
# Apostila de Cobol Básico

mundo COBOL

http://www.mundocobol.com

DTS Latin America Software

www.dtslatin.com

MICRO FOCUS

*Índice*

***Introdução***

Esta apostila tem como objetivo treinar pessoas com algum conhecimento em lógica de programação. Visualizando aspectos básicos da linguagem, o suficiente para que a pessoa consiga desenvolver programas básicos utilizando a linguagem COBOL (COmmon Bussines Oriented Language). Esta é uma linguagem voltada exclusivamente para ambientes comerciais, ela possui uma arquitetura de programação muito fácil e clara, tornando os programas auto-documentáveis. Recentemente, a Merant Micro Focus com o produto Net Express empregou uma série de novas tecnologias à linguagem. Dentre elas: Programação orientada a objetos, suporte ao desenvolvimento de aplicações gráficas GUI (Graphical User Interface ou simplesmente aplicações Windows), suporte ao desenvolvimento de aplicações CGI (Common Gateway Interface ou simplesmente aplicações voltadas para os ambientes de Internet/Intranet/Extranet), acesso via ODBC (Open Data Base Connectivity) que permite a um programa escrito em COBOL acessar praticamente qualquer tipo de base de dados ou banco de dados existente atualmente no mercado. Outra característica muito importante é a portabilidade do código gerado pelo ambiente Net Express chamado de .INT, este arquivo é interpretado em tempo de execução pelo Run-time instalado no ambiente, desta forma podemos escrever nossas aplicações no ambiente Windows e utilizar todos os poderosos recursos de depurações fornecidas pela ferramenta e executar no ambiente Linux, por exemplo. Existem muitas plataformas suportadas pela Merant Micro Focus. Esta apostila não aborda nenhuma destas novas tecnologias mencionadas, apenas cláusulas de um fonte COBOL.
Para executar os exercícios e compilar os programas exemplos dados nesta apostila é necessário que você tenha um compilador Cobol na sua máquina. O ideal seria que você tivesse o Netexpress(COBOL Microfocus), pois com ele você poderia testar a sua aplicação. Comprando qualquer produto pelo site Mundocobol.com na **Cobol Store** você ganha um CD demo do NetExpress e pode trabalhar com ele por 30 dias, gerando executáveis, etc.
Caso não possua o compilador Cobol utilize o Notepad para editar os programas exemplo e pelo menos visualizar a estrutura de um programa Cobol com toda a identação das seções.
Para qualquer informação adicional sobre Cobol, as evoluções e compiladores acesse www.mundocobol.com. O Objetivo deste apostila é com que **VOCÊ** conheça e aprenda mais sobre o COBOL esta poderosa linguagem de programação.

**Explanação de um fonte Cobol**

Todo programa escrito na linguagem Cobol possuía algumas regras rígidas a serem seguidas como uma redação que possui início, meio e fim, mas isto já é opcional, com o advento de novas tecnologias é possível escrever um código fonte em *free format*, ou seja, não é necessário seguir todas as regras da linguagem. O código fonte possui quatro divisões que devem ser utilizadas nesta ordem: IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION e PROCEDURE DIVISION.

Estas divisões devem ser escritas seguindo algumas regras de posicionamento:

```
|   |||  ||                                              |
....|....1....|....2....|....3....|....4....|....5....|....6....|....7....|...
```

| | |
|---|---|
| Colunas de 1 a 6: | Área de numeração sequencial |
| Coluna 7: | Área de indicação |
| Colunas de 8 a 11: | Área A |
| Colunas de 12 a 72: | Área B |

Pode-se digitar até a coluna 80 mas tudo que for digitado entre 73 e 80, será considerado como um comentário.

***Área de numeração sequencial***

Normalmente consiste em seis dígitos em ordem crescente que normalmente são utilizados para marcas de identificação do fonte. Segundo as regras no ANS85 pode-se também colocar comentários nesta área. Além disso podemos colocar um asterisco na coluna 1 (um) ou qualquer outro caractere onde sua representação ASCII seja menor do que 20 (ESPAÇO), fazendo com que a linha inteira seja considerada como um comentário.

***Área de indicação***

Um hífen (-) nesta posição indica que existe uma continuação de uma cadeia de caracteres que foi iniciada na linha anterior, ou seja, quando desejamos atribuir um valor inicial a uma variável em sua declaração ou quando desejamos exibir um texto muito extenso. Opcionalmente podemos colocar o texto inteiro na linha de baixo para não termos que *quebra-lo*.

Um asterisco (*) indica que toda a linha deve ser tratada como um comentário.

Uma barra (/) além de tratar a linha como comentário fará com que ao gerar um arquivo de listagem (.LST) será adicionado um salto de página para impressões.

Um dólar ($) indicará ao compilador que a linha terá diretivas de compilação ou compilação condicional. Estas diretivas são utilizadas para modificar o comportamento e compatibilidade dos programas no ambiente Merant Micro Focus.

### *Área A e B*

Originalmente havia uma separação do que poderia ser escrito na área A e na área B, com o advento do ANS85 isto não existe mais. Esta distinção implicava que as divisões, seções, declaração de variáveis e parágrafos do programa eram os únicos que podiam utilizar a área A e outros comandos a área B.

Vamos analisar o código fonte a seguir:

```
....|....1....|....2....|....3....|....4....|....5....|....6....|....7..
      identification division.
      program-id.    CalcArea.
      author.        DTS Latin America Software.
     *
      data division.
      working-storage section.
      77 largura                  pic 9(003) value zeros.
      77 altura                   pic 9(003) value zeros.
      77 area-result               pic 9(006) value zeros.
     *
      procedure division.
      inicio.
          display erase
          display "Calculo de area (Quadrados/Retangulos)" at 0521
          display "Largura: " at 1010
          display "Altura : " at 1210
          accept largura at 1019
          accept altura at 1219
          multiply largura by altura giving area-result
          display "Area   : " at 1410 area-result
          stop run
          .
```

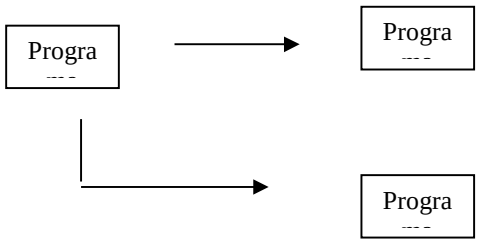
### *Identification Division*

Divisão de identificação do programa. Esta divisão possui a seguinte estrutura:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. nome-programa.
AUTHOR.              comentário.
INSTALLATION.        comentário.
DATE-WRITTEN.        comentário.
DATE-COMPILED.       comentário.
SECURITY.            comentário.
```

Todos os comandos que possuem a palavra comentário na frente, não possuem nenhum efeito na aplicação, são apenas parâmetros opcionais para documentação do programa, mas, caso queira utilizar algum deve ser escrito corretamente para que não cause erro de compilação. Deve-se optar em deixar o parâmetro nome-programa com o mesmo nome externo do fonte, pois, este parâmetro será utilizado para identificação em memória após a primeira carga, ou seja, quando chamamos um programa, o primeiro local de pesquisa feita pelo compilador será em memória e caso não encontre ele irá procurar no disco e carregá-lo

na memória, após isto, toda vez em que o programa for chamado de novo não será necessário a pesquisa em disco, pois, o programa já estará na memória (caso não tenha sido cancelado), agora vejamos o seguinte exemplo:




O Programa Chamador (Principal.cbl) se trata de um programa Batch que irá executar um processo isolado de tratamento dos dados e ele irá executar o Programa Chamado 1 (Prog1.cbl), quando o Prog1 voltar o controle ao Principal este chamará o Programa Chamado 2 (Prog2.cbl) e este Prog2 foi criado a partir de uma cópia de Prog1 e o programador esqueceu de modificar o seu Program-ID deixando-o com Prog1, bem, a confusão irá começar quando o Principal for chamar o Prog1 novamente, como o Prog2 possui a mesma referência de identificação de Prog1 o compilador irá executar o Prog2 imaginando que se trata do Prog1.

### *Data Division*

Divisão voltada única e exclusivamente à definição de estruturas de registros, variáveis e constantes do programa, ou seja, uma área de alocação de memória para todo o espaço necessário ao seu programa.

Esta divisão possui a Working-Storage Section. Esta seção da Data Division é voltada para a declaração das variáveis e constantes do programa.

Todos os nomes utilizados no programa devem ser exclusivos, existem meios de quebrar esta regra conforme visto posteriormente, e de no máximo trinta caracteres, caso o nome de algum elemento ultrapasse estes trinta caracteres simplesmente terá seu excedente ignorado pelo compilador.

```
77 largura                pic 9(003) value zeros.
```

Esta linha está declarando uma variável chamada largura de 3 posições numéricas e lhe atribui como valor inicial: zero. O número 77 à frente indica o nível da variável. Isto será discutido mais à frente.

### *Procedure Division*

Esta divisão controla a execução do programa, é onde colocamos os comandos a serem executados em uma ordem lógica. Esta execução é controlada por parágrafos (existe um chamado INICIO no programa exemplo) eles funcionam como identificações de blocos de comandos.

display erase

O comando *display* é utilizado para exibir informações na tela em ambientes caracteres. A palavra reservada *erase* é utilizada em conjunto com display para limpar a tela.

display "Calculo de area (Quadrados/Retangulos)" at 0521

Esta forma de utilização do display irá exibir na tela a cadeia de caracteres entre as aspas (" ") na posição especificada por *at 0521*, ou seja, linha 5 e coluna 21 que são compreendidas entre linhas de 1 a 25 e colunas de 1 a 80.

accept largura at 1019

Este comando é utilizado para *aceitarmos* alguma informação, neste caso estaremos esperando que usuário informe algo na posição 1019 (Seguem as mesmas regras do comando display), o usuário indica para a aplicação que terminou de fornecer estar informações pressionando a tecla ENTER.

multiply largura by altura giving area-result

O comando *multiply* é um dos comandos aritméticos da linguagem, ele é utilizado para funções de multiplicação, neste caso ele irá multiplicar o conteúdo numérico da variável *largura* por *altura* movendo o resultado para a variável *área-result*.

stop run

Este comando encerra a aplicação, na verdade o *run* é um parâmetro do comando *stop* onde podemos substituir o *run* por uma cadeia de caracteres (string). Por exemplo: stop "Teste…". Este comando irá exibir a mensagem e esperar por um ENTER para que seja finalizado.

Todo parágrafo deve possuir um *ponto final* ou o último comando deve possuir um ponto. Em versões anteriores da linguagem o ponto era obrigatório para cada comando.

Exercícios:

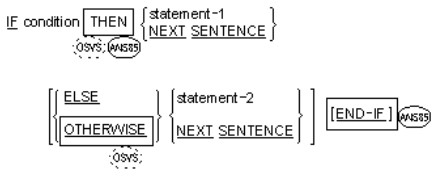
1º - Digite o programa ilustrado em um Exerc1.cbl e execute-o.

2º - Altere o Exerc1.cbl fazendo com que todos os display's e accept's fiquem duas linhas mais abaixo na sua apresentação e execute-o.

***Comandos condicionais***

Os programas sempre necessitam tomar decisões sobre que rumo tomar em certas circunstâncias. Para que o programa possa fazer isto existem dois *comandos condicionais*: o *IF* e o *EVALUATE*:

A estrutura do *IF* é:




O comando *IF* é utilizado para tomadas de decisões simples do tipo *Se for verdade faça isto Senão faça aquilo*, *condition* indica a condição a ser avaliada e se for verdade o comando *statement-1* será executado, caso a avaliação da condição não seja verdadeira o *IF* executará o *statement-2* caso o mesmo exista, caso contrário simplesmente o compilador irá executar o próximo comando. Por exemplo:

```
...
if nome equal "FABIO"
  display nome
else
  display "Desconhecido..."
end-if
...
```

Neste exemplo, caso a variável *nome seja igual a "FABIO"* então o comando *display nome* será executado, caso contrário será executado o comando *display "Desconhecido..."*. Note que neste exemplo não foi utilizada a cláusula *THEN* pois a mesma é opcional. Outro detalhe importante é referente ao ponto, caso o programador opte em adotar um ponto por comando, ele deve se atentar no seguinte:

```
*> Isto esta errado!!!
...
if nome equal "FABIO".
  display nome.
else.
  display "Desconhecido...".
end-if.
...

*> Isto esta certo!!!
...
if nome equal "FABIO"
  display nome
else
  display "Desconhecido..."
end-if.
...
```

O comando *IF* irá considerar como o fim de seus comandos a cláusula *END-IF* ou o primeiro ponto encontrado.

Para as condições válidas temos:

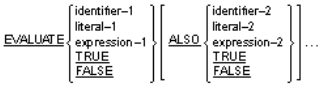
| ***Condição*** | ***Símbolo*** | ***Cláusula*** |
|---|---|---|
| Igual | = | *EQUAL* |
| Menor que | < | *LESS THAN* |

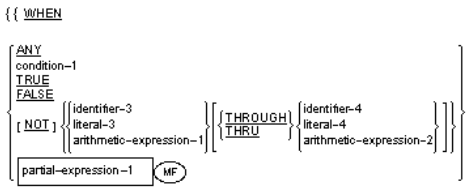
| | | |
|---|---|---|
| Menor igual que | <= | *LESS THAN OR EQUAL TO* |
| Maior que | > | *GREATER THAN* |
| Maior igual que | >= | *GREATER THAN OR EQUAL TO* |
| Diferente | <> | *NOT EQUAL* ou *UNEQUAL TO* |

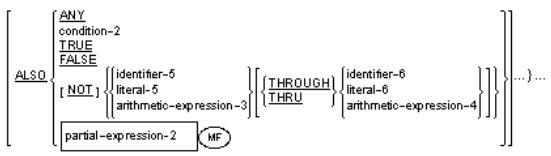
Para condições de NÃO, ou seja, *não é igual*, *não é diferente*, basta preceder com a cláusula NOT. Todos os exemplos a seguir são válidos:

```
...
if nome NOT EQUAL "FABIO"
  display nome
end-if
...

...
if nome UNEQUAL "FABIO"
  display nome
end-if
...

...
if nome EQUAL "FABIO" AND estado EQUAL "SP"
  display nome
end-if
...

...
if nome EQUAL "FABIO" OR estado EQUAL "SP"
  display nome
end-if
...

...
if (nome EQUAL "FABIO" AND estado EQUAL "SP") OR idade > 20
  display nome
end-if
...
```

A estrutura do *EVALUATE* é:




imperative statement-1} ...

[WHEN OTHER imperative-statement-2]

[END-EVALUATE]

A cláusula *Evaluate* possui um sofisticado mecanismo de *análises condicionais*, onde podemos utilizar as mesmas condições aceitas pela cláusula *IF* e além disso algumas cláusulas próprias do comando. *ALSO* indica que EVALUATE deverá analisar duas condições distintas simultaneamente. Podemos também utilizar a instrução *TRUE* e *FALSE*, que indicarão que a análise deverá ser feita nas entradas *WHEN*. *WHEN OTHER* é uma alternativa caso nenhuma das condições de *WHEN* gere um *TRUE* na avaliação. Todos os exemplos a seguir são válidos:

```
...
Evaluate true
When nome equal "FABIO"
  Display "Teste 1"
When nome equal "LEANDRO"
  Display "Teste 2"
When nome equal "LEONARDO"
  Display "Teste 3"
When other
  Display "Erro..."
End-evaluate
...
```

```
...
Evaluate nome
When "FABIO"
  Display "Teste 1"
When "LEANDRO"
  Display "Teste 2"
When "LEONARDO"
  Display "Teste 3"
When other
  Display "Erro..."
End-evaluate
...
```

```
...
Evaluate numero
When 1
  Display "Teste 1"
When 2 thru 5
  Display "Teste 2"
When 6
  Display "Teste 3"
When other
  Display "Erro..."
End-evaluate
...
```

```
...
Evaluate numero also nome
When 1 also "FABIO"
  Display "Teste 1"
```

```
When 2 thru 5 also "LEANDRO"
  Display "Teste 2"
When 6 also "LEONARDO"
  Display "Teste 3"
When other
  Display "Erro..."
End-evaluate
...
```

Exercícios:

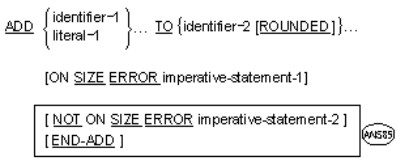
1º - Copie o Exerc1.cbl para um Exerc2.cbl e faça uma condição para quando área-result for menor do que 100 exiba "Area (peq)" e caso contrário "Area (gnd)".

### *Cálculos aritméticos*

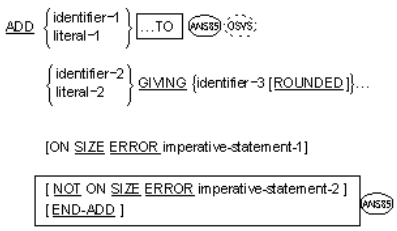
A linguagem Cobol possui rotinas que auxiliam o programador a efetuar cálculos aritméticos, são eles:

*ADD*: Utilizado para somar valores, sua sintaxe é:

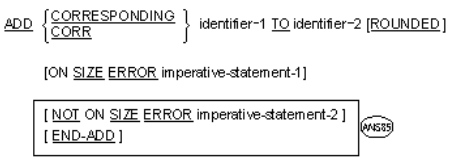
Formato 1:




Formato 2:



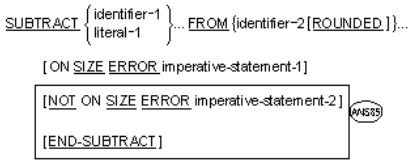

Formato 3:



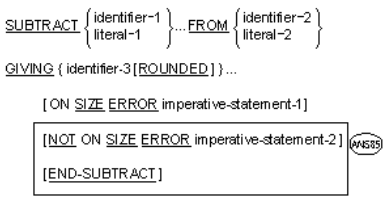

*Identifier-1, Identifier-2* e *Identifier-3* representam variáveis numéricas. *Literal-1* e *Literal-2* representam valores fixos, ou seja, o número escrito no programa. *Rounded* indica ao comando que os valores decimais devem ser arredondados, caso esta palavra reservada não seja indicada o compilador irá *truncar* estes valores, ou seja, um valor 10.25 truncado ficará como 10.2. *GIVING* indica que utilizaremos uma variável como repositório do resultado obtido pela soma de *Identifier-1* com *Identifier-2*, caso *GIVING* não seja especificado o *Identifier-2* será afetado pelo comando, ou seja, ele será utilizado para o resultado da soma. *Imperative-statement-1* e *Imperative-statement-2* indicam que algum comando deverá ser executado, desde que este comando não seja de análise condicional, por exemplo, um IF. ON SIZE ERROR ocorrerá sempre que o *Indentifier-2* ou *Indentifier-3* não comportarem o valor especificado e NOT ON SIZE ERROR ocorrerá na situação de normalidade do processo. *CORRESPONDING* é utilizado para somar uma TABELA inteira, ou seja, podemos construir tabelas nas definições das variáveis e com este recurso do comando somar a tabela de uma única vez, sem a necessidade da construção de laços para somar item a item. *END-ADD* é opcional. *ANS85* indica que está é uma convenção nova e *OSVS* é uma convenção antiga utilizada no *MainFrame*.

*SUBTRACT*: Utilizado para subtrair valores, sua sintaxe é:

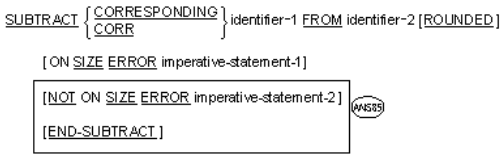
Formato 1:




Formato 2:



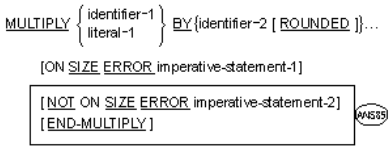

Formato 3:



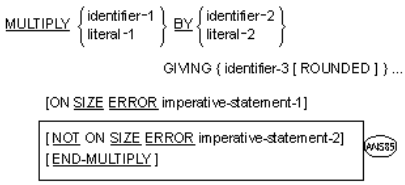

Segue as mesmas regras do comando ADD.

*MULTIPLY*: Utilizado para multiplicar valores, sua sintaxe é:

Formato 1:



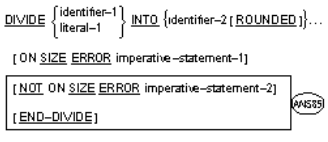

Formato 2:



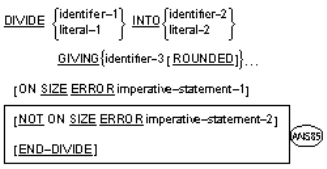

Segue as mesmas regras do comando ADD.

*DIVIDE*: Utilizado para dividir valores, sua sintaxe é:

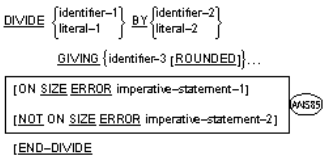
Formato 1:



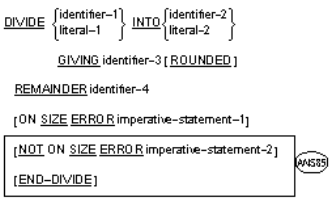

Formato 2:



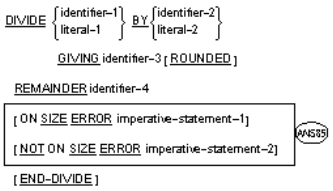

Formato 3:




Formato 4:



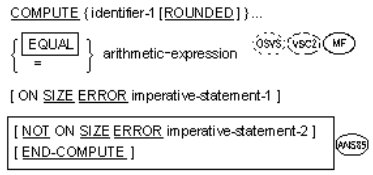

Formato 5:




Quando utiliza-se a palavra reservada *BY* estamos informando que se trata de uma divisão do *Identifier-1* pelo *Identifier-2.* Utilizando a palavra reservada *INTO* invertemos a situação. *REMAINDER* indica que o comando *DIVIDE* não deverá truncar o resultado e mover o resto para o *Idenfifier-4*, desde que o *Idenfifier-2* (caso GIVING não seja especificado) ou *Identifier-3* não tenham casas decimais.

*COMPUTE*: Utilizado para calcular valores, sua sintaxe é:

Formato 1:




Podemos utilizar qualquer expressão aritimética válida (excluindo-se chaves "{}" e colchetes "[]") para que o *COMPUTE* a execute. Os sinais esperados são:

| | |
|---|---|
| + | Adicionar |
| - | Subtrair |
| * | Multiplicar |
| / | Dividir |
| ** | Exponenciação |

Esta expressão também poderá conter "Parênteses" ( ).

Por exemplo:

COMPUTE resultado = 2 + 3 * 4 – ( 3 + 4 )

***Laços***

Existem basicamente dois tipos de programas, os *on-line* e *batch*, os programas de denominação *on-line* são aqueles que funcionam com interação direta e exclusiva de um usuário controlando-o. Os programas denominados *batch* são programas que podem ser chamados a partir destes programas *on-line* (muitos programas *on-line* não utilizam programas *batch*) mas em sua grande maioria são programas de uso restrito de usuários de áreas responsáveis pelo controle e qualidade das informações (CPD), onde estes programas são iniciados e podem não solicitar nada para executar seus trabalhos e simplesmente informar apenas o término do processo ou solicitar apenas algumas informações essenciais para fazer o trabalho. Estes programas inevitavelmente terão que executar um determinado grupo de comandos e condições para cada registro de uma base de informações (base de dados nativa ou bancos de dados), isto é, controlados por *laços*, são dispositivos fornecidos pela linguagem para que possamos executar os mesmos comandos com dados diferentes até que uma determinada condição seja satisfeita, por exemplo, o fim dos registros a serem tratados.

Existem dois comandos utilizados para controle de *laços* o *PERFORM* e o *GO TO*, o comando *perform* possui mecanismos mais apurados para tratamento de laços e o *go to* é muito mais simples e também é conhecido como o "*patinho feio"* entre os programadores da linguagem *Cobol* por possibilitar fazer o desvio da execução de parágrafo tornando a manutenção de códigos, muito mais complexa e trabalhosa.

A diferença mais destacada entre estes dois comandos é que quando o *PERFORM* executa um parágrafo ele retorna para executar o próximo comando e o *GO TO* não faz isto, ele simplesmente desvia a execução.

Exemplo de um PERFORM:

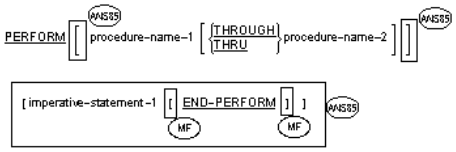
```
*> se o valor de numero for igual a 1 executa o paragrafo-1
*> antes de exibir "Processo ok..."
If numero not equal 1
  Perform paragrafo-1
End-if
display "Processo ok..."
...
```

Exemplo de um GO TO:

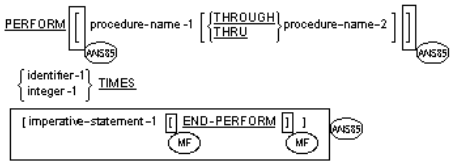
```
*> se o valor de numero for igual a 1 nao exibe
*> "Processo ok..."
If numero not equal 1
  Go to paragrafo-1
End-if
display "Processo ok..."
...
```

A seguir vamos ver a estrutura do comando *Perform*:

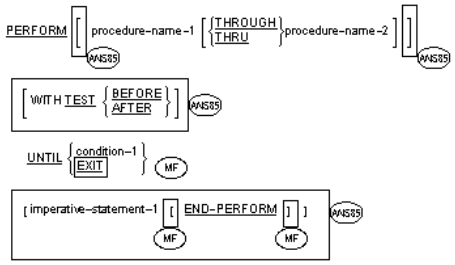
Formato 1:



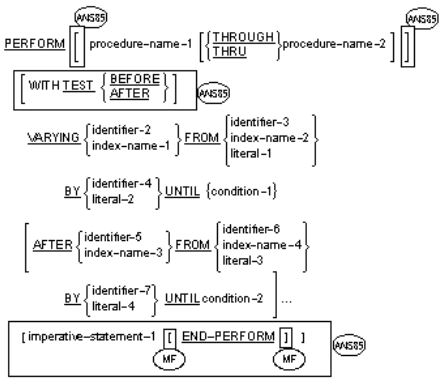

Formato 2:




Formato 3:



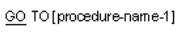

Formato 4:




*THROUGH* e *THRU* são equivalentes, *procedure-name-1* e *procedure-name-2* são nomes de parágrafos a serem executados. Caso opte-se em utilizar um *imperative-statement-1*, ou seja, um comando a ser executado várias vezes não pode-se utilizar o nome de parágrafos em conjunto. *END-PERFORM* indica o término dos comandos a serem executados dentro do laço *in-line*, logo quando se utiliza algum parágrafo não é permitido a utilização de *END-PERFORM*. *Identifier-1* ou *Integer-1* em conjunto com *TIMES* indica que o laço deve ser executado um determinado número de vezes. *WITH TEST BEFORE* (padrão) e *WITH TEST AFTER* informam ao laço se o testes condicionais devem ser feitos antes ou depois de uma nova execução. *UNTIL* determina uma condição de término do laço, esta condição respeitará a determinação de *WITH TEST AFTER*, se for especificado EXIT ao invés de uma condição para UNTIL quando estiver executando um parágrafo ele só poderá ser encerrado com um *STOP RUN* ou *EXIT PROGRAM* (este último comando tem a mesma função de *STOP RUN* para um programa secundário, ou seja, executado por outro programa), quando estamos executando um laço *in-line* ele só será encerrado após a execução de um *EXIT PERFORM* ou *GO TO*. *VARYING* fará com que o laço incremente uma variável numérica identificada por *Identifier-2* a partir de *Identifier-3* com um incremento especificado em *Identifier-4* até que uma condição seja verdadeira, condição essa especificada em *condition-1*, esta condição não precisa ter necessariamente a mesma variável envolvida em *VARYING*. AFTER possui as mesmas características técnicas de *VARYING* com seu funcionamento acionado para cada valor diferenciado na variável especificada em *VARYING*.

A seguir, vamos ver a estrutura do comando *Go To*:

Formato 1:




Formato 2:




Este comando faz um desvio para outro parágrafo do programa não dando continuidade aos comandos que existirem abaixo dele no parágrafo onde foi executado. Caso seja utilizado a cláusula *DEPENDING ON* a variável numérica identificada por *identifier* terá um número de

correspondência de qual *parágrafo* será executado, ou seja, se for especificada uma lista com três nomes de parágrafos e a variável for igual a 3 então o terceiro parágrafo será executado.

Exemplo de um laço:

```
       identification division.
       program-id.        Laco.
       author.            DTS Latin America Software.
      *
       data division.
       working-storage section.
       01 DataSistema.
          10 AnoSistema              pic 9(004) value zeros.
          10 MesSistema              pic 9(002) value zeros.
          10 DiaSistema              pic 9(002) value zeros.
       77 Contador                   pic 9(003) value zeros.
      *
       procedure division.
       inicio.
          display erase
          perform pega-data-sistema
          perform exibe-tela
          perform varying contador from 1 by 1
                  until contador > 10
            display "Contador: " contador
          end-perform
          stop run
          .
      *
       pega-data-sistema.
          accept datasistema from date yyyymmdd
          .
      *
       exibe-tela.
          display "Programa batch..."
          display "Data de execucao: " diasistema "/" messistema "/"
                                  anosistema
          display " "
          .
```

Análise do fonte:

```
          perform pega-data-sistema
```

Este *perform* irá executar um parágrafo chamado pega-data-sistema e retornar para o próximo comando que é outro *perform* que executa um parágrafo chamado *exibe-tela*. É o tipo de *perform* mais simples.

```
          perform varying contador from 1 by 1
                  until contador > 10
            display "Contador: " contador
          end-perform
```

Este é um perform *in-line*, ou seja, todo o processo será feito sem a execução de um parágrafo, além de incrementar o valor de contador, que é uma variável da área de *Working-Storage* a partir de 1 em 1 até que (until) o seu valor seja maior do que 10. Durante esta execução serão feitos diversos *displays* exibindo o valor da variável, encontrando *End-Perform* indicará o fim dos comandos a serem executados neste laço.

***Variáveis***

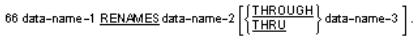
```
01 DataSistema.
   10 AnoSistema          pic 9(004) value zeros.
   10 MesSistema          pic 9(002) value zeros.
   10 DiaSistema          pic 9(002) value zeros.
```

Esta declaração irá criar uma variável numérica chamado DataSistema de 8 (oito) posições, mas para tornar a manipulação dos dados, mais fácil, nós a subdividimos em três variáveis distintas, desta forma podemos acessar facilmente diferentes posições da variável. Esta facilidade é denominada de Grupo, um grupo de itens (variáveis) pode conter subgrupos. Estes grupos possuem números de níveis que identificam seu nível dentro do grupo. Estes níveis variam de 1 a 49, quanto menor o número maior é o seu nível, por exemplo:

```
01 Endereco.
      10 Texto      pic x(030).
      10 Numero     pic 9(006).
```

Isto cria uma variável chamada *Endereço* que aloca 36 bytes de memória composta por duas referências, *Texto* que acessa os primeiros trinta bytes de *Endereço* que irão possuir um conteúdo alfanumérico (x) e *Numero* com seis bytes numéricos. Existem ainda os níveis especiais 66, 77, 78 e 88.

O nível 66 é utilizado para *renomear* outras variáveis do programa, este nível é utilizado da seguinte forma:



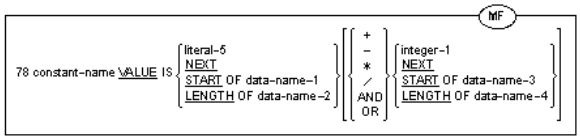

Isto é útil quando queremos por exemplo utilizar apenas o mês e o dia de uma data:

```
01 DataSistema.
   10 AnoSistema          pic 9(004) value zeros.
   10 MesSistema          pic 9(002) value zeros.
   10 DiaSistema          pic 9(002) value zeros.
66 MesDiaSistema renames MesSistema thru DiaSistema.
```

Desta forma acessando a variável *MesDiaSistema* estaremos enxergando o conteúdo das variáveis *MesSistema* e *DiaSistema*.

O nível 77 é utilizado para declarar variáveis que não irão possuir sub-itens, este nível é utilizado da mesma forma que o nível 01.

O nível 78 é utilizado para declarar constantes, ou seja, são referências que não podem ter o seu conteúdo alterado no programa. Sua estrutura é:




Nesta declaração é informado o nome da constante e seu valor. *Literal-5* representa uma *string* a ser associada. *NEXT* indica que esta variável terá a posição de *offset* referente ao segundo byte da declaração anterior, por exemplo:

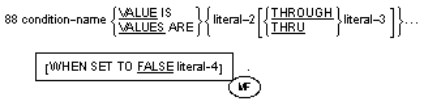
```
01 x1              pic x(10).
01 x2 redefines x1   pic x.
78 next-offset     value next.
01 x3              pic xx.
```

Neste exemplo, a constante next-offset terá em sua representação a referência do segundo byte da variável x1.

*START OF data-name-1* indica que a constante terá a posição de offset da variável informada em *data-name-1.*

*LENGTH OF data-name-2* indica que a constante terá o tamanho alocado pela variável informada em data-name-2.

O nível 88 é utilizado para representações de *VERDADEIRO* e *FALSO*, sua sintaxe:




Este nível é utilizado em itens de grupo, por exemplo:

```
01 FormaPagamento          pic 9(001) value zeros.
   88 Cartao               value 1.
   88 Dinheiro             value 2.
   88 Cheque               value 3.
```

Na situação inicial desta declaração *Cartão*, *Dinheiro* e *Cheque* estarão valendo *FALSE*, devido ao *value zeros* de *FormaPagamento*, quando queremos que o programa identifique qual a opção desejada pelo cliente basta mover 1 (por exemplo) para FormaPagamento, isto fará com que *Cartão* fique igual a *TRUE* e *Dinheiro* e *Cheque* fiquem igual a *FALSE* e no momento em que *FormaPagamento* for modificado para 3 a variável *Cheque* se tornará *TRUE* e *Cartão* e *Dinheiro* ficarão com *FALSE*. Pode-se utilizar também um comando do compilador chamado *SET*, por exemplo: *SET CARTÃO TO TRUE*. *THRU* indica que cada nível pode abranger uma faixa de valores para se tornar verdadeiro. Por exemplo:

```
01 FormaPagamento        pic 9(001) value zeros.
   88 Cartao             value 1 thru 2.
   88 Dinheiro           value 3 thru 4.
   88 Cheque             value 5 thru 6.
```

*WHEN SET TO FALSE literal-4* indica um valor opcional de falso para a condição. Na maioria dos casos este termo não é utilizado, sempre termos um dos itens selecionados ou *FormaPagamento setado* para zero.

A palavra reservada *PIC [PICTURE]* identifica o tipo e o tamanho da variável. Existem 6 classes de variáveis: Alfabética, Numérica, Alfanumérica e Ponto flutuante:

***Alfabética***

Representado pelo caractere "A", indica que esta variável suportará apenas caracteres alfabéticos ou espaço. Por exemplo:

```
01 nome          PIC A(010) value spaces.
```

Pode-se formatar a variável com o caractere "B", desta forma podemos formatar uma *string*:

```
01 nome          PIC A(005)BA(004) value spaces.
```

Se movermos "testenome" para a variável nome ela armazenará "teste nome".

***Numérica***

Representado pelo caractere "9", indica que esta variável suportará apenas caracteres de 0 a 9. As variáveis numéricas suportam no máximo 18 dígitos incluindo as casas decimais. Por exemplo:

```
77 numero        PIC 9(006) value zeros.
```

Podemos utilizar alguns mecanismos de formatação das variáveis para apresentação ou armazenamento, são eles:

O "P" é utilizado para economizar espaço em memória para o armazenamento de números onde cada "P" representa um *ZERO*, por exemplo:

```
77 numero        PIC 9(003)PPP value zeros.
```

Esta variável será capaz de armazenar até 999000, utilizando apenas três bytes da memória. Caso o programador coloque este "P" na frente da representação (PIC PPP9(003)) fará com que o compilador interprete os três primeiros caracteres como zeros. Este tipo de variável é para uso interno da aplicação.

O "S" é utilizado para informar ao compilador que a variável numérica poderá armazenar um sinal de *positivo* ou *negativo*. Este sinal não irá consumir espaço em memória.

```
77 numero        PIC S9(006) value zeros.
```

Desta forma a variável poderá armazenar a faixa de valores compreendentes entre –999999 e +999999. Pode-se utilizar também os sinais "-", "+", "CR" e "DB" (Estes dois últimos apresentarão dois espaços caso o valor seja positivo e o texto informado no caso de valores negativos) isto não irá restringir a variável a ter exclusivamente valores negativos ou positivos, seu comportamento será o mesmo do "S".

O "V" é utilizado para indicar onde se encontrará o ponto decimal da variável. Assim como o "S" ele não irá consumir espaço em memória.

```
77 numero          PIC S9(004)V99 value zeros.
```

Com esta declaração estaremos consumindo 6 bytes de memória com a possibilidade de guardar se este número é negativo ou positivo de quatro posições e com duas casas decimais. Por exemplo: -1234.56.

O "Z" é utilizado para formatar uma apresentação ao usuário, para cada "Z" à esquerda da variável indica que gostaríamos de ignorar a apresentação dos zeros e exibir espaços em seu lugar. Por exemplo:

```
77 numero          PIC SZ,ZZ9.99 value zeros.
```

Isto irá criar uma variável numérica onde os zeros à esquerda serão substituídos em uma apresentação em tela.

Outros caracteres válidos na representação numérica são "," e "." como visto anteriormente.

A "/" é utilizada para formação de campos de data, por exemplo:

```
77 data-nasc       PIC 99/99/9999 value zeros.
```

Mas isto não quer dizer que esta variável será tratada como uma data visto que compilador não possui uma variável específica para isto.

Um 0 (zero) indica uma separação, assim como vimos com o caractere "B" nos alfabéticos.

### *Alfanumérica*

Representado pelo caractere "X", indica que esta variável suportará todos os caracteres da tabela ASCII. Por exemplo:

```
01 nome            PIC X(010) value spaces.
```

Isto irá criar uma variável com 10 caracteres alfanuméricos, ou seja, qualquer caractere da tabela ASCII poderá ser utilizada nesta variável.

***Ocorrências***

Em conjunto com estas definições pode-se utilizar a declaração de uma *Tabela de Ocorrências* (conhecida também como *Vetor* ou *Array*), esta tabela simplifica a necessidade do programa ter que armazenar várias informações relacionadas, por exemplo, vamos imaginar que o programa tenha a necessidade de armazenar quais são os dias úteis do mês para determinar prováveis datas de pagamento, este programa pode possuir uma tabela com 31 ocorrências de campos numéricos de 1 byte que possuíram 0 e 1 para identificar se aquele dia é útil ou não. Veja o exemplo:

```
 identification division.
 program-id.      tabela.
*
 data division.
 working-storage section.
 77 indextab                pic 9(002) value zeros.
 77 tabmes                  pic 9(001) value zeros
                         occurs 31 times.
 01 datasistema.
    10 anosistema           pic 9(004).
    10 messistema           pic 9(002).
    10 diasistema           pic 9(002).
 77 inteirodata             pic 9(008) value zeros.
 77 diasmes                 pic 9(002) value zeros.
 77 diasemana               pic 9(001) value zeros.
 77 anobissexto             pic 9(004)v99 value zeros.
 77 anobissprova            pic 9(004) value zeros.
*
 procedure division.
 inicio.
    *> identifica o mes
    accept datasistema from date yyyymmdd
    *> Identifica quantos dias existe no mes
    evaluate messistema
    when 1
      move 31 to diasmes
    when 2
      move 28 to diasmes
      move zeros to anobissexto
      divide anosistema by 4 giving anobissexto
      move anobissexto to anobissprova
      subtract anobissprova from anobissexto
      if anobissexto not equal zeros
        move 29 to diasmes
      end-if
    when 3
      move 31 to diasmes
    when 4
      move 30 to diasmes
    when 5
      move 31 to diasmes
    when 6
      move 30 to diasmes
```

```
when 7
  move 31 to diasmes
when 8
  move 31 to diasmes
when 9
  move 30 to diasmes
when 10
  move 31 to diasmes
when 11
  move 30 to diasmes
when 12
  move 31 to diasmes
end-evaluate
*> Laco para cada dia do mes
perform varying indextab from 1 by 1
       until indextab > diasmes
  *> Dia a ser analisado
  move indextab to diasistema
  *> A funcao nao aceita um item de grupo
  move datasistema to inteirodata
  *> Identifica o dia da semana
  *> 1 = Segunda-feira
  *> 2 = Terca-feira
  *> 3 = Quarta-feira
  *> 4 = Quinta-feira
  *> 5 = Sexta-feira
  *> 6 = Sabado
  *> 7 = Domingo
  move function rem(function integer-of-date(inteirodata), 7)
      to diasemana
  if diasemana >= 1 and
     diasemana <= 5
    *> E um dia util
    move 1 to tabmes(indextab)
  end-if
end-perform
stop run
.
```

Exercícios:

1º - Faça um programa (Exerc3.cbl) que aceite um número e incremente-o de 2 em 2 exibindo os resultados na tela até que este número seja maior do que 50. Para este laço não utilize parágrafos.

2º - Repita o exercício anterior (Exerc4.cbl) utilizando parágrafos.

### *Exibindo e aceitando informações do usuário*

Já vimos os comandos DISPLAY e ACCEPT, que são utilizados para intereção com o usuário. Bem, existe uma seção da *Data Division* chamada *Screen Section*. Nesta seção são

especificadas às propriedades de telas a serem utilizadas pelo programa para interagir com o usuário. A seguir vemos um exemplo:

```
       identification division.
       program-id.       Tela.
      *
       data division.
       working-storage section.
       77 wrk-opcao                  pic x(001) value spaces.
       screen section.
       01 tela-principal.
          02 blank screen.
          02 line 1 col 1 value "----------------------------------------
      -                 "----------------------------------------".
          02 line 2 col 21 value "Manutencao de Clientes - Menu Principa
      -                 "l".
          02 line 3 col 1 value "----------------------------------------
      -                 "----------------------------------------".
          02 line 11 col 34 value "1 - Inclusao".
          02 line 12 col 34 value "2 - Alteracao".
          02 line 13 col 34 value "3 - Consultar".
          02 line 14 col 34 value "4 - Excluir".
          02 line 15 col 34 value "5 - Sair".
          02 line 17 col 34 value "Opcao: ( )".
          02 opcao line 17 col 42 pic x using wrk-opcao auto.
      *
       procedure division.
       inicio.
          display tela-principal
          accept tela-principal
          ...
          ...
```

Este programa irá resultar na seguinte apresentação:

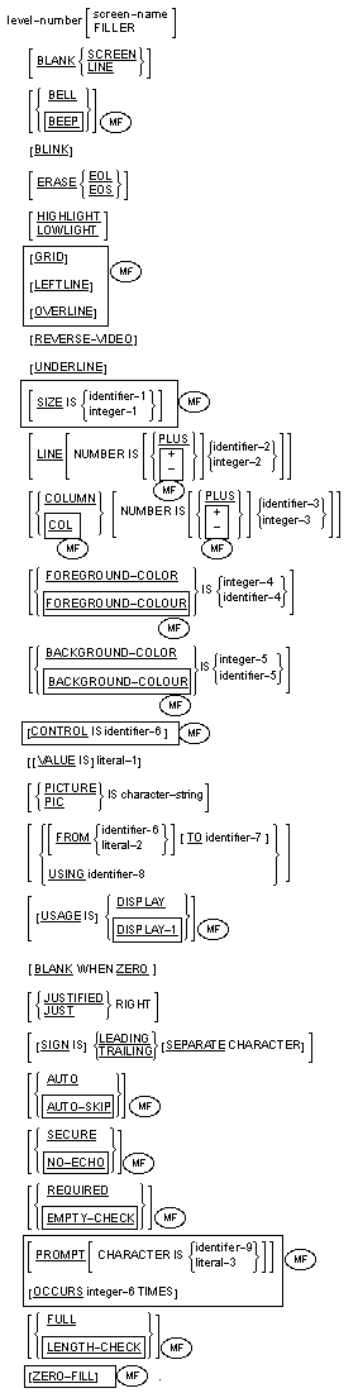
```
--------------------------------------------------------------------------------
                  Manutencao de Clientes - Menu Principal
--------------------------------------------------------------------------------



                          1 - Inclusao
                          2 - Alteracao
                          3 - Consultar
                          4 - Excluir
                          5 - Sair

                          Opcao: (_)
```

Note que além de exibir a tela o programa fez um *ACCEPT*. Se verificarmos o código fonte existe a seguinte linha:

```
02 opcao line 17 col 42 pic x using wrk-opcao auto.
```

Isto fará com que o programa transfira o que for digitado em *opcao* para *wrk-opcao*, desta forma o programa poderá identificar o que foi digitado pelo usuário.

A *descrição* de uma tela na Screen Section possui a mesma estrutura de um item de grupo, onde se especifica linha a linha o que vai ser exibido e o que vai ser solicitado ao usuário. A Screen Section é um recurso opcional para montagem de telas, o programa pode tranqüilamente montar suas apresentações de telas utilizando comandos de *ACCEPT* e *DISPLAY* logo a utilização ou não desta seção fica ao critério do programador. Uma coisa que o programador deve se atentar é que esta estrutura de tela estará consumindo memória. A seguir, vemos a estrutura completa dos itens de uma Screen Section:




Um ponto interessante a ser comentado é a utilização de cores através das cláusulas *FOREGROUND-COLOR* e *BACKGROUND-COLOR*, através destas cláusulas o programa pode especificar quais cores deseja utilizar, seguindo a tabela a seguir:

| *Código* | *Cor* |
|---|---|
| 0 | Preto |
| 1 | Azul |

| | |
|---|---|
| 2 | Verde |
| 3 | Ciano |
| 4 | Vermelho |
| 5 | Magenta |
| 6 | Marrom ou amarelo |
| 7 | Branco |

Adicionando-se a cláusula HIGHLIGHT a estas cores temos uma combinação de 15 possíveis cores, por exemplo, o marrom vira amarelo.

*No-echo* é uma cláusula para campos de senhas, onde o que esta sendo digitado não pode ser visto na tela.

Boa parte destas cláusulas da *Screen Section* pode ser utilizada em conjunto com o comando *DISPLAY* e *ACCEPT*.

Exercícios:

1º - Faça um programa (Exerc5.cbl) similar ao utilizado no exemplo.

### *Comunicação entre programas*

Não é interessante para um programador ter todas as rotinas em um único programa, o primeiro bom motivo para que ele não faça isso é a manutenção, imagine um parágrafo que faça um determinado cálculo o qual foi duplicado em 200 programas (.CBL). Agora imagine o transtorno no caso de uma pequena alteração deste parágrafo e a possibilidade de algum programa ser esquecido? Existe uma solução paliativa para este caso, o comando *COPY*, em sua forma mais simples, temos:

```
COPY teste.
```

Onde *teste* é o nome de um arquivo *teste.cpy* no diretório corrente ou em algum diretório que esteja informado em um *set* de ambiente chamado *COBCPY* (pertencente ao ambiente Merant). Pode informar também este um nome de arquivo entre aspas, podendo-se adicionar o caminho inteiro do arquivo.

Mas mesmo utilizando o *COPY* ainda persiste um problema, as variáveis que este parágrafo utiliza no cálculo, elas também devem ser replicadas em todos os programas, o *COPY* resolveria isto com a mesma facilidade, mas existe um problema mais sério ainda, a alocação de memória indevida, sem contar o perigo do esquecimento da atualização de alguma variável envolvida no cálculo e o valor anterior afetar o resultado.

Vistos todos estes problemas é muito mais prático o uso de *sub-programas*, desta forma centralizamos as rotinas comuns a mais de um programa e facilitamos a manutenção de determinados processos.

O comando utilizado para chamar sub-programas é o *CALL*, este comando transfere o fluxo de execução para outro programa e aguarda o seu retorno para dar continuidade nos comandos após a sua utilização. Por exemplo:

```
...
call "Chamado" using Nome
```

```
cancel "Chamado"
...
```

Este bloco de comando irá executar um programa *Chamado* passando como parâmetro uma variável do programa, chamada *Nome*, este programa *Chamado* irá possuir uma seção em sua *Data Division* chamada *Linkage Section*. Esta seção possuirá a declaração de uma variável para receber o conteúdo existente em *Nome* no momento da chamada. Esta variável de *Linkage* não precisa ter o mesmo nome da variável passada para o programa, apenas o mesmo tipo e tamanho para que receba o valor corretamente.

Este seria um programa *Chamado*:

```
identification division.
program-id.       Chamado.
author.           DTS Latin America Software.
*
data division.
linkage section.
77 Nome                    pic x(040) value zeros.
*
procedure division using Nome.
inicio.
   display Nome
   exit program
   .
```

Note que a Procedure Division recebeu um USING NOME. Isto fará com que o seu programa tenha acesso aos dados enviados pelo programa *Chamador*, podemos atualizar estes dados fazendo com que o programa *Chamador* tenha consiga ver os dados alterados. Note também que este programa utilizou um *EXIT PROGRAM* ao invés de um *STOP RUN*, este comando deve ser utilizado por programas que irão retornar o fluxo de execução para algum *Chamador*.

Neste caso vemos outra grande importância do *PROGRAM-ID*, pois, para que possamos retirar um programa da memória e liberar todo o espaço que estava alocado precisamos fazer um *CANCEL program-name*, este nome de programa referência-se ao *PROGRAM-ID* do programa a ser cancelado e retirado da memória.

Exercícios:

1º - Repita os procedimentos do Exerc3.cbl em um Exerc4.cbl empregando os novos comandos vistos. (O programa chamado deverá se chamar Exerc5.cbl).

### *Tratamento e armazenamento de informações*

O principal fundamento de um programa é a capacidade de saber tratar e armazenar informações devidamente validadas e consistentes. Existem quatro tipos de estruturas de armazenamentos de dados que o Cobol utiliza de forma nativa: Sequential, Line Sequential, Relative e Indexed. Todas estas estruturas estão diretamente relacionadas a arquivos armazenados em discos locais ou na rede. De forma nativa também o compilador possui controles próprios para tratamento em rede. Todo arquivo é composto por registros e cada

registro é composto por campos. Estes registros possuem a mesma estrutura de um *Grupo* da área de *Working*, onde cada registro é composto por vários campos de diferentes tipos.

### *Sequential*

A expressão seqüencial significa que os registros serão armazenados um após o outro, ou seja, assim que termina um registro outro terá início. Por exemplo:

Definição do registro:

```
01 Registro-Clientes.
   10 Clientes-Nome          pic x(010).
   10 Clientes-Telefone      pic x(030).
```

Forma de armazenamento de dos registros:

```
Registro 1                        Registro 2
....|....1....|....2....|....3....|....4....|....5....|....6....|....7....|...
Fabio     055 011 3044-5032           Leandro   055 011 3044-5032
```

É desta forma que o compilador irá armazenar os registros na estrutura de arquivo denominada seqüencial. Note que ao somarmos o total de bytes do registro obtemos 40, logo, a cada 40 bytes temos o início de um novo registro. O arquivo não possui nenhum caractere especial de terminação.

***Line Sequential***

A expressão, linha seqüencial, significa que os registros serão armazenados em linhas, ou seja, ao término do registro existirão caracteres de controle que indicam a finalização dos dados. No caso dos ambientes PC. Este formato de arquivo contém uma seqüência x"0D0A" (Carriage return + Line feed = ENTER), esta seqüência de caracteres fará com que o próximo registro seja gravado na linha seguinte, outro diferencial deste arquivo é que a seqüência x"0D0A" vem logo depois do último byte diferente de espaço, isto significa que se o registro possui 50 caracteres e o programa grava somente 1 byte então a linha terá o tamanho de 1 byte, isto afeta significamente no tamanho do arquivo. O formato de arquivo denominado *Line Sequential* é similar ao que costumeiramente chamamos de arquivo texto (TXT). Os caracteres de finalização variam conforme o sistema operacional. Em ambientes baseados em Unix, por exemplo, os caracteres de finalização x"0D0A" são substituídos por um simples x"0A". Por exemplo:

Definição do registro:

```
01 Registro-Clientes.
   10 Clientes-Nome          pic x(010).
   10 Clientes-Telefone      pic x(030).
```

Forma de armazenamento de dois registros:

```
....|....1....|....2....|....3....|....4
Fabio     055 011 3044-5032x"0D0A"
Leandro   055 011 3044-5032x"0D0A"
```

Note que apesar do registro possuir 40 bytes (Nome x(10) + Telefone x(30)) a sequência x"0D0A" foi adicionada logo após o último caractere do registro.

***Relative***

A expressão relativo significa que os registros serão armazenados em linhas e sua chave será a posição relativa em que ele se encontra dentro deste arquivo, ou seja, assim como um line sequential cada registro estará em uma linha com a diferença de que cada registro preenche toda a linha, mesmo que o campo com 50 bytes esteja sendo gravado com um único byte o compilador irá preencher o restante da linha com espaços até completar os 50 bytes, em seguida os caracteres de finalização x"0D0A" (Lembrando que em outros ambientes operacionais estes caracteres podem mudar). Desta forma, temos um registro de tamanho fixo e quando dizemos que queremos acessar o registro relativo 10, basta o compilador multiplicar o tamanho do registro por 10 e acessar a posição *offset* referente ao mesmo.

***Indexed***

É muito importante que um programa possibilite ao usuário um acesso rápido aos registros desejados, para tanto existe um recurso não apenas no Cobol, mas em muitas outras linguagens e bancos de dados, chamado *Chave*. Uma chave é um campo do registro que foi eleito como um meio de acesso direto a um determinado registro ou um conjunto de campos, esta última definição é conhecida com *split-key* ou chave-dividida. Esta chave normalmente possui um conteúdo numérico, mas, pode possuir outros valores. Existe também a possibilidade de se fazer uma pesquisa rápida por campos que não sejam a chave primária, estes são chamados de *Chaves Alternadas*, estas chaves podem, opcionalmente, possuir valores repetidos, por exemplo, um nome.

Este é o tipo de arquivo do compilador Cobol de maior utilização, devido aos seus recursos de pesquisa extremamente rápidos. Um arquivo indexado é composto por dois arquivos: O arquivo de dados e o arquivo de índice. Este arquivo auxiliar de índice possui mecanismos para encontrar os registros rapidamente através de uma referência do valor da *chave* com sua posição dentro do arquivo. Esta *chave* pode possuir duas características: *RECORD KEY* e *ALTERNATE KEY*, onde a *RECORD KEY* é referenciada a um campo do registro o qual irá possuir valores que nunca se repetirão, já a chave *ALTERNATE KEY* possui esta mesma característica mas possui artifícios para permitir a duplicação de valores. Podemos pesquisar pelas duas chaves, mas uma de cada vez.

Veja o seguinte exemplo:

```
       identification division.
       program-id.  AccessFile.
       author.      DTS Latin America Software.
      *
       environment division.
       configuration section.
       input-output section.
       file-control.
           select clientes assign to "clientes.dat"
           organization is indexed
           access mode is dynamic
           record key is clientes-codigo
           alternate key is clientes-nome with duplicates
           file status is fs-clientes.
      *
       data division.
       file section.
       fd clientes.
       01 reg-clientes.
          10 clientes-codigo          pic 9(006).
          10 clientes-nome            pic x(050).
          10 clientes-data-nasc.
             20 clientes-data-nasc-ano pic 9(004).
             20 clientes-data-nasc-mes pic 9(002).
             20 clientes-data-nasc-dia pic 9(002).
          10 clientes-telefone        pic x(030).
          10 clientes-endereco        pic x(050).
          10 clientes-bairro          pic x(030).
          10 clientes-cidade          pic x(030).
```

```
          10 clientes-estado          pic x(002).
          10 clientes-cep             pic 9(008).
          10 clientes-e-mail          pic x(050).
       *
        working-storage section.
        01 fs-clientes.
          10 fs-clientes-1            pic x(001).
          10 fs-clientes-2            pic x(001).
          10 fs-clientes-r redefines fs-clientes-2 pic 99 comp-x.
       *
        ...
        ...
```

Note que neste trecho de código vemos a *ENVIRONMENT DIVISION*, esta divisão foi comentada anteriormente e vamos disseca-la. Esta divisão possui a seguinte estrutura:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
I-O CONTROL.
```

Esta divisão é composta de duas seções:

***Configuration Section***

Esta seção destina-se a configuração do ambiente, ela é composta por três partes: *SOURCE-COMPUTER, OBJECT-COMPUTER* e *SPECIAL-NAMES*. Uma identifica o computador onde foi confeccionado o programa e a segunda identifica o computador do ambiente de produção, ambas servem apenas para comentários haja visto que ambas deixaram de ser obrigatórias mas caso o programador opte em utiliza-las deve escrever sua sintaxe corretamente para não causar erros de compilação. *SPECIAL-NAMES* específica o sinal monetário, escolhe o tipo de ponto decimal, específica caracteres simbólicos e possibilitar adaptar o programa para se comunicar com programas de outras linguagens.

***Input-Output Section***

Esta seção destina-se a configuração do ambiente de *Leitura* e *Gravação*, ela possui duas partes: *FILE-CONTROL* e *I-O-CONTROL*. A primeira destina-se a especificação dos arquivos que o programa irá acessar. A segunda foi descontinuada nas versões mais atuais do compilador, valendo apenas para os ambientes de *Mainframe*.

```
*
environment division.
configuration section.
input-output section.
file-control.
    select clientes assign to "clientes.dat"
```

```
           organization is indexed
           access mode is dynamic
           record key is clientes-codigo
           alternate key is clientes-nome with duplicates
           file status is fs-clientes.
       *
```

Neste exemplo, o programa não faz referências as cláusulas SOURCE-COMPUTER, OBJECT-COMPUTER e SPECIAL-NAMES, as duas primeiras por não serem obrigatórias e a terceira pelo fato do programa não precisar, por exemplo, identificar que as casas decimais serão identificadas por uma vírgula.

Como ele irá tratar um arquivo, temos a seção *INPUT-OUTPUT SECTION*, a referência desta seção é opcional, podemos escrever diretamente a cláusula *FILE-CONTROL*. O comando *SELECT* é utilizado para declarar um arquivo no programa, todos os arquivos que o programa irá manipular devem possuir uma *SELECT*.

A cláusula *SELECT* terá a seguinte estrutura para arquivos *seqüenciais*:

```
SELECT nome-arquivo ASSIGN TO nome-externo
ORGANIZATION IS SEQUENTIAL
ACCESS MODE IS SEQUENTIAL
FILE STATUS IS fs-arquivo.
```

*Nome-arquivo* indica o nome que será referenciado internamente pelo programa. *Nome-externo* indica onde os dados serão gravados, ou seja, o nome do arquivo, caso não seja especificado o caminho do arquivo será considerado o diretório corrente. Pode-se substituir este *nome-externo* por *DISK* e especificar o nome-externo na FD (será visto mais à frente) e se mesmo assim o nome-externo não for informado o compilador irá criar no diretório corrente de execução um arquivo com o mesmo *nome-arquivo*. *ORGANIZATION IS SEQUENTIAL* indica o tipo de organização do arquivo. *ACCESS MODE IS SEQUENTIAL* o tipo de acesso (sendo que este é o único tipo de acesso permitido para esta organização). *FILE STATUS IS fs-arquivo* representa uma referência de um campo da *Working-Storage section* que será atualizada a cada operação feita no arquivo, indicando qual é a situação atual do arquivo, é desta forma que o programa pode fazer testes para saber se pode ou não continuar a execução sem problemas.

A cláusula *SELECT* terá a seguinte estrutura para arquivos de linhas *seqüenciais*:

```
SELECT nome-arquivo ASSIGN TO nome-externo
ORGANIZATION IS LINE SEQUENTIAL
ACCESS MODE IS SEQUENTIAL
FILE STATUS IS fs-arquivo.
```

Note que a única mudança é na organização do arquivo *LINE SEQUENTIAL*. Todas as outras informações são idênticas ao *SEQUENTIAL*.

A cláusula *SELECT* terá a seguinte estrutura para arquivos randômicos:

```
SELECT nome-arquivo ASSIGN TO nome-externo
ORGANIZATION IS RELATIVE
ACCESS MODE IS RANDOM/SEQUENTIAL/DYNAMIC [RELATIVE KEY IS chave-randomica]
FILE STATUS IS fs-arquivo.
```

Nesta *SELECT* temos uma modificação na organização que agora é *RELATIVE* e temos a possibilidade escolher entre três tipos de acesso:

***RANDOM (Aleatório)***

O programa pode via *chave-randômica* ir para qualquer registro, informando apenas o número do registro que se deseja acessar.

***SEQUENTIAL (Seqüencial)***

O programa irá acessar este arquivo apenas para leituras seqüências, ou seja, irá avançar a leitura registro a registro até o fim de todos os registros que estão no arquivo.

***DYNAMIC (Dinâmico)***

O programa pode fazer acessos do tipo *RANDOM* e também do tipo *SEQUENTIAL*, ou seja, o programa pode alternar o tipo de acesso no momento que for oportuno.

A cláusula *SELECT* terá a seguinte estrutura para arquivos indexados:

```
SELECT nome-arquivo ASSIGN TO nome-externo
ORGANIZATION IS INDEXED
 ACCESS MODE IS RANDOM/SEQUENTIAL/DYNAMIC
RECORD KEY IS nome-campo/chave-separada
ALTERNATE KEY IS nome-campo/chave-separada [WITH DUPLICATES]
FILE STATUS IS fs-arquivo.
```

Nesta organização indexada temos duas novas cláusulas *RECORD KEY* e *ALTERNATE KEY*, elas referenciam-se aos campos do registros pelo qual será criado o índice, opcionalmente a *ALTERNATE KEY* pode possuir valores duplicados, ou seja, mais de um registro onde o campo referenciado como uma chave alternada terá um valor repetido.

Exercícios:

1- Definição de um arquivo indexado com a seguinte estrutura (Exerc6.cbl):

Clientes

| | |
|---|---|
| Codigo | numérico de seis posições (Chave) |
| Nome | alfanumérico de 50 posições (Chave alternada com duplicações) |
| Data de nascimento | numérico de 8 posições |
| Telefone | alfanumérico de 30 posições |
| Endereco | alfanumérico de 50 posições |
| Numero | numérico de seis posições |
| Complemento | alfanumérico de 30 posições |
| Bairro | alfanumérico de 30 posições |
| Cidade | alfanumérico de 30 posições |
| Cep | numérico de 8 posições |
| Estado | alfanumérico de 2 posições |
| E-mail | alfanumérico de 50 posições |

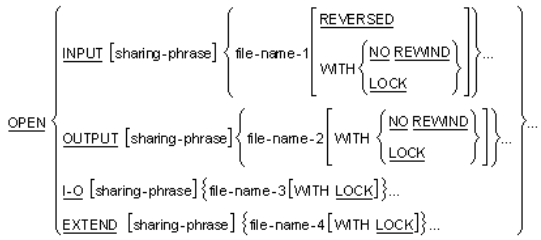
Crie parte de um programa com todas as definições necessárias para este arquivo, utilizando a IDENTIFICATION, ENVIRONMENT e DATA DIVISION. Este programa se chamará APLIC.CBL.

### *Manipulando as informações*

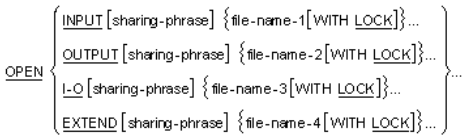
Até agora foram vistas todas as divisões do programa, suas principais seções, seus tipos de arquivos nativos, seus tipos de variáveis e alguns comandos de *PROCEDURE*, agora iremos ver os comandos utilizados para tratar os arquivos e os dados contidos neles.

Antes de tratar qualquer dado dos registros de um arquivo precisamos *abri-lo* e então começar a manipular os registros. O comando para abrir um arquivo é o *OPEN* e sua estrutura é:

Formato 1 (Para arquivos seqüências somente)




Formato 2 (Para todos os arquivos)




O comando *OPEN* irá disponibilizar um canal com o arquivo para que o programa faça as operações necessárias. *INPUT* indica que o programa poderá apenas fazer leituras no arquivo não podendo fazer nenhuma gravação, para tanto o arquivo deverá ter sido criado anteriormente. *OUTPUT* indica que o programa poderá apenas gravar no arquivo, não podendo fazer leituras, o arquivo não poderá existir, caso exista ele será sobreposto por um arquivo vazio. *I-O* possibilitará ao programa fazer as operações de leitura/gravação, este é o tipo de acesso mais utilizado. *EXTEND* tem a mesma funcionalidade de *OUTPUT* com o diferencial de não sobrepor o arquivo os registros existentes, mas complementando-o, ou seja, quando o arquivo for aberto o *ponteiro* de registros será posicionado logo após o último registro existente e adicionará ao arquivo os registros gravados. *REVERSED* fará com que os registros serão acessados do último para o primeiro. *WITH LOCK* destina-se ao tratamento do arquivo em rede, isto será visto mais à frente.

Depois de abrir o arquivo, nós podemos introduzir novas informações, atualizar informações existentes, apagar informações e ainda consultar os dados existentes. Para cada uma dessas ações existem comandos específicos, conforme vistos a seguir:

***Gravando novas informações***

O comando utilizado para gravar novos registros em um arquivo é o *WRITE* este comando insere um novo registro no arquivo baseado nas informações contidas no registro informado. Este comando é utilizado da seguinte forma:

```
WRITE registro-arquivo
```

O *registro-arquivo* referência-se ao item de grupo informado como sendo o registro do arquivo especificado na *SELECT* do programa. No caso de arquivos com estrutura *SEQUENTIAL*, *LINE SEQUENTIAL* e RELATIVE o comando irá gravar o registro e caso alguma coisa de errado aconteça então será retornada um *FILE STATUS* diferente de zeros. É sempre muito importante verificar o *status* do arquivo afim de constatar se está tudo bem. No caso do *INDEXED* além de uma verificação idêntica à anterior será constatado se não existem uma situação de *chave duplicada*, esta situação irá acontecer caso o programa tente gravar um registro onde o valor do campo definido como *RECORD KEY* seja idêntico ao de um registro já gravado no arquivo. Esta mesma situação pode ocorrer para campo que tenham sido definidos como chaves do tipo *ALTERNATE KEY* com um diferencial, caso o programador deseje pode existir uma chave duplicada, basta que a *ALTERNATE KEY* deste campo tenha a cláusula *WITH DUPLICATES*, desta forma obteremos um *FILE STATUS* que indicará que o registro foi gravado com sucesso com uma de suas chaves alternadas duplicadas.

***Regravando novas informações***

Sempre existirá a necessidade de um programa *atualizar* os registros de um arquivo, esta atualização é feita com o seguinte comando:

```
REWRITE registro-arquivo
```

Antes de se utilizar estes comandos deveremos *carregar* este registro para nossa *área de FD*, ou seja, no caso de arquivos indexados devemos *ler* o registro que deverá ser atualizado, fazer as modificações necessárias e então regravá-los. Isto deve ser feito para se evitar a perda de informações, por exemplo:

Registro original:

```
COD   NOME                          ESTADO
000001FABIO FERREIRA COSTA                SP
```

Situação da *área de FD* após o *OPEN*:

```
COD   NOME                          ESTADO
```

Imagine que este é o registro que está atualmente gravado em disco e o programa deverá alterar o nome que está armazenado, pois bem, ao abrirmos o arquivo o nosso registro possivelmente valerá espaços, o programa já sabe o código do registro a ser alterado e sabe que o nome deve ser alterado para FABIO COSTA mas não sabe qual é o valor de estado, logo o registro deve ser lido para que todas estas informações estejam na *área de FD*, assim o programa pode alterar o nome e regravar o registro sem nenhum perigo de perder alguma informação previamente gravada.

### *Apagando informações*

Um arquivo às vezes necessita *ser limpo*, ou seja, registros que não interessam mais estarem no cadastro devem ser removidos, isto faz parte da *manutenção de cadastro* e o comando a seguir executa esta operação:

```
DELETE arquivo
```

O comando *DELETE* irá remover o registro que está atualmente apontado pelo *ponteiro de registros*, ou seja, devemos primeiro fazer uma leitura do registro assim como é necessário para o comando *REWRITE*. No caso de arquivos indexados podemos simplesmente mover o valor da chave a ser removida e executar o comando. Caso o registro informado não seja encontrado, será devolvido um *FILE STATUS* indicando que não foi possível encontrar o registro.

### *Consultando informações*

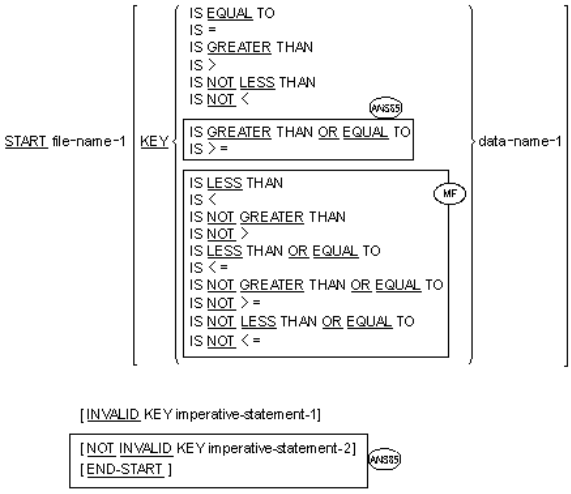
Para termos acesso aos registros contidos nos arquivos devemos ler estes registros, para tanto utilizamos o seguinte comando:

```
READ arquivo [NEXT/PREVIOUS]
```

O comando *READ* irá carregar o registro para a *área de FD* definida da *FILE SECTION*. Para todas as estruturas de arquivos podemos executar um *READ arquivo NEXT* fazendo uma leitura do próximo registro, somente para arquivos relativos e indexados, podemos fazer um *READ arquivo PREVIOUS* que permitirá o acesso do registro anterior ao atual. Para arquivos indexados podemos mover o valor para sua chave e efetuar um simples *READ* arquivo, isto irá posicionar o ponteiro de registros.

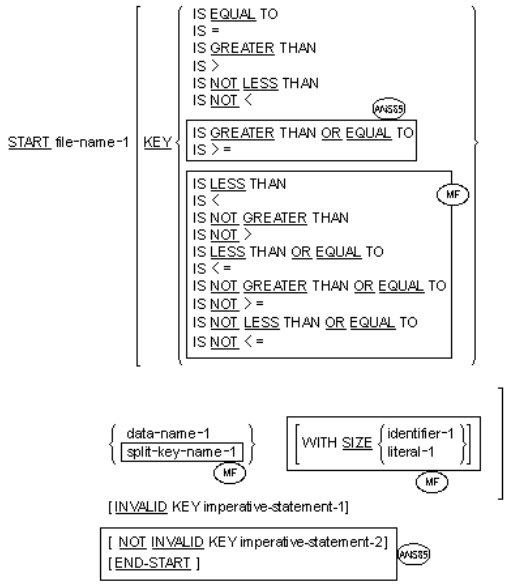
Existem situações em que precisamos saltar alguns registros para processarmos um lote específico de registros, por exemplo, o programa deve processar todos os registros em que o estado seja igual a "SP". Para esta tarefa utilizamos o comando *START*, sua sintaxe é:

Formato 1 (Arquivos relativos):




Para os arquivos de estrutura *RELATIVE* o *data-name-1* indica a posição relativa do registro a ser encontrado. Note que o *START* aceita condições de pesquisa assim como o *IF*.

Formato 2 (Arquivos indexados):




Para arquivos indexados o *data-name-1* ou a *split-key-name-1* representam as chaves do arquivos especificados na *SELECT* do programa. Uma observação importante sobre o comando *START* é que ele sempre se utiliza de chaves para posicionar os registros, não pode-se utilizar campos que não sejam *RECORD KEY* ou *ALTERNATE KEY*. O exemplo a seguir vemos um arquivo onde o campo estado foi definido na SELECT como uma chave alternada com duplicações permitidas (caso contrário não poderíamos ter mais do que um registro por estado) e o objetivo do programa é exibir todos os registros onde estado seja igual a "SP":

```
       identification division.
       program-id.    ListaClientes.
       author.        DTS Latin America Software.
      *
       environment division.
       configuration section.
       input-output section.
       file-control.
           select clientes assign to "clientes.dat"
           organization is indexed
           access mode is dynamic
           record key is clientes-codigo
           alternate key is clientes-estado with duplicates
           file status is fs-clientes.
      *
       data division.
       file section.
       fd clientes.
       01 reg-clientes.
          10 clientes-codigo          pic 9(006).
```

```
           10 clientes-nome             pic x(050).
           10 clientes-data-nasc.
              20 clientes-data-nasc-ano pic 9(004).
              20 clientes-data-nasc-mes pic 9(002).
              20 clientes-data-nasc-dia pic 9(002).
           10 clientes-telefone         pic x(030).
           10 clientes-endereco         pic x(050).
           10 clientes-bairro           pic x(030).
           10 clientes-cidade           pic x(030).
           10 clientes-estado           pic x(002).
           10 clientes-cep              pic 9(008).
           10 clientes-e-mail           pic x(050).
       *
        working-storage section.
        01 fs-clientes.
           10 fs-clientes-1             pic x(001).
           10 fs-clientes-2             pic x(001).
           10 fs-clientes-r redefines fs-clientes-2 pic 99 comp-x.
       *
        procedure division.
        inicio.
           open input clientes
           move "SP" to clientes-estado
           start clientes key is equal clientes-estado
           if fs-clientes equal zeros
             read clientes next
             perform until fs-clientes equal "10" or
                           clientes-estado unequal "SP"
               display clientes-codigo "-" clientes-nome
               read clientes next
             end-perform
           end-if
           close clientes
           stop run
           .
```

Em todos os comandos vistos para tratamento de registros dos arquivos de estrutura *RELATIVE* e *INDEXED* existe uma cláusula opcional chamada *INVALID KEY comando*. Esta cláusula executará o comando, sempre que existir um erro na chave do registro (duplicado ou não encontrado). Pode-se executar todos os comandos da linguagem exceto os comandos condicionais *IF* e *EVALUATE*.

### *Fechando os arquivos*

Depois de terminar de tratar os registros dos arquivos é necessário fechar este arquivo. Para tanto utilizamos o comando *CLOSE*, conforme sintaxe a seguir:

```
CLOSE file-name-1 [WITH LOCK] [file-name-2 [ WITH LOCK ] ] ...
```

Caso seja especificada a cláusula *WITH LOCK*, o programa não poderá reabrir este arquivo durante a sua execução. Esta cláusula *WITH LOCK* não tem nenhuma referência com tratamento de rede.

Exercícios:

1º - Crie um pequeno sistema que faça o controle do registro do Exerc6.cbl para que o mesmo faça o tratamento completo (Inclusão, Alteração, Consulta e Exclusão) do arquivo de clientes.

### *Ordenando as informações*

Em certas situações existe a necessidade do programa listar as informações ordenadas por campos que não estão especificados como chaves do arquivo, para que isto seja possível existe um mecanismo chamado *SORT*. O *SORT* é um método da linguagem onde podemos ordenar determinados campos em ordem ascendente ou descendente por qualquer campo ou grupo de campos. Vamos pegar o arquivo anterior como exemplo e supor que necessitássemos listar os registros ordenados pelo e-mail. Veja o exemplo a seguir:

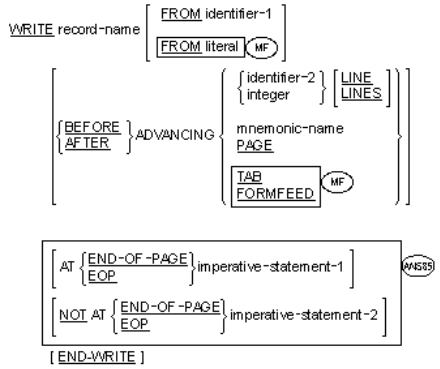
```
       identification division.
       program-id.    ListaClientes.
       author.        DTS Latin America Software.
      *
       environment division.
       configuration section.
       input-output section.
       file-control.
           select clientes assign to "clientes"
           organization is indexed
           access mode is dynamic
           record key is clientes-codigo
           alternate key is clientes-estado with duplicates
           file status is fs-clientes.

           select ord-clientes assign to "sortcli.dat"
           sort status is ss-clientes.
      *
       data division.
       file section.
       fd clientes.
       01 reg-clientes.
          10 clientes-codigo          pic 9(006).
          10 clientes-nome            pic x(050).
```

```
      10 clientes-data-nasc.
         20 clientes-data-nasc-ano pic 9(004).
         20 clientes-data-nasc-mes pic 9(002).
         20 clientes-data-nasc-dia pic 9(002).
      10 clientes-telefone         pic x(030).
      10 clientes-endereco         pic x(050).
      10 clientes-bairro           pic x(030).
      10 clientes-cidade           pic x(030).
      10 clientes-estado           pic x(002).
      10 clientes-cep              pic 9(008).
      10 clientes-e-mail           pic x(050).

   sd ord-clientes.
   01 sort-reg-clientes.
      10 sort-clientes-codigo      pic 9(006).
      10 sort-clientes-nome        pic x(050).
      10 sort-clientes-e-mail      pic x(050).
  *
   working-storage section.
   01 ss-clientes.
      10 ss-clientes-1             pic x(001).
      10 ss-clientes-2             pic x(001).
      10 ss-clientes-r redefines ss-clientes-2 pic 99 comp-x.
   01 fs-clientes.
      10 fs-clientes-1             pic x(001).
      10 fs-clientes-2             pic x(001).
      10 fs-clientes-r redefines fs-clientes-2 pic 99 comp-x.
  *
   procedure division.
   inicio.
      sort ord-clientes ascending sort-clientes-e-mail
           input procedure sortin-clientes
           output procedure sortout-clientes
      stop run
      .
  *
   sortin-clientes.
      open input clientes
      read clientes next
      perform until fs-clientes equal "10"
        move clientes-codigo to sort-clientes-codigo
        move clientes-nome to sort-clientes-nome
        move clientes-e-mail to sort-clientes-e-mail
        release sort-reg-clientes
        read clientes next
      end-perform
      close clientes
      .
  *
   sortout-clientes.
      perform pega-reg-sort
      perform lista-clientes until ss-clientes equal "10"
      .
```

```
*
 lista-clientes.
    display sort-clientes-codigo
    display sort-clientes-nome
    display sort-clientes-e-mail
    display "-------------------"
    perform pega-reg-sort
    .
*
 pega-reg-sort.
    return ord-clientes
          at end move "10" to ss-clientes
    .
```

***Impressão***

A impressão dos dados pelo programa COBOL também é muito simples, basta o programa possuir uma referência de um arquivo de estrutura LINE SEQUENTIAL associado a um *device* válido para impressão. A partir de então utilizamos os mesmos comandos de tratamento de arquivo para tratar a impressão. É claro, o comando READ não terá efeito sobre este tipo de arquivo. O comando WRITE é provido de alguns mecanismos para a impressão das linhas, conforme sintaxe vista a seguir:




Note que nesta estrutura do *WRITE* existem as cláusulas *BEFORE* e *AFTER* que indicam ao compilar como deve fazer a impressão da nova linha. *AT END-OF-PAGE* executará um comando quando atingir o fim da página, por exemplo um *PERFORM* para imprimir um cabeçalho. A seguir temos um exemplo de impressão.

```
 identification division.
 program-id.   Impressao.
 author.       DTS Latin America Software.
*
 environment division.
 configuration section.
 file-control.
     select clientes assign to "clientes.dat"
     organization is indexed
     access mode is dynamic
```

```
           record key is clientes-codigo
           alternate key is clientes-estado with duplicates
           file status is fs-clientes.
       *
           select impressao assign to "LPT1"
       *    >>> As duas linhas abaixo tambem sao validas <<<
       *    select impressao assign to "SAIDA.TXT"
       *    select impressao assign to "\\Estacao\Impressora"
           organization is line sequential
           access mode is sequential
           file status is fs-impressao.
       *
        data division.
        file section.
        fd clientes.
        01 reg-clientes.
           10 clientes-codigo          pic 9(006).
           10 clientes-nome            pic x(050).
           10 clientes-data-nasc.
              20 clientes-data-nasc-ano pic 9(004).
              20 clientes-data-nasc-mes pic 9(002).
              20 clientes-data-nasc-dia pic 9(002).
           10 clientes-telefone        pic x(030).
           10 clientes-endereco        pic x(050).
           10 clientes-bairro          pic x(030).
           10 clientes-cidade          pic x(030).
           10 clientes-estado          pic x(002).
           10 clientes-cep             pic 9(008).
           10 clientes-e-mail          pic x(050).
        fd impressao linage is 66 lines with footing at 64
                   lines at top 2 lines at bottom 1.
        01 reg-impressao.
           10 impressao-filler         pic x(080).
       *
        working-storage section.
        77 Pagina                     pic 9(006) value zeros.
        01 fs-clientes.
           10 fs-clientes-1           pic x(001).
           10 fs-clientes-2           pic x(001).
           10 fs-clientes-r redefines fs-clientes-2 pic 99 comp-x.
        01 fs-impressao.
           10 fs-impressao-1          pic x(001).
           10 fs-impressao-2          pic x(001).
           10 fs-impressao-r redefines fs-impressao-2 pic 99 comp-x.
        01 Linha-cabecalho.
           10 filler                  pic x(031) value spaces.
           10 filler                  pic x(017) value "Lista de Client
       -                                          "es".
        01 Linha-cabecalho-labels.
           10 filler                  pic x(006) value "Codigo".
           10 filler                  pic x(001).
           10 filler                  pic x(050) value "Nome".
           10 filler                  pic x(001).
```

```
      10 filler                pic x(010) value "Nasc.".
      10 filler                pic x(001).
      10 filler                pic x(030) value "Telefone".
   01 Linha-cabecalho-linha.
      10 filler                pic x(006) value all "=".
      10 filler                pic x(001).
      10 filler                pic x(050) value all "=".
      10 filler                pic x(001).
      10 filler                pic x(010) value all "=".
      10 filler                pic x(001).
      10 filler                pic x(030) value all "=".
   01 Linha-detalhe.
      10 detalhe-codigo          pic z(006).
      10 filler                pic x(001).
      10 detalhe-nome             pic x(050).
      10 filler                pic x(001).
      10 detalhe-nasc.
        20 detalhe-dia           pic 9(002).
        20 filler               pic x(001) value "/".
        20 detalhe-mes            pic 9(002).
        20 filler               pic x(001) value "/".
        20 detalhe-ano            pic 9(004).
      10 filler                pic x(001).
      10 detalhe-telefone         pic x(030).
   01 Linha-rodape.
      10 filler                pic x(066) value spaces.
      10 filler                pic x(008) value "Pagina: ".
      10 rodape-pagina            pic z(006) value zeros.
  *
   procedure division.
   inicio.
      open input clientes
      open output impressao
      read clientes next
      if fs-clientes equal zeros
        perform imprime-cabecalho
        perform until fs-clientes equal "10"
          move clientes-codigo to detalhe-codigo
          move clientes-nome to detalhe-nome
          move clientes-data-nasc-dia to detalhe-dia
          move clientes-data-nasc-mes to detalhe-mes
          move clientes-data-nasc-ano to detalhe-ano
          move clientes-telefone to detalhe-telefone
          perform imprime-linha
          read clientes next
        end-perform
      end-if
      close clientes impressao
      stop run
      .
  *
   imprime-cabecalho.
      write reg-impressao from Linha-cabecalho
```

```
                before advancing 3 line
   write reg-impressao from Linha-cabecalho-labels
                before advancing 1 line
   write reg-impressao from Linha-cabecalho-linha
                before advancing 1 line
   .
*
 imprime-linha.
   write reg-impressao from Linha-detalhe
                before advancing 1 line
                at end-of-page
                  perform imprime-rodape
                  perform imprime-cabecalho
   .
*
 imprime-rodape.
   add 1 to pagina
   move pagina to rodape-pagina
   write reg-impressao from spaces
                before advancing 1 line
   write reg-impressao from Linha-rodape
                before advancing page
   .
```

Exercícios:

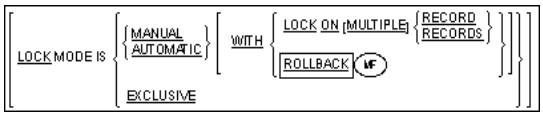
1º - Complemente o sistema anterior adicionando ao sistema uma opção para impressão da lista de clientes cadastrados.

### *Ambientes multiusuário (Redes)*

Em ambientes multiusuário a aplicação deve se preocupar com uma situação que não ocorre quando esta trabalhando sozinha com o arquivo: *acessos simultâneos*. Existem dois tipos de acessos simultâneos: Arquivos e Registros. Estes acessos devem ser controlados pelo programa de acordo com as ações que da aplicação. Por exemplo, quando a aplicação necessita alterar alguns dados de um determinado registro é importante que o programa faça um *LOCK* (bloqueio) do registro, garantindo assim que outros usuários não façam alterações antes dele.

O bloqueio pode ser feito ao nível de arquivo ou ao nível de registro. Enquanto sua aplicação, esta com o arquivo ou registro *bloqueado*, não será possível que outros usuários da rede possam acessar aquela informação. Os *bloqueios* podem ser feitos por métodos automáticos e manuais. Os métodos automáticos garantem uma maior segurança as alterações no registro, pois, não há como ocorrer os riscos do programador esquecer de bloquear os registros com os quais irá trabalhar.

Para que uma aplicação possa ser capaz de bloquear um registro, ela deve possuir a seguinte cláusula na *SELECT* do arquivo:




Quando o tipo de *lock* é *MANUAL*, estamos informando que o programa irá fazer o *bloqueio* e o *desbloqueio*, para tanto quando o programa for ler um registro, por exemplo, pode-se adicionar a cláusula *WITH [KEPT] LOCK*:

```
read clientes with kept lock
```

Isto irá bloquear o registro para que outros usuários não possam alterar este registro. Para desbloquear o registro, utilizamos o UNLOCK:

```
unlock clientes records
```

Caso o programa tenha a cláusula *AUTOMATIC* não será necessário a utilização do bloqueio no *READ*, para cada leitura feita será feito um bloqueio automático e o registro anteriormente bloqueado será desbloqueado, agora, se a aplicação necessitar bloquear mais do que um registro então utiliza-se a cláusula *WITH LOCK ON MULTIPLE RECORDS*, já que o padrão é *WITH LOCK ON RECORD*. *EXCLUSIVE* é o padrão, mas pode-se especifica-lo. *ROLLBACK* utilizada junto com um produto da *Merant Micro Focus* chamado *FileShare*, que possibilita à aplicação desfazer a gravação de registros se necessário.

Exercícios:

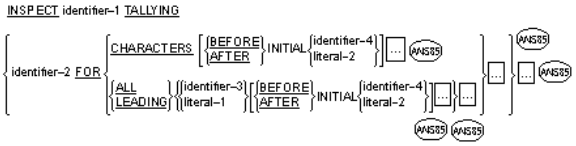
1º - Altere o programa Exerc6.cbl para que o mesmo faça o controle para ambientes multiusuário.

## ***Tópicos adicionais***

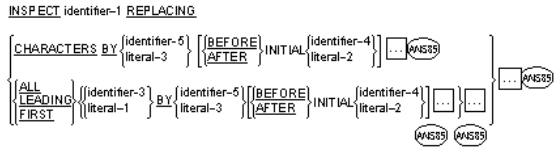
### ***Tratamentos de dados alfanuméricos***

Existem três comandos de grande utilidade para o tratamento de cadeias de caracteres ou *strings*, são eles: *INSPECT*, *STRING* e *UNSTRING*. A estrutura do comandos *INSPECT* é:

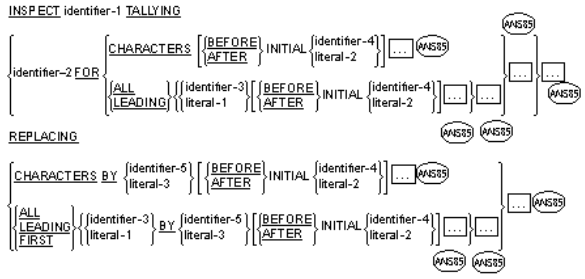
Formato 1:



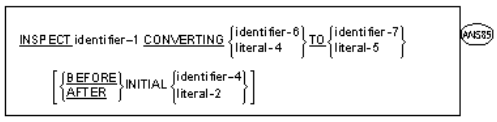

Formato 2:



Formato 3:




Formato 4:



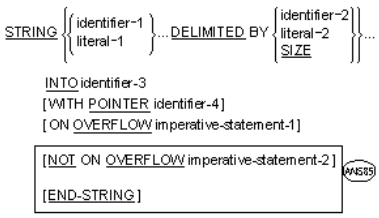

O comando *INSPECT* possui vários recursos para a manipulação de *strings*, dentre estes recursos os mais utilizados são os recursos para identificação de um determinado caractere ou *string* em uma variável ou ainda substituir caracteres de uma variável. O comando a seguir pesquisa a existência do caractere "a" dentro da variável *nome*:

```
Inspect nome tallying existe for all "a"
```

Caso exista algum caractere "a" dentro da variável nome a variável *existe* terá o seu valor modificado para a quantidade de ocorrências de "a" dentro da variável *nome*. O comando a seguir substitui as ocorrências de ":" por ";":

```
Inspect linha replacing all ":" by ";"
```

O comando *STRING* é utilizado para unificar o conteúdo de variáveis. Sua sintaxe é vista a seguir:




Este comando é muito utilizado quando desejamos, por exemplo, montar o nome completo de um cliente a partir dos campos *Primeiro-Nome* e *Ultimo-Nome*, por exemplo:

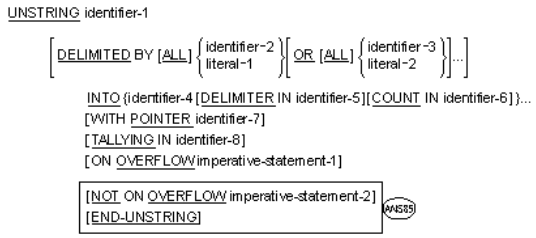
```
String primeironome delimited by spaces
    " " delimited by size
    segundonome delimited by spaces
    into nomecompleto
```

Neste exemplo a variável primeironome possui "fabio" e segundonome possui "costa", desta forma nomecompleto irá possuir "fabio costa". As cláusulas *DELIMITED BY* indicam ao comando *STRING* o que deve ser considerado para delimitar os caracteres unificados. Pode-se informar uma string para *DELIMITED BY*, por exemplo:

```
String primeironome delimited by "a"
    " " delimited by size
    segundonome delimited by "o"
    into nomecompleto
```

*Nomecompleto* será igual a "fc".

O comando *UNSTRING* possui as características inversas de *STRING*. Sua sintaxe é:




A diferença básica entre STRING e UNSTRING, é que o primeiro, une os bytes informados em uma variável e UNSTRING desune os bytes em várias variáveis.

### *Recuperando arquivos corrompidos*

Em certas circunstâncias pode ocorrer uma corrupção dos arquivos indexados. Esta corrupção pode ocorrer quando o programa está fazendo alguma operação de gravação e por algum motivo esta operação é encerrada no meio do processo. Quando se grava um registro em um arquivo indexado, o programa deve atualizar o arquivo de dados e depois o arquivo de índice, se ocorrer a situação em que o arquivo de dados é atualizado e o de índice não, irá existir a corrupção do índice, ou seja, o arquivo de índice deixará de estar equalizado com o arquivo de dados. Para corrigir estes problemas existe um utilitário chamado *REBUILD*. Este utilitário foi desenvolvido exclusivamente para corrigir estes problemas e oferece uma série de recursos, onde pode se reconstruir o índice a partir do arquivo de dados simplesmente. Sua sintaxe mais simples é: *REBUILD arquivo.dat*.