenia specjalnego bloku kodu od kodu otaczającego . Użyj estetycznie wyglądającej linii gwiazdek lub kresek jeśli potrzebujesz rozdzielenia między dwoma blokami kodu (nie przesadzaj jednak)

Jeśli dwie sekwencje instruckji asemblerowych odpowiada mniej więcej dwóch instrukcji HLL,jest dobrym pomysłem wstawienie pustej linii między dwoma sekwencjami. To pomaga wyjaśnić dwa segmenty kodu w umyśle czytelnika. Oczywiście, łatwo jest dać się ponieś i wstawić za dużo spacji w programie, więc używaj ich rozważnie.

Wskazówka: Jeśli dwie sekwencje kodu asemblerowego odpowiadają dwóm sąsiednim instrukcjom w HLL, wtedy użyj pustej lini do oddzielenia tych dwóch asemblerowych sekwencji (zakładajac ,że sekwencje są rzeczywiście krókie)

Powszechnym problemem w dowolnym języku (nie asemblerze) jest linia zawierająca komentarz związany z jedną lub dwoma liniami kodu. Takli program jest bardzo trudny do odczytania ponieważ trudno jest określić gdzie kończy się kod a zaczyna komentarz (lub vice-versa). Jest to szczególnie prawdziwe kiedy komentarz zawiera próbkę kodu. Jest to szczególnie trudne do określenia jeśli to co widzisz to jest kod czy komentarz.

Wymuszona Reaguła: Zawsze wstawiaj przynajmniej jedną pustą linię między kod komentarz (zakładając oczywiście ,że komentarz jest umieszczony sam w linii, tzn jest komentarzem kńcowym)

CZĘŚĆ VI: KOMENTARZE

Komentarze w programie asemblerowym generalnie ma dwie postacie: komentarz linii końcowej i kometarz samodzielny. Jak wskazują nazwy, komentarz linii końcowej zawsze występuje na końcu instrukcji źródłowej a kometarz samodzielny usadowiony jest w samej linii. Te dwa typy komentarzy mają odrębne cele.

6.1 Co to jest zły komentarz?
Zadziwiające jest jak wielu prgoramistów uważa ,że ich kod jest dobrze skomentowany. Czy można zliczyć znaki między (lub po) ograniczniku komentarza. Spójrzmy na poniższy komentarz:

```
mov     ax, 0           ;ustaw AX na zero
```

Szczerze mówiąc, ten komentarz jest gorszy niż w ogóle brak komentarza. Nie mówi czytelnikowi niczego ,czego nie można dowiedzieć się z samej instrukcji i wymaga od czytającego czasu aby dowiedział się ,że jest nic nie warta. Jeśli ktoś nie może odczytać ,że ta isntrukcja jest ustawieniem AX na zero, to nie ma sensu aby czytał kod programu asemblerowego.

Reguła: Wybierz grupę odbiorców dla swojego kodu źródłowego i pisz komentarze dla nich. Przy kodzie asemblerowym możesz założyć ,że docelowi odbiorcy są tymi którzy znają wystarczająco język asemblera.

Nie wyjaśniaj działania instrukcji asemblerowych w kodzie, chyba ,że instrukcja robi coś nieoczywistego (i większość czasu powinieneś rozglądać się za zmianami w sekwencji kodu jeśli nie jest oczywiste co on robi) .Zamaist tego wyjaśnij jak ta instrukcja jest pomocna dla rozwiązania danego problemu. Poniżej mamy znacznie lepszy komentarz:

```
mov             ax, 0           ;AX  to suma wynikowa. Zainicjuj go
```

Zwróć uwagę, że komentarz nie mówi "Zainicjuj go zerem". Chociaż nie byłoby w tym nic złego, wyrażenie "Zainicjuj go" pozostaje prawdą bez względu na wartość jaką można przypisać do AX. To uczyni zarządzanie kodem (i komentowaniem) dużo łatwiejszym ponieważ nie musisz zmieniać komentarza przy każdej zmianie stałej związanej z instrukcją.

Wskazówka: Pisz swoje komentarze w taki sposób aby główne zmiany w instrukcji nie wymagały zmian w odpowiednim komentarzu.

Notatka: Chociaż trywialny komentarz jest zły (rzeczywiście, gorzej niż brak komentarza wcale), Rozważmy poniższą instrukcję:

```
mov     ax, 1           ;Ustawia AX na zero
```

Zadziwiające jest jak długo zwykli ludzie patrząc na ten kod próbując odgadnąć jak program może ustawić AX na zero, skoro jest oczywiste ,że tego nie robi. Ludzie zawsze będą wierzyć komentarzom w kodzie. Jeśli jest jakaś nieścisłość między komentarzami a kodem, zakładają ,że kod jest coś nie taki, ale komentarz jest poprawny. Tylko po wyczerpaniu wszystkich możliwych opcji ,przeciętny człowiek dojdzie do wniosku ,że komentarz nie jest poprawny.

Wymuszona Reguła: Nigdy nie zezwalaj na niepoprawne komentarze w programie

Jest inny powód nie wstawiania trywialnych komentarzy takich jak "Ustawia AX na zero" w kodzie. Po zmodyfikowaniu programu ,te komentarze staną się najprawdopodobniej niepoprawne, po zmianie kodu komentarze są tudne do synchronizacji. Jednak, nawet pewne nietrywialne komentarze mogą stać się niepopawne przez zmianę kodu. Dlatego zawsze postępuj wedle tej reguły:

Wymuszona Reguła: Zawsze aktualizuj wszystkie komentarze, na które miał wpływ kod, bezpośrednio po dokonaniu zmian w kodzie

Niewątpliwie słyszałeś zdanie "upewnij się,że skomentowałeś kod tak jakby ktoś napisał go dla ciebie; inaczej w ciągu sześciu miesięcy będziesz życzył sobie go mieć". Zdanie to obejmuje dwa pojęcia. Po pierwsze nie myśl nigdy ,ze zrozumienie bieżącego kodu potrwa. Podczas pracy nad daną sekcją programu prawdopodobnie zainwestujesz sporo myśli i badań aby zrozumieć co się dzieje.. Sześć miesięcy później jednak , zapomnisz wiele z tego czego się dowiedziałeś ,a komentarze mogą przejść daleką drogę do uzyskania swego znaczenia.Po drugie kod ten sprawia ,że inni również odczytają i zapiszą kod. Jeśli przeczytasz czyjś kod, inni będą chcieli przeczytać twój. Jeśli piszesz kometarze w sposób w jaki chcesz aby inni napisali dla ciebie, są duże szanse ,że twoje komentarze będą pracować dla nich.

Reguła: Nigdy nie używaj rasistowskich, seksistowskich, obscenicznych lub innych niepoprawnych politycznie zwrotów w kometarzach. Niewątpliwie będziesz się wstydził takiego języka w swoich kometarzach w przyszłości. Ponadto wątpliwe, że taki język pomoże komuś lepiej zrozumieć program.

Dużo łatwiej jest podać przykłady złych komentarzy niż omawiać dobre komentarze. Poniższa lista opisuje niektóre z najgorszych możliwych komentarzy jakie możesz wstawić do programu (od najgorszych do ledwie tolerowalnych):

- Absolutnie najgorszy komentarz jaki możesz wstawić do programu to niepoprawny kometarz.

  ```
  mox    ax, 10;          {Ustawia  AX na 11}
  ```

  Zadziwiające jest jak wielu programistów automatycznie będzie zakładało ,że komentarz jest poprawnyi będzie próbowało odgadywać jak ten kod zarządza ustawieniem zmiennej "AX" na 1 kiedy jest oczywiste ,że ustawia ją na 10.

- Drugi najgorszy komentarz jaki możesz umieścić w programie to komentarz, który wyjaśnia co robi dana instrukcja. Typowym przykładem jest coś takiego jak "mov ax, 10 ; {Ustawia AX na 10}. W przeciwieństwie do poprzedniego przykładu, komentarz ten jest poprawny. Ale jest jeszcze gorszy niż brak komentarza ponieważ jest niepotrzebny i zmusza czytającego do spędzenia dodatkowego czasu do odczytania kodu (czas czytania jest proprocjonalny do trudności odczytywania). Czyni to również trudniejszym do zarządzania ponieważ niewielkie zmainy w kodzie (np "mov ax,9") wymaga zmodyfikowania komentarza, który w sumie jest niepotrzebny.
- Trzeci najgorszy komentarz w programie to komentarz nie związany z temate,. Mówiąc żartobliwie może wydawać się słodki ale to mało dla poprawienia czytelności programu;co więcej odwraca uwagę i koncentrację.
- Czwarty najgorszy komentarz to wogóle brak komentarza
- Piąty najgorszy komentarz to komentarz przestarzały lub nieaktualny (chociaż nie niepoprawny). Na przykład, komentarz na początku pliku może opisywać aktualną wersję modułu i kto ostatnio nad nim pracował. Jeśli ostatni programista zmodyfikował plik i nie zaktualizował komentarza , komentarze są teraz nieaktualne.

6.2 Komentarze końca linii kontra komentrze niezależne?

Wskazówka: Kiedy komentarz pojawia się sam w linii, zawsze wstawiaj średnik w kolumnie jeden. Możesz wciąć tekst jeśli jest to właściwe i estetyczne

Wskazówka: Przylegające linie komentarza nie powinny mieć przeplatanych pustych linii. Pusta linia komentarza powinna, przynajmniej, zawierać średnik w kolumnie jeden

Powyższe wskazówki sugerują aby twój kod wyglądał tak:

```
; To jest komentarz z pustą linią między nim a kolejnym komentarzem.
;
; To jest druga linia z komentarzem.
```

Zamiast tak

```
;  To jest komentarz z pustą linią między nim a kolejnym komentarzem.

;  To jest druga linia z komentarzem.
```

Średnik pojawił się między tymi instrukcjami sugerując kontynuację, która nie jest obecna kiedy usuniesz ten średnik. Jeśli dwa bloki komentarzy są naprawdę oddzielone a spacja między nimi jest właściwa, powinieneś rozważyć odseparowanie ich przez dużą liczbę pustych linii dla całkowitegoo wyeliminowania możliwych połączeń między nimi dwoma. Komentarze niezależne świetnie się nadają do opisania działań kodu występuje bezpośrednio po. Więc czy komentarze końca linii są przydatne? Komentarze te mogą wyjaśniać jak sekwencje instrukcji implementują algorytm opisany w poprzednim zbiorze komentarzy niezależnych. Rozważmy kod:

```
; Oblicz transpozycję macierzy przy użyciu tego algorytmu:
;
;       for i := 0 to 3 do
;               for j := 0 to 3 do
;                       swap( a[i][j], b[j][i] );
                forlp   i, 0, 3

                forlp    j, 0, 3

                mov     bx, i                   ;Oblicz adres a[i][j] używając
                shl     bx, 2                   ; porządek wierszy (i*4 + j)*2.
                add     bx, j
                add     bx, bx
                lea     bx, a[bx]
                push    bx                      ;Odłożenie adresu a[i][j] na stos.

                mov     bx, j                   ;Obliczanie adresu b[j][i] używając
                shl     bx, 2                   ;porządku wierszy (j*4 + i)*2.
                add     bx, i
                add     bx, bx
                lea      bx, b[bx]
                push    bx                      ;Odłożenie adresu b[j][i] na stos.

                call    swap                    ;Wymiana a[i][j] z b[j][i].

                next
                next
```

Zwróćuwagę,że blok komentarzy przed tą sekwencją wyjaśnia, w terminologi wysokopoziomowej, co ten kod robi. Komentarze końca linii wyjaśniają jak sekwencja instrukcji implementuje ogólny algorytm. Jednak zauważ ,że komentarze końca linii nie wyjaśniają co robi każda instrukcja (przynajmniej na poziomie maszynowym) .Zamiast twierdzić ,że "add bx, bx" to mnożenie wartości BX przez dwa, ten kod zakłada ,że czytelnik sma może się tego dowiedzieć (każdy rozsądny programista to wie). Poraz kolejny, zapamiętaj o odbiorcach i pisać komentatrze dla nich.

6.3 Kod nieskończony

Często jest to przypadek, kiedy programista będzie pisał sekcję kodu która (częściowo) osiąga jakieś zadanie ale potrzebuje dalszej pracy dla zakończnia zbioru funkcji, czyniąc go mniej wydajnym lub usunąć jakieś znane defekty w tym kodzie. To jest wspólne dla takich programistów, którzy umieszczają komentarze w kodzie takie jak "To wymaga dalszej pracy", "Prowizorka" itp. Problem z tymi komentarzami jest taki ,że są często zapominane. To nie jest tak dopóki. Najlepiej by było nie umieszczać takiego kodu w programie. Oczywiście, najlepiej , gdyby programy nie miały żadnych wad w sobie. Ponieważ taki kod nieuchronnie znajduje swoją drogę do programu, najlepiej mieć zasadę w miejscu działania z nim. Nieskończony kod dzieli sięna 5 ogólnych kategorii: kod niefunkcjonalny, kod częściowo funkcjonalny, kod podejrzany, kod potrzebujący naprawy i kod dokumentujący. Kod niefunkcjonalny może to być procedurą pośredniczącą lub driverem, który może być zastępiony w przyszłości, lub jakiś kod który ma dość poważne wady, który jest bezużyteczny z wyjątkiem małych specjalnych przypadków,. Ten kod jest rzeczywiście zły, na szczęście jego ciężkości zapobiegamy ignorując go. Jest mało prawdopodobne aby ktoś przegapił tak źle skonstruowany fragment kodu na wczesnym etapie testowania przed jego wypuszczeniem. Kod częściowo funkcjonalny jest być może największym problemem. Ten kod działa dość dobrze przechodząc jakieś proste testy zawierające jeszcze szereg wad ,które powinny być usunięte. Co więcej te wady nie są znane. Oprogramowanie często zawiera dużą liczbę nieznanych wad; to wstyd abyu jakieś znane wady wypłynęły w produkcie dlatego ,że programista zapomniał o wadach lub nie mógł ich później znaleźć. Kod podejrzany, jest tym na co wskazuje nazwa – podejrzanym kodem. Programista może nie być świadom kwantyfikowanego problemu ale może podejrzewać ,że istnieje problem. Taki kod będzie później przeglądany aby zweryfikować czy jest poprawny.Czwarta kategoria, kod potrzebujący naprawy jest najmniej poważny. Na przykład, aby przyspieszyć publikację, programista moze wybrać do użycia prosty algorytm zamiast złożonego , szybszego algorytmu. Można stworzyć komentarz w kodzie taki jak "To liniowe wwyszukiwanie powinno być zastąpione przez wyszukiwanie w tablicy rozproszone w przyszłych wersjach tego oprogramowania" Chociaż może to nie być konieczne do porawy tego problemu, będzie miło wiedzieć o takim problemie, który będzie rozwiązywany w przyszłości. Piąta kategoria, dokumentacja, odnosi się do zmian w oprogramowaniu, które będą wpływać na odpowiednią dokumentację (podręcznik użytkownika, dokumentacja projektowa itp).
Ten standard definiuje mechanizm dla działania z tymi pięcioma klasami problemów. Każde wystąpienie kodu nieskończonego będzie poprzedzone komentarzem, które przybierze jedną z następujących form (gdzie " _ " oznacza pojedynczą spację):

;_#defect#severe_;
;_#defect#functional_;
;_#defect#suspect_;
;_#defect#enhancement_;
;_#defect#documentation_;

Ważne jest aby użyć małych liter i zweryfikować poprawną pisownię więc łatwo znaleźć te komentarze używając wyszukiwania w edytorze tekstu lub narzędzia takiego jak grep. Oczywiście, oddzielny kometarz wyjaśniający sytuację musi poprzedzać te komentarze w kodzie źródłowym. Przykład:

; #defect#suspect ;
; #defect#enhancement ;
; #defect#documentation ;

Zauważ użycie ogranicznika komentarza (średnika) po obu stronach chociaż asembler tego nie wymaga.

Wymuszona Reguła: Jeśli moduł zawiera jakieś wady, które nie mogą być usunięte natychmiast z powodu czasu lub innych przeszkód, program będzie wstawiał ustandaryzowane komentarze przed kodem tak aby możne je było łatwo zlokalizować w przyszłości. Pięć znormalizowanych komentarzy to:

";_#defect#severe_;,
";_#defect#functional_;",
";_#defect#suspect_;",
";_#defect#enhancement_;",
";_#defect#documentation_;" gdzie "_" oznacza pojedynczą spację.
Odstępy i pisowania powinny być dokładne ,aby można je było łatwo wyszukać na drzewie źródłowym.

6.4 Odsyłacze w kodzie do innych dokumentów

W wielu przypadkach kod sekcji może być nierozerwalnie związany z jakimś innym dokumentem. Na przykład możesz przenieść czytającego do dokumantacji dla użytkownika wewnątrz komentarzy w programie. Proponuję standardowy sposób zrobienia tego aby można było relatywnie łatwo zlokalizować odniesienia pojawiające się w kodzie źródłowym. Ta technika jest podobna do tej z raportowania wad, z wyjątkiem komentarza przybierającego postać:

; tekst #link# położenie tekst;

Tekst jest opcjonalny i przedstawia jakiś tekst (chociaż w rzeczywistości jest przeznaczony dla osadzania poleceń html dla hiperłączy do określonego dokumentu) "Położenie" opisuje dokument i sekcję gdzie może być znaleziona określona informacja. Przykłady:

; #link#User's Guide Section 3.1 ;
; #link#Program Design Document, Page 5 ;
; #link#Funcs.pas module, "xyz" function ;
; <A HREF="DesignDoc.html#xyzfunc"> #link#xyzfunc ;

Wskazówka: Jeśli moduł zawiera jakieś oodniesienia do innych dokumentów, powinny znajdować się komentarze przybierające postać ";tekst #link# położenie tekst"; które dostarczają odniesienia do innych dokumentów. W tym komentarzu tekst przedstawia jakiś opcjonalny tekst (zwykle zarezerwowane tagi html'a) a "położenie" jest opisowym tekstem, który opisuje ten dokument (i pozycję w tym dokumencie) powiązany z bieżącą sekcją kodu w programie.

CZĘŚĆ VII: NAZWY, INSTRUKCJE, OPERATORY I OPERANDY

Chociaż cechy programu takie jak dobry komentarz, właściwe odstępy instrukcji i dobra modularyzacja mogą pomóc programom stać się czytelniejszymi; ostatecznie programista musi przeczytać instrukcję aby zrozumieć co ona robi. Dlatego, nie należy lekceważyć tworzenia instrukcji tak czytelnymi jak to możliwe.

7.1 Nazwy

Według badań przeprowadzonych przez IBM, użycie wysokiej jakości identyfikatorów w programie bardziej przyczynia się do czytelności tego programu niż jaki kolwiek inny czynnik, w tym wysokiej jakości komentarz. Jakość identyfikatorów może porawić lub pogorszyć twój program; program z wysokiej jakości identyfikatorami może być bardzo łatwy do odczytania,

program ze złymi identyfikatorami będzie trudny do odczytania. Jest kilka "sztuczek" związanych z tworzeniem wysokiej jakości nazw; większość reguł to nic więcej jak staromodny zwykły rozsądek. Niestety programiści (zwłaszcza programiści C/C++) opracowali wiele arkan nazewnictwa, któe ignorują zdrowy rozsądek. Największą przeszkodą dla programistów któzy muszą nauczyć się jak tworzyć dobre nazwy jest niechęć do rezygnacji z istniejących konwencji. Jednak ich jedyną obroną, gdy pytani są , dlaczego są one zgodne z (istniejącymi) złymi konwencjamii ,wydaje się być "bo to jest tak, jak zawsze robiliśmy i to jest tak jak wszyscy to robią." Naukowcy z IBM opracowali kilka programów z następującymi zbiorami atrybutów:

- Zły komentarz, zła nazwa
- Zły komentarz, dobra nazwa
- Dobry komentarz, zła nazwa
- Dobry komentarz, dobra nazwa

Powinno być oczywiste, że program który ma złe komentarze i nazwy byłby najtrudniejszy do odczytania; podobnie te programy z dobrymi komentarzami i nazwami są łatwiejsze do odczytania. Zaskakujące wyniki dotyczyły dwóch pozostałych przypadków. Większość ludzi zakłada ,że dobre komentarze są ważniejsze niż dobra nazwa w programie. Jak się okazuje dobre nazwy są nawet ważniejsze niż dobre komentarze w programie. Nie powiedziano ,że komentarze nie są ważne, są wyjątkowo ważne; jednak warto wskazać ,że jeśli spędzasz dużo czasu pisząc dobre komentarze a potem wybierasz złe nazwy dla identyfikatorów programu, masz nasruszoną czytelność programu mimo pracy włożone w swoje komentarze. Przeczytaj szybko ten kod:

```
mov         ax, SignedValue
cwd
add          ax, -1
rcl          dx, 1
mov         AbsoluteValue, dx
```

Pytanie: Co ten kod wylicza i co przechowuje w zmiennej AbsoluteValue?

- Znak rozszerzenia SignedValue
- Negację SignedValue
- Wartość absolutną SignedValue
- Wartość logiczną wskazującą wynik dodatni lub ujemny
- Signum(SignedValue)(-1,0,+1 jeśli ujemen, zero , dodatnie)
- Ceil(SignumValued)
- Floor(SignumValued)

Oczywista odpowiedzią jest wartość absolutna SignedValue. To jest również niepoprawne.
Poprawna odpowiedź to signum:

```
mov     ax, SignedValue          ;Pobranie wartości do sprawdzenia.
cwd                              ;DX = FFFF jeśli ujemne, 0000 inaczej.
add     ax, 0ffffh               ;Przeniesienie=0 jeśli ax to zero, 1 inaczej.
rcl     dx, 1                    ;DX = FFFF jeśli AX jest ujemn neg, 0 if ax=0,
mov     Signum, dx               ; 1 jeśli ax>0.
```

To prawda, to trudny kawałek kodu. Mimo to nawet bez komentarzy prawdopodobnie można dkryć co ta sekwencja kodu robi jeśli nie możesz się zorientować jak on to robi.

```
mov     ax, SignedValue
cwd
add     ax, 0ffffh
rcl     dx, 1
mov     Signum, dx
```

W oparciu o same nazwy możesz prawdopodobnie odkryć ,że ten kod wylicza funkjcę signum. To jest to "80% zrozumienia kodu" jak wspominałem wcześniej. Zwróć uwagę ,że nie trzeba mylących nazw aby zaciemnić kod. Rozważmy kod ,który nie korzysta z mylących nazw:

```
mov     ax, x
cwd
add     ax, 0ffffh
rcl     dx, 1
mov     y, dx
```

Jest to bardzo prosty przykład.Teraz wyobra sobie duży program, który ma wiele nazw. Jeśli liczba nazw zwiększa się w programie,staje się trudniejszy w śledzeniu ich wszystkich. Jeśli same nazwy nie stanowią dobrych wskazówekco do znaczenia, zrozumienie programu staje się trudne.

Wymuszona Reguła: Wszystkie identyfikatory pojawiające się w programie asemblerowym muszą być opisowymi nazwami których znaczenie i użycie jest jasne

Ponieważ etykiety (tj identyfikatory) są celami instrukcji jmp i call, typowy program asemblerowy będzie miał dużą liczbę identyfikatorów. Dlatego też, kuszące jest aby używać nazw Label1, Label2, Label3 itd... Unikaj tego!

Reguła: Nigdy nie używaj nazw takich jak "Lbl0, Lbl1,Lbl2,..." w programie

7.1.1 Konwencje nazewnicze

Konwencje nazewnicze przedstawiają jeden obszar w Informatyce gdzie jest zbyt wiele różnych poglądów (układ programu jest innym pryncypialnym obszarem). Podstawowym celem nazwy obiektu w języku programowania jest opis użycia i/lub zawartości tego obiektu.Drugorzędne znaczenie może mieć opis typu obiektu. Programiści używają różnych mechanizmów do obsługi tych celów. Niestety, jest stanowczo za wiele "konwencji". Zdecydowana większość programistów zna tylko angielski. Dla innych programistów angielski to drugi język i mogą nie być zaznajomieni z powszechnymi nie angielskimi frazami, których nie ma w ich ojczystym języku (np rendezvous) Ponieważ angielski jest popularnym językiem wśród programistów, wszystkie identyfikatory powinny używać łatwo rozpoznawalnychangielskich słów i fraz.

Reguła: Wszystkie identyfikatory które przedstawiają słowa lub frazy muszą być słowami i frazami angielskimi

7.1.2 Rozważania nad wielkością liter

Identyfikatory neutralne na wielkośćliter będą pracowały poprawnie kiedy kompilujesz kompilatorem czułym na wilekość liter identyfikatorów czy też nie. W praktyce oznacza to że wszystkie identyfikatory muszą być literowane dokładnie w ten sam sposób i że żaden inny identyfikator nie istnieje któy tylko różni się wielkością liter w identyfikatorze. Na przykład, jeśli deklarujesz identyfikator "ProfitsThisYear" w Pascalu (język nie czuły na wielkość liter), możesz poprawnie się odnieść do tej zmiennej jako "profitsThisYear" i "PROfiTSTHISYEAR". Jednak , nie jest to neutralne użycie ponieważ jezyki czułe na wielkość liter będą traktowały te trzy identyfikatory jako różne nazwy. Natomiast, w językach czułych na wielkość liter takich jak C/C+

+, możliwe jest stworzenie dwóch różnych identyfikatorów z nazwą "PROFITS" i "profits" w programie. To też nie jest neutralen ze względu na wielkość liter ponieważ próba użycia tych dwóch identyfikatorów w językach nieczułych na wielkośćliter (jak Pascal) tworzy błąd ponieważ jezyki czułe na wielkośćliter będą sądzić ,że to to ta sama nazwa.

Wymuszona Reguła: Wszystkie identyfikatory muszą być "neutralne co do wielkości liter"

Różni programiści (szczególnie w różnych językach) używają wielkości liter do oznaczania różnych obiektów.Na przykład, konwencja kodowania C/C++ jest używana do wsyztskich dużych liter do oznaczenia stałych, makr lyb definicji typu i używa małych liter do oznaczania nazw zmiennych lub zarezerwowanych słów. Programiści używaja inicjującychmałych liter do ozanczania zmiennych. Istnieją kompatybilne konwencje kodowania. Niestety jest wiele różnych konwencji ,które używają wielkości liter są prawie bezwartościowe

Reguła: Nigdy nie używaj wielkości liter do ozanczania typu, klasyfikacji lub innych programów (chyba ,że składania języka określa takie wymaganie)

Wilekość liter ma jeden użyteczny cel dla identyfikatorów – jest użyteczne dla oddzielania słów w identyfikatorze składającym się z wielu słów; nie możemy stosować tej techniki przy identyfikatorach. Niestety pisanie wielowyrazowych identyfikatorów czyni niemożliwym odczytywanie jeśli nie uczynimy nic dla odróżnienia pojedynczych słów. Jest parę dobrych konwencji dla rozwiązania tego problemu. Standardową konwencją jest duża litera pierwszego znaku każdego słowa w środku identyfikatora

Reguła: Używaj dużej pierwszej litery wewnętrznych słów w identyfikatorze wielowyrazowych

Zwróć uwagę ,że powyższa zasada nie określa czy pierwsza litera identyfikatora jest dużą czy małą literą.
Małe litery są łatwiejsze do odczytania niż duże. Identyfikatory pisane całkowicie dużymi literami wymagają dwukrotnie więcej uwagi i dlatego pogarszają czytelność programu. Duże litery wyróżniają identyfikator. Taki nacisk jest raczej rzadki w rzeczywistych programach.

Reguła: Unikaj używania samych dużych liter w identyfikatorze

7.1.3 Skróty

Podstawowym celem identyfikatora jest opis użytkowania lub wartości z nim związanej. Najlepszym sposobem tworzenia identyfikatorów dla obiektów jest opis danego obiektu po angielsku a potem stworzenie nazwy zmiennej dla tego opisu. Nazwy zmiennych powinny być konkretne, zwięzłe i jednoznaczne,dla programisty przeciętnie znającego angielski Unikajmy krótkich nazw. Niektóre badania pokazują ,że programy używające identyfikatorów których średnia długość to 10 – 20 znaków są generalnie łatwiejsze do debuggowania niż programy ze znacznie krószymi lub dłuższymi identyfikatorami. Unikajmy skrótów jak to tylko możliwe. Może się wydawać ,że skrót całkowicie czytelny dla ciebie może być całkowicie nieroumiały dla kogoś innego. Rozważmy nazwy zmiennych, które rzeczywiście pojawiły się w oprogramowaniu komercyjnym:
NoEmployees, NoAccounts, pend

Zmienne "NoEmployees' i "NoAccounts " wydają się być zmiennymi logicznymi wskazującymi obecność lub brak pracowników i kont. Faktycznie, te określony programista użył (całkowicie uzsasadnionego w świecie rzeczywistym) skrótu do "number" wskazującego liczbę pracowników i numerów kont. Nazwa "pend" odnosi się do końca procedury niż jakiejś operacji w toku.

Programiści często używają skrótów w dwóch sytacjach: słabo im idzie pisanie i chcą zredukować wysiłek przy pisaniu lub dobra nazwa opisowa dla obiektu jest po prostu zbyt długa. W pierwszym przypadku niedopuszczalne jest używanie skrótów. W drugim przypadku, zwłaszcza jeśli zwróci się uwagę, można okazjonalnie mieć korzyść ze skrótu.

Wskazówka: Unikaj wszelkich skrótów identyfikatorów w programach. Kiedy jest to konieczne, użyj ustandaryzowanych skrótów lub poproś kogoś aby sprawdził skróty. Jeśli użyjesz skrótów w programie, stwórz "słownik danych" w komentarzu, przy definicji nazwy, który dostarczy pełnej nazwy i opisu dla tego skrótu.

Nazwy zmiennych jakie tworzysz powinny być do wymówienia. "NumFiles" jest dużo lepsze niż "NmFls". Pierwsza może być wymówiona, druga musi być już przeliterowana Unikaj homonimów i długich nazw, które są identyczne z wyjątkiem kilku sylab. Jeśli wybrałeś dobre nazwy dla identyfikatorów, powinieneś móc odczytać listing programu przez telefon bez problemu dla słuchającego.

Reguła: Wszystkie identyfikatory powinny być do wymówienia (po angielsku) bez przeliterowania więcej niż jednej litery

7.1.4 Pozycja elementów wewnątrz identyfikatora

Kiedy przeglądają listing, większość programistów tylko czyta pierwsze kilka znaków identyfikatora. Dlatego jest ważne umieszczanie najważniejszych rzeczy (które definiują i czynią ten idenyfikator unikalnym) w pierwszych kilku znakach tego identyfikatora. Powineneś unika tworzenia identyfikatorów ,które będą zaczynały się od tej samej frazy lub sekencji znaków poneważ wymusi to na programiście mentalne przetwarzanie dodatkowych znaków w identyfikatorze podczas czytania listingu. Ponieważ to spowalnia czytającego, czyni program trudniejszym do odzytania.

Wskazówka: Spróbuj tworzyć bardziej unikalne identyfikatory na pierwszych kilku pozycjach znaków. Czyni to program łatwiejszym do przeczytania.

Następstwo: Nigdy nie używaj przyrostków numerycznych dla rozróżniania nazw.

Wielu programistów, szczególnie programistów Microsoft Windows, zaadoptowało formalną konwencję nazewniczą znaną jako "Notacja Węgierska". Termin "węgierska" odnosi się do faktu ,że nazwy w tej konwencji wyglądają jak słowa w obcym języku i tego ,że jej twórca Charles Simonyi był Węgrem. Jedna z pierwszych zasad odnośnie identyfikatorów mówiła ,ze wszystkie identyfikatory powinny nosić angielskie nazwy. Czy rzeczywiście chcemy stworzyć "sztuczą obcość " identyfikatorów? Notacja węgierska w rzeczywistości łamie również inną regułę: nazwy używające notacji węgierskiej generalnie mają dużo wspólnnych przedrostków, czyli czynią je trudnymi do odczytania. Węgierska notacja ma kilka dornych zalet ,ale wady przewyższają te zalety. Poniższa list opsiuje wady notacji węgieskiej:

Notacja węgierska generalnie definiuje obiekty w zakresie podstawowych typów maszynowych zamiast w kategoriach abstrakcyjnych typów danych.
Notacja węgierska łączy znaczenie z reprezentacją. Jednym z podstawowych celów języka wysokopoziomowego jest przedstawienie abstrakcji. Na przykład, jeśli deklarujesz zmienne typu integer, nie powinieneś zmieniać nazwy zmiennej ponieważ zmieniłeś jego typ na real.
Notacja węgierska zachęca do leniwych, nieinformacyjnych nazw zmiennych. Rzeczywiście powszechne jest znajdowanie w programach Windows nazw zmiennych, które zawierają tylko wpisane znaki przedrostków bez dołączonych nazw opisowych.

Wskazówka: Unikaj notacji węgierskiej i innych formalnych kpnwencji nazewniczych, które dołaczają informację typu niskopoziomowego do identyfikatorów.

Chociaż dołączanie informacji typu maszynowego ogólnie jest złą ideą, dobrze przemyślana nazwa może z powodzeniem skojarzyć kilka informacji typu wysokopoziomowego z identyfikatorem, zwłaszcza jeśli nazwa implikuje typ lub wskazuje typ informacji pojawiający się jako rozszerzenie. Na przykład nazwa "PencilCount" i "BytesAvailable" sugeruje wartości całkowite. Podobnie nazwa "IsReady" i "Busy" wskazuj wartości logiczne. "KeyCode" i "MiddleInitial" sugeruje zmienne znakowe. Nazwa taka jak "StopWatchTime" prawdopodobnie wskazuje wartość rzeczywistą. Podobnie "CustomerName" jest prawdopodobnie zmienną łańcuchową. Niestety, nie zawsze jest możliwe wybranie dobrej nazwy, która opisuje zarówno zawartość jak i typ obiektu, jest to szczególnie prawdziwe kiedy obiekt jest instancją (lub definicją) jakiegoś abstrakcyjnego typu danych. W takiej instancji, pewne dodatkowe teksty mogą poprawiać identyfikator. Notacja węgierska to surowa próba zrobienia tego, niestety nieudana z wielu powodów. Lepszym rozwiązaniem jest użycie przyrostka zwrotu oznaczającego typ lub klasę identyfikatora. Powszechną konwencją UNIX/C jest stosowanie przyrostka "_t" dla oznaczenia nazwy typu (np size_t, key_t itp). Konwencja ta ma przewagę nad notacją węgierską z kilku powodów: (1) "typ frazy" jest przyrostkiem i nie wpływa na czytanie nazwy, (2) ta określona konwencja określa kalsę obiektu (const, var, type , function itp) zamiast niskopoziomowego type i (3)Z pewnością ma sens zmiana identyfikatora jeśli zmieni się klasyfikacja

Wskazówki: Jeśli chcesz rozróżniać identyfikatory, które są stałymi, typami definicji i nazwy zmiennych używają przyrostka "_c", "_t" i "_v", odpowiednio.

Reguła: Klasyfikacja przyrostka nie powinna być tylko elementem, który rozróżnia identyfikatory

Czy możemy stosować tą idee przyrostków do zmiennych i unikać pułapek? Czasami. Rozważmy typ danych wysokopoziomowy "button" odpowiadający przyciskowi na formatce Visual BASIC lub Delphi. Zmienna o nazwie "CancelButton" ma właściwy sens. Podobnie, etykiety pojawiające się na formatce mogą używać nazw takich jak "ETTWWLabel" i "EditPageLabel" Zwróóć uwagę ,że te przyrostki cierpią z powodu faktu ,że zmiana typu będzie wymagała zmiany nazwy zmiennej. Jednak zmiana w typach wysokopoziomowych jest dużo mniej powszechna niż zmiany w typach niskopoziomowych, więc nie powinno to być dużym problemem.

7.1.5 Nazwy których należy unikać

Unikanie używania symboli w identyfikatorze, które łatwo pomylić z innymi symbolami. Obejmuje to zbiór {"1" (jeden), "I" (duże "I") i "l" (małe "L"), {"0" (zero) i "O" (duże "O")}, {"2" (dwa) i "Z" (duże "Z")}, {"5" (pięć) i "S" (duże "S")} i {"6" (sześć) i "G" (duże "G")}

Wskazówka: Unikaj symboli w identyfikatorach, które są łatwo mylone z innymi symbolami

Należy unikać mylących nazw i skrótów. Na przykład, FALSE nie powinno być identyfikatorem ,który oznacza "Failed As a Legitimate Software Engineer". Podobnie, nie powinieneś wyliczać ilość wolnej pamięci dostępnej dla programu i zmienić nazw na "Profits"

Reguła: Unikaj mylących skrótów i nazw

Powinieneś unikać nazw o podobnym znaczeniu. Na przykład, jeśli masz dwie zmienne "InputLine" i "InputLn", jakiej użytkownik używa dla dwóch oddzielnych celów, będziesz niewątpliwie mylił się przy tych dwóch kiedy będziesz pisał lub odczytywał kod. Jeśli możesz zmienić nazwy dwóch obiektów a program ma jeszcze sens, powinieneś zmienić nazwy tych identyfikatorów. Zwróć uwagę,że nazwy nie muszą być podobne, tylko ich znaczenie. "InputLine" i "LineBuffer" są

oczywistymi, różniącymi się ale możesz dalej je mylić w programie.

Reguła: Nie używaj nazw o podobnych znaczeniach dla różnych obiektów w programach

W podobnym duchu powinieneś unikać dwóch lub więcej zmiennych, które mają różne znaczenia ale podobne nazwy. Na przykład, jeśli piszesz program do oceny dla nauczycieli, prawdopodobnie nie będziesz chciał użyć nazwy "NumStudent" aby wskazać liczbę studentów w klasie razem ze zmieną "StudentNum" przechowującą numer identyfikacyjny studenta."NumStudent" i "StudentNuym" są bardzo podobne.

Reguła: Nie używaj podobnych nazw, które mają różne znaczenia

Unikaj nazw ,które brzmią podobnie kiedy czytasz głośno,zwłaszcza z kontekstu.Na przykład takie jak "hard" i "heart", "Knew" i "new" itp. Pamiętaj co pisałem o skrótach w ostatniej sekcji, powinieneś omówić taki poblem z kimś przez telefon.

Reguła: Używaj homonimów w identyfikatorach

Unikaj słów źle napisanych w nazwach i nazw, które są powszechnie błednie zapisane. Większość programistów notorycznie robi błędy ortograficzne (spójrz do swoich własnych komentarzy!) Literuj słowa poprawnie, pamiętając ,że nieporawny identyfikator jest nawet trudniejszy do napisania.

Wskazówka: Unikaj źle napisanych słów i nazw, które są błędnymi identyfikatorami

Jeśli definiujesz ponownie nazwę jakiegoś podprogramu bibliotecznego w swoim kodzie, inny program będzie z pewnością mylił twoją nazwę z wersją biblioteczną. Jest to prawdziwe kiedy działamy z podprogramami bibliotecznymi i API.

Wymuszona Reguła: Nie używaj ponownie istniejących nazw podprogramów bibliotecznych w swoich programach chyba ,że specjalnie zastąpiłeś ten podprogram takim który ma podobną semantykę.

7.2 Instrukcje, dyrektywy i pseudo – opkody

Twój wybór sekwencji asemblerowych, samych instrukcji i wybór dyrektyw i pseudo-opkodów ma duży wpływ na czytelność twoich programów.

7.2.1 Wybór najlepszej sekwencji instrukcji

Podobnie jak dowolnym języku możesz rozwiązać dany problem używając szerokiego wachlarza rozwiązań wykorzystując różne sekwencje instrukcji. Jako przykład rozważmy taką sekwencję kodu:

```
mov     ax, SignedValue     ;Pobranie wartości do sprawdzenia
cwd                         ;DX = FFFF jeśli ujemne, 0000 inaczej.
add     ax, 0ffffh          ;Przeniesienie=0 jeśli ax jest zerem.
rcl     dx, 1               ;DX = FFFF jeśli AX jest ujemne, 0 jeśli AX=0,
mov     Signum, dx          ; 1 jeśliAX>0.
```

Teraz rozważmy poniższą sekwencję kodu, która również oblicza funkcję signum:

```
mov     ax, SignedValue     ;Pobranie wartości do sprawdzenia
cmp     ax, 0               ;Sprawdzenie znaku.
```

```
            je      GotSignum       ;Robimy jeśli zero.
            mov     ax, 1           ;Zakładamy ,że dodatnie.
            jns     GotSignum       ;Skok jeśli dodatnie.
            neg a   x ;             ;Zwraca -1 dla wartości ujemnych.
GotSignum:  mov     Signum, ax
```

Tak, druga wersja jest dłuższa i wolniejsza. Jednak, większość osób może odczytać sekwencję instrukcji i odkryć co robi; zatem druga wersja jst dużo łatwiejsza do odczytania niż pierwsza. Która sekwencja jest lepsza? Jeśli szykość i miejsce nie są najbardziej krytycznymi czyunnikami i możesz wykazać ,że ten podpogram jest ścieżką krytcyznego wykonania, wtedy druga wersja jest oczywiście lepsza. Więc w jaki sposób można wybrać odpowiednie sekwencje instrukcji, gdy istnieje wiele możliwych sposobów, aby osiągnąć ten sam cel? Najlepszym sposobem jest upewnienie się ,że masz wybór. Chociaż jest wiele różnych sposobów wykonania operacji, kilka osób stara się rozważąć sekwencję instrukcji inną niż pierwsza która przychodzi do głowy. Niestety, "najlepsza" sekwencja instrukcji jest rzadką pierwszą sekwencją instrukcji ,która przychodzi większości ludziom przychodzi do głowy. Aby mieć wybór, musisz mie wybór tworzenia. Oznacza to ,że możesz stworzyć przynajmniej dwie różne sekwencje kodu dla danej operacji jeśli jest zawsze pytanie dotyczące o czytelność kodu. Kiedy masz przynajmniej dwioe wersje, możesz wybierać między nimi w oparciu o towje potrzeby. Chociaż jest niepraktyczne "pisanie programu dwukrotnie", będziesz miał wybór dla każdej sekwencji instrukcji w programie, powinieneś stosować tę technikę do szczególnie uciążliwych sekwencji kodu.

Wskzówka: Dla szczególnie trudnych niezrozumiałych sekcji kodu, spróbuj rozwiązać problem na kilka różnych sposobów. Wybierz wtedy najłatwiej zrozumiałe rozwiązanie dla faktycznego włączenia do programu.

Jeden problem z powyższą wskazówką jest taki,że często opierasz swoją pracę od decyzji "ten kod nie jest zbyt trudny do zrozumienia, nie muszę się martwić o to"

Wskazówka: Skorzystaj z podglądu do określenia tych sekcji kodu w programie, które mogą być ponownie napisane aby uczynić je łatwiejszym do zrozumienia

7.2.2 Struktury sterujące

Asembler można podsumować tak: "Język asemblera czyni trudnym napisanie łatwego programu i łatwego napisania trudnego do odczytania programu" Potrzeba znacznej dyscypliny aby napisać czytelny program w asemblerze, ale można to zrobić. Niestety, większość kodu asemblerowego można nawet dzisiaj znaleźć źle napisanego. Rzeczywiście, taki stan spraw jest całkowitym powodem tego dokumentu. Kiedy pominiemy kwestie takie jak komentarze, konwencję nazewnictw, sprawy takie jak kontrola przepływu programu i struktury danych mają największy wpływ na czytelność programu.Ponieważ większość asembleró buduje brakującą strukturę kontroli przepływu, jest jeden obszar gdzie niezdyscyplinowani programiści rzeczywiście mogą pokazać jak smutny piszą kod. Na szczęście, przy małej dyscyplinie możliwe jest napisanie czytelnego kodu programów asemblerowych. Jak zaprojektować strukturę sterującą ,która może mieć duży wpływ na czytelność programów. Najlepszym sposobem zrobienia tego może być zsumowanie dwóch słóww : unikaj spaghetti. Spaghetti kod jest nazwą nadawaną programom, które mają dużą liczbę poprzeplatanych skoków i skoków docelowych wewnątrz sekwencji kodi. Spójrzmy na poniższy przykład:

```
            jmp L1
L1:         mov ax, 0
            jmp L2
L3:         mov ax, 1
```

```
                    jmp L2
L4:                 mov ax, -1
                    jmp L2
L0:                 mov ax, x
                    cmp ax, 0
                    je L1
                    jns L3
                    jmp L4
L2:                 mov y, ax
```

Ta sekwencja kodu, swoją drogą, jest baszym starym znajomym, funkcją Signum. Zabierze trochę czasu nim odkryjemy to ponieważ ręczne śledzenie kodu zajmuje sporo czasu. Teraz jest to ekstremalny przykład, ale jest również stosunkowo krótki. Kod spaghetti został nadany temu ponieważ przypomina miskę spaghetti. To znaczy, czy rozważamy ścieżkę kontroli w programie makaronu spaghetti, kod spaghetti zawiera wiele splecionych gałęzi do i z różnych sekcji programu. Nie trzeba dodawać ,że większość programów spaghetti jest trudna do zrozumienia, generalnie zawierają wiele błędów i są często niewydajne (nie zapomnij ,że rozgałęzienia są jednymi z najwolniejszych instrukcji w nowoczesnych procesorach). Więc jak to rozwiązać? Najłatwiej przez fizyczne zaadaptowanie technik programowania struktur w kodzie asemblerowym. Oczywiście assembler 80x86 nie dostarcza if..then..else..endif, while...endwhile, repeat..until i innych takich instrukcji, ale możemy je zasymulować. Rozważmy poniższą sekwencję kodu w językach wysokopoziomowych:

```
if(wyrażenie)           then
        << instrukcje do wykonania jeśli wyrażenie jest prawdziwe >>
else
        <<  instrukcje do wykonania jeśli wyrażenie jest fałszywe >>
endif
```

Prawie w każdym prgoramie wysokiego poziomu można dowiedzieć się co ten kod robi. Programiści asemblerowi powinni wykorzystać tą wiedzę przez wybróbowanie organizacji ich kodu w tej samej formie. Wersja asemblerowa powinna wyglądać mniej więcej tak:

```
        << Kod asemblerowy do wylicznie wartości wyrażenia >>

        JNxx        ElsePart ;xx jest przeciwieństwem warunku jaki chcemy sprawdzać.

        << Kod asemblerowy odpowiedni dla tej części >>

        jmp         AroundElsePart
ElsePart:
        << v Kod asemblerowy odpowiedni dla tej części >>

AroundElsePart:
```

Jako konkretny przykład rozważmy to :

```
        if ( x=y ) then
                write( 'x = y' );
        else
                write( 'x <> y' );
```

```
        endif;
; Odpowiedni kod asemblerowy:
                        mov ax, x
                        cmp ax, y
                        jne ElsePart

                        print "x=y",nl
                        jmp IfDone

ElsePart:               print "x<>y",nl
IfDone:
```

Chociaż może się to wydawać oczywistym sposobem organizowania instrukcji if..then..else..endif, niespodzianką jest jak wielu ludzi naturalnie zakłada umieszczanie części else gdzieś w programie, jak poniżej:

```
                mov     ax, x
                cmp     ax, y
                jne     ElsePart

                print "x=y",nl
IfDone:
                .
                .
                .
ElsePart:       print "x<>y",nl
                jmp IfDone
```

Taka orgaznizacja kodu czyni program trudniejszym. Większość programistów ma w tle HLL i pomimo bieżącego przypisania, nadal głównie pracują pod HLL'ami. Program asemblerowy będzie bardziej czytelny jeśli będzie naśladował formy sterujące z HLL. Z podobnych powodów powinieneś spróbować zorganizować kod asemblerowy, który symuluje pętle while, repeat...until, for itd,aby kod był podobny do kodu HLL (na przykład, pętla while powinna fizycznie przetestować warunek na początku pętli, która skacze na dół pętli)

Reguła: Próbuj projektowania programów korzystając ze struktur sterujących HLL. Organizuj kod asemblerowy, który piszesz tak aby powinien fizycznie być zorganizowany podobnie jak odpowiedni program HLL.

Asembler oferuje elastyczność projektowania dowolnych struktur sterujących. Ta elastyczność jest jednym z powodów w jaki dobry programista asemblerowy może pisać lepszy kod niż stworzony przez kompilator (który może działać tylko z wysokopoziomowymi strukturami). Jednak zapamiętaj ,że szybki program nie musi zawierać ścisły możliwy kod w każdej sekwencji. Szybkość wykonania jest prawie bez znaczenia w większej części programu. Poświęcenie czytelności na rzecz szybkości nie jest najlepszym sposobem w większości programów.

Wskazówka: Unikaj struktur sterujących które niezbyt łatwo odwzorowują dobrze znane z języków wysokopoziomowych struktury sterujące w swoich programach asemblerowych. Odbiegające od normy struktury sterujące powinny się pojawiać tylko w małej części kodu keidy wymaga ich zastosowania wydajność.

7.2.3 Synonimy instrukcji

MASM definiuje kilka synonimów dla popularnych instrukcji. Jest to szczególnie prawdziwe dla skoków warunkowych i instrukcji "ustawianych na kod warunku". Na przykład, JA i JNBE są synonimami . Logicznie, jedna może używać drugiej w tym samym kontekście. Jednak, wybór synonimów może mieć wpływ na czytelność sekwencji kodu. Zobaczmy to:

```
                if( x <= y ) then
                        << instrukcje  dla true>>
                else
                        << instrukcje dla false>>
                endif
;Kod asemblerowy:
                        mov ax, x
                        cmp ax, y
                        ja ElsePart
                        << kod true >>
                        jmp IfDone

ElsePart:               << kod false >>
IfDone:
```

Kioedy ktoś czyta ten program, instrukcja "JA" skacze przez część true. Niestety, instrukcja "JA" daje iluzję, że sprawdzamy aby zobaczyć czy cośjest większe niż coś innego; w rzeczywistości testujemy czy jakiś warunek jest mniejszy lub rony ,nie większy niż. Jako taki, ta sekwencja kodu ukrywa pierwotne intencje algorytmu wysokopoziomwego. Jedynym rozwiązaniem jest przesunięcie części false i true kodu:

```
                mov ax, x
                cmp ax, y
                jbe ThenPart
                << kod false >>
                jmp IfDone

ThenPart:       << kod true >>
IfDone:
```

Ta sekwencja kodu używa skoku warunkowego pasującego do testu algorytmu wysokopoziomowego (mniejszy niż lub równy). Jednak kod ten jest teraz zorganizowany w stylu niestandadowym (jest instrukcja if...else...then...endif). To bardziej rani czyetlność niż użycie właściwego pomocnego skoku. Rozważmy poniższe rozwiązanie:

```
                mov             ax, x
                cmp             ax, y
                jnbe             ElsePart
                << kod true >>
                jmp             IfDone
ElsePart:       << kod false  >>
IfDone:
```

Ten kod jest zorganizowany w tradycyjny sposób if..then..else..endif. Zamiast użycia JA do przeskoczenia nad tą częścią, użyliśmy JNBE do zrobienia tego. Pomaga to wskazać, w bardziej czytelnej modzie, że kod nie dojdzie do skutku jeśli jest poniżej lub równy i wystąpi skok jeśli nie jest poniżej lub równy. Ponieważ ta instrukcja (JNBE) jest łatwiejsza w odniesieniu do

oryginalnego testu (<=) niż JA, czyni to tą sekcję kodu trochę mniej czytelną

Reguła: Kiedy przeskakujemy nad kodem ponieważ jakiś warunek jest błędny, zawsze używaj skoku warunkowego w postaci "Jnxx" do przeskoczenia nad tą sekcją kodu. Na przykład, jeśli jedna wartość jest mniejsza niż inna, użyj instrukcji JNL lub JNB dla przeskoczenia nad kodem. Oczywiście, jeśli testujesz warunek ujemny, wtedy użyj instrukcję w postaci JX dla przeskoczenia nad kodem.

8.0 Typy danych

Przed nadejściem MASM, większość asemblerów wspierało bardzo małe możliwości deklarowania i alokowania złożonych typów danych. Generalnie, można było alokować bajty, słowa i inne prymitywne struktury maszynowe. Można było również odłożyć blok bajtów. Przy możliwościach języków wysokopoziomowych deklarowania i używania abstrakcyjnych typów danych ,język asembler corz bardziej podupadał. Wtedy MASM nadszedł i wszystko zmienił. Niestety przez długi czas programiści asemblerowi nie chciało się nauczyć nowej składni MASM dla takich rzeczy jak tablice, struktury i inne typy danych. Podobnie, wielu nowych programistów asemblerowych nie przeszkadzało nauczenie się i używanie możliwości tych typów danych ponieważ byli już przyłoczeni językiem asemblera i chcieli zminimalizować liczbę rzeczy jakich musieli się uczyć. To rzeczywiście wstyd ponieważ typy danych MASM są jedną z największych poprawek w asemblerze ponieważ używa mnemonikó zamiast binarnych opkodów dla programowania na poziomie maszynowym. Zwróć uwagę ,że MASM jest asemblerem "wysokiego poziomu"Wprowadził spatrwdzanie typów operandów i raportowania błędów jeśli istnieją rozbieżności. Niektórzy ludzie, którzy używali asemblera na innych maszynach ,uważali to za denerwujace. Jednak jest to świetny pomysł w asemblerze z tego samego powodu jakim był w HLL'ach. Cechy te miały jeden korzystny efekt uboczny: pomagały innym zrozumieć co próbujesz zrobić w swoim programie.

8.1 Definiowane nowego typu danych przez TYPEDEF

MAMS wspiera małą liczbę prymitywnych typów danych. Dla typowej aplikacji, bajty, słowa, swordy, dwordy, sdwordy i różne formaty zmiennoprzecinkowe są najczęściej używanymi, dostępnymi skalarnymi typami danych. Możesz konstruować więcej abstrakcyjnych typów danych przez użycie tych wbudowanych typów. Na przykład, jeśli chcesz znaku, normalnie deklarujesz zmienną byte. Jeśli chcesz 16 bitową liczbę całkowitą, zazwyczaj użyjesz deklaracji sword (lub word). Oczywiście, kiedy napotkasz dekalrację zmiennej taką jak "answer byte ?" jest trochę trudno odgadnąć jaki to jest rzeczywisty typ. Czy mamy znak, wartość logiczną, małą liczbę całkowitą, lub coś innego? Ostatecznie nie ma znaczenia maszyna; bajt jest bajtem.Interpretacja jako znak, wartość logiczna lub wartość całkowita jest definiowana przez instrukcję mszynową która na niej działa, a nie przez sposób jej zdefiniowania. Niemniej jednak to rozróżnienie jest ważne dla kogoś kto czyta twój program. Dyrektywa typedef MASM'a może pomóc uczynić to rozróżnienie wyraźniejszym. W swojej najprostsej postaci, dyrektywa typedef zachowuje się jak textequ. Pozwala ci zastąpić jeden łańcuch w programie, innym. Na przykład możesz stworzyć poniższe definicje z MASM:

```
char        typedef byte
integer     typedef sword
boolean     typedef byte
float       typedef real4
IntPtr      typedef far ptr integer
```

Kiedy zadeklarowałeś te nazwy, możesz zdefiniować zmienne char, integer,boolean i float jak następuje:

```
MyChar          char ?
```

```
I                   integer ?
Ptr2I               IntPtr I
IsPresent           boolean ?
ProfitsThisYear     float ?
```

Reguła: Używaj istniejących typów danych MASM jako typ wbudowanych bloków. Dla większości typów danych jakie możesz stworzyć , powinieneś je deklarować wyraźnie nazwą typu używając dyrektywy typedef.

8.2 Tworzenie typów tablicowych

MAMS dostarcza interesujących cech dla rezerwowania bloków pamięci – operator DUP. Operator ten jest niezwykły (wśród języków asembler) ponieważ jest definiowany rekurencyjnie. Podstawowa definicja:

DupOperator = wyrażenie ws* 'DUP' ws* '(' ws* operand ws* ') %%

Zwróć uwagę ,że "wyrażenie" rozwija się w poprawną wartość numeryczną (lub wyrażenie numeryczne), "ws*" oznacza zero lub więcej białych znaków a "operand" rozszerza się w to co jest porawne w polu operandu MASM dyrektywy word/dw, byte/db itd. Typowym zastosowanie tego operatora do rezerwacji bloku pamięci :

```
ArrayName       integer  16  dup  (?)        ;Deklaracja tablicy 16 słów
```

Ta deklaracja ustawia 16 sąsiadujących słów w pamięci. Interesującą rzeczą o operatorze DUP jest to ,że każde poprawne pole operandu dla dyrektywy jak bajt lub słowo może pojawiać się wewnątrz nawiasów, w tym dodatkowe wyrażenia DUP. Operator DUP po prostu mówi "duplikuj ten obiekt określoną ilość razy". Na przykład "16 dup (1,2)" mówi "daj mi 16 kopii pary wartości jeden i dwa". Jeśli operand ten pojawił się w polu operandu jako dyrektywa byte, zarezerwuje 32 bajty, zawierające alternatywne wartości jeden i dwa. Więc co się zdarzy jeśli zastosujemy tą technikę rekurencyjnie? Cóż "4 dup (3 dup (0))" kiedy odczytujemy rekurencyjnie "daj mi cztery kopie tego co jest w wewnątrz (zewnętrznych) nawiasów". To okazuje się być wyrażeniem "3 dup (0)" , które mówi "daj mi trzy zera" Ponieważ oryginalny operand mówi aby dać cztery kopie trzech kopii zera, wynikiem jest to wyrażenie tworzące 12 zer. Teraz rozważmy dwie deklaracje:

```
Array1 integer 4 dup ( 3 dup (0))
Array2 integer 12 dup (0)
```

Obie definicje ustawiają 12 liczb całkowitych w pamięci (każda inicjowana zerem) . Dla asemblera są one niemal identyczne; dla 80x86 są one absolutnie identyczne. Dal czytelnika jednak, są one oczywiście różne. Dla zadeklarowania dwóch identycznych jednowymiarowych, użyjemy dwóch różnych dekalracji tworzące program niespójnym i trudnym do odczytania. Jednak , możemy wykorzystać te różnice dla zadeklarowania wielowymiarowych tablic. Pierwszy przykład powyżej, sugeruje ,że mamy cztey kopie tablicy zawierające trzy liczby całkowite każda. Odpowiada to popularnej funkcji dostępu do tablicy z szeregowym układem. Drugi przykład sugeruje ,że mamy jednowymiarową tablicę zawierającą 12 liczb całkowitych

Wskazówka: Weź pod uwagę rekurencyjną naturę operatora DUP dla zadeklarowania wielowymiarowych tablic w programie.

8.3 Deklarowanie struktur w asemblerze

MASM dostarcza doskonałej cechy dla deklarowania i używania struktur, unii i rekordów; z pewnych powodów, wielu programistów asemblerowych ignoruje je i ręcznie oblicza offset do pól

wewnątrz struktury w swoim kodzie. Nie tylko czyni to robi to procedurę trudną do odczytaniA.

Reguła: Kiedy typ danej struktury jest właściwy w programie asmeblerowym,zadeklaruj odpowiednią strukturę w programie i użyj jej. Nie wyliczaj offsetu pola w strukturze ręcznie, użyj standardowej struktury "notacja kropki" dla dostępu do pól tej struktury

Jedne problem z użyciem struktur wystąpi kiedy masz dostęp do pól struktury pośrednio (tj przez wskaźnik). Pośredni dostęp zawsze występuje przez rejestr (dla bliskiego wskaźnika) lub pary segment / rejestr (dla dalekiego wskaźnika) Kiedy ładujesz wartość wskaźnika do rejestru lub pary rejestrów, program nie wskaże prosto jakiego wskaźnika użyłeś. Jest to prawda jeśli używasz pośredniego dostępu kilka razy w sekcji kodu bez przeładowania rejestru(-ów). Spójrzmy na ten kod:

```
s                   struct
a                   Integer ?
b                   integer ?
s                   ends
                    .
                    .
                    .
r                   s               {}
ptr2r       dword   r
                    .
                    .
                    .
                    les             di, ptr2r
                    mov             ax, es:[di].s.a ;Brak wskazanie ,że to ptr2r!
                    .
                    .
                    .
                    mov             es:[di].b, bx   ;Rzeczywiście brak wskazania!
```

A teraz spójrzmy na ten:

```
s                   struct
a                   Integer ?
b                   integer ?
s                   ends
sPtr                typedef far ptr s
                    .
                    .
                    .
q                   s               {}
r                   sPtr            q
r@                  textequ         <es:[di].s>
                    .
                    .
                    .
                    les             di, ptr2r
                    mov             ax, r@.a                ;Teraz jest jasne użycie r
                    .
```

```
.
.
mov             r@.b, bx                ;Kopia
```

Zwróć uwagę ,że symbol "@" jest poprawnym identyfikatorem znaku w MASM, zatem "r@" jest innym symbolem . Jako ogólną zasadę powinieneś przyjąć unikanie symboli takich jak "@" jako identyfikatorów, ale służy jako dobry cel – wskzuje ,że mamy pośredni wskaźnik. Oczywiście, musisz zawsze upewnić się ,że załadowałeś wskaźnik do ES:DI kiedy używasz powyższego textequ. Jeśli korzystasz z kilku różnych par segment / rejestr dla dostępu do danych które wskazuje "r" ta sztuczka może nie uczynić kodu bardziej czytelnym ponieważ będziesz potrzebował kliku textequ , które wszystkie oznaczają tą samą rzecz.

8.4 Typy danych a Biblioteka Standardowa UCR

Biblioteka Standardowa UCR dla Programistów Język Asemblera 80x86 dostarcza zbioru makr, które pozwalają ci zadeklarować tablice i wskaźniki używając składi C. Poniższe przykłady demonstrują te możliwości.

```
var
                integer         i, j, array[10], array2[10][3], *ptr2Int
                char            *FirstName, LastName[32]
endvar
```

Ta deklaracja daje poniższy kod asemblerowy:

```
i               integer ?
J               integer 25
array           integer 10 dup (?)
array2          integer 10 dup ( 3 dup (?))
ptr2Int         dword ?
LastName        char 32 dup (?)
Name            dword LastName
```