



# PROGRESS 4GL



## Sumário

<b>Capítulo 1: A PROGRESS SOFTWARE</b>	3
<b>APRESENTAÇÃO</b>	3
<b>HISTÓRICO VERSÕES</b>	5
<b>PROPOSTA</b>	6
<b>CONVENÇÕES DA APOSTILA</b>	7
<b>A LINGUAGEM</b>	7
<b>EDITORES</b>	8
<b>CHARACTER</b>	8
<b>GRÁFICO</b>	9
<b>PRODUTOS PROGRESS</b>	10
<b>CHARACTER</b>	10
<b>CLIENT / SERVER</b>	10
<b>WEBSPEED</b>	11
<b>PONTUAÇÃO</b>	12
<b>Capítulo 2: BANCO DE DADOS</b>	14
<b>O BANCO DE DADOS</b>	14
<b>O MODELO DE DADOS</b>	15
<b>A TRANSAÇÃO</b>	16
<b>ATOMICIDADE</b>	16
<b>CONSISTÊNCIA</b>	16
<b>ISOLAMENTO</b>	16
<b>DURABILIDADE</b>	16
<b>O CONTROLE DE CONCORRÊNCIA</b>	19
<b>SEGURANÇA EM BANCO DE DADOS</b>	19
<b>DICIONÁRIO DE DADOS</b>	20
<b>RELACIONAMENTOS</b>	20
<b>Capítulo 3: DEFININDO BANCO DADOS</b>	21
<b>CRIANDO UM BANCO DE DADOS</b>	22
<b>CRIANDO UMA TABELA</b>	23
<b>CRIANDO CAMPOS NA TABELA</b>	25
<b>CRIANDO ÍNDICES</b>	27
<b>CRIANDO SEQUENCES</b>	28
<b>Capítulo 4: RECURSOS DA LINGUAGEM</b>	30
<b>DECLARANDO VARIÁVEIS</b>	30
<b>DECLARANDO UM FRAME</b>	32
<b>OPERADORES LÓGICOS</b>	34
<b>Capítulo 5: COMANDOS</b>	40
<b>COMANDOS DE REPETIÇÃO</b>	40
<b>REPEAT</b>	40
<b>FOR</b>	41
<b>COMANDOS DE CONDIÇÃO</b>	42
<b>IF .. THEN .. ELSE</b>	43



CASE .....	44
COMANDOS DE SELEÇÃO .....	45
COMANDOS DE SAÍDA .....	48
COMANDO MESSAGE .....	49
COMANDO PUT .....	50
COMANDOS E FUNÇÕES DIVERSOS .....	50
Capítulo 6: MANIPULAÇÃO DE DADOS .....	81
CRIANDO UM REGISTRO .....	82
LENDO UM REGISTRO .....	83
DELETANDO UM REGISTRO .....	85
Capítulo 7: MANIPULAÇÃO DE TABELAS .....	86
TEMP-TABLE .....	86
BUFFER .....	90
ORDENAÇÃO .....	92
Capítulo 8: DISPOSITIVOS DE SAÍDA .....	93
OUTPUT TO .....	93
EM TELA .....	94
NA IMPRESSORA .....	95
EM ARQUIVO .....	96
NA ÁREA DE TRANSFERÊNCIA .....	97
EM STREAM .....	99
Capítulo 9: INTERAGINDO COM OUTROS PROGRAMAS .....	100
PROCEDURES EXTERNAS .....	100
PROCEDURES INTERNAS .....	101
PASSAGEM PARÂMETROS .....	102
INCLUDES .....	105
EXTENSÕES PROGRESS .....	106
Capítulo 10: AMBIENTE .....	108
SISTEMA OPERACIONAL .....	108
MANIPULANDO O SO .....	109
COMPILAÇÃO .....	110
PROPATH .....	112
ÁREAS DE ARMAZENAMENTO .....	113
Capítulo 11: OTIMIZAÇÃO .....	115
UTILIZAÇÃO DO IF .....	116
UTILIZAÇÃO DE BLOCOS .....	117
UTILIZAÇÃO DO CAN-FIND .....	119
VARIÁVEIS LÓGICAS .....	120
REDUÇÃO DAS TRANSAÇÕES .....	120
PARÂMETRO NO-UNDO .....	122
RESPEITE O ÍNDICE .....	122

# Capítulo 1: A PROGRESS SOFTWARE



Capítulo 1 – Progress Software

1	• Apresentação
2	• Proposta Curso
3	• Convenções da Apostila
4	• A Linguagem
5	• Editor
6	• Produtos Progress
7	• Pontuação

LHC

## APRESENTAÇÃO

Desde 1981 Progress® Software Corporation gera produtos de software e serviços que fortalecem os seus parceiros e clientes para melhorar drasticamente a sua elaboração, implantação, integração e aplicações de gestão de qualidade em todo o mundo.

Progress® Software foi fundada em 1981 como linguagem de programação por Joseph Alsop, Clyde Kessel, e Charles Ziering. Centrando-se na Aplicação do desenvolvimento e implantação de software, apresentou a sua linguagem Progress® para gestão de dados, aplicativos e desenvolvimento em meados dos anos 1980.

A Progress® Software pelo início da década de 1990 havia melhorado os seus produtos com interfaces para bases de dados como a Oracle. A empresa liberou a seu público em 1991 a primeira versão de seu software Windows.

Em 1990 tinha estabelecido um Progress®o global, o que permite a Progress® oferecer ferramentas para o desenvolvimento de aplicações baseadas em Java.

A Progress® em 1999 criou uma nova unidade para oferecer produtos e serviços a fornecedores, independentes de software. Em 2001 a empresa dividiu seus negócios em três unidades operacionais, que constitui em desenvolvimento de software empresarial Progress® Company (uma divisão da Progress® Software), e formando as filiais Sonic Software Corporation (messaging software) e NuSphere Corporation (open source software e serviços). Acrescentados novos negócios, ex: unidade PeerDirect Corporation para aplicações de computação distribuída em 2002. Nesse ano a empresa adquiriu database

eXcelon empresa de software por US\$ 24 milhões em uma jogada para expandir suas ofertas para incluir serviços e banco de dados baseado em XML.

No ano seguinte, a Progress® adquiriu a DataDirect Technologies, fornecedor de conectividade de dados disponível, por cerca de US\$ 88 milhões. A aquisição da Persistence Software por US\$ 16 milhões em 2004 e Apama por cerca de US\$ 25 milhões em 2005 ajudou a expandir a sua unidade ObjectStore.

Durante ano fiscal de 2006, a empresa continuou a realizar Progress®os significativos no seu objetivo estratégico de proporcionar um mais rico e mais amplo portfólio de produtos para ajudar as empresas e Profissionais de TI em desenvolvimento, implantação, integração e gestão de aplicativos empresariais críticos.

No início do ano fiscal de 2006, adquiriu NEON Progress® Software Systems, fornecedor líder de mainframe o acesso aos dados e integração de software, e Actional Corporation, um fornecedor de uma arquitetura orientada para serviços (SOA) software de gestão.

Em junho de 2006, adquiriu Pantero Progress® Software, o primeiro produto da indústria para resolver os dados em tempo real integração semântica desafios através de uma abordagem orientada para modelo. Em outubro de 2006, a empresa adquiriu OpenAccess Software, uma fornecedora de ferramentas para desenvolvimento rápido de ODBC e JDBC, bem como ADO.NET e fornecedores OLE DB.

Em junho de 2008, adquiriu Xcalia Progress® Software, uma integração de dados líder que iscombined com DataDirect Technologies e reforça o apoio da empresa para acessar os dados com base em padrões. Também em junho, adquiriu Mindreef Progress® Software Inc. desenvolve e comercializa Mindreef da premiada Mindreef® SOAPscope produtos®, que permite que as empresas, analistas, arquitetos sistema, desenvolvedores, testadores, operações e pessoal de apoio para criar, implantar e manter um melhor software em cada fase de uma arquitetura SOA, Web Services ou compostos de aplicação e desenvolvimento ciclo de vida.

Em 2010 reconhecimento pelo quinto ano consecutivo da plataforma OpenEdge pela International Data Corporation.

O nascimento do Progress® 11 veio no ano de 2011ofertando algumas novas possibilidades para administração de seu banco de dados, também neste ano adquiriu uma nova empresa a Corticon, com um conjunto de ferramentas que possibilita o Analista de Negócios criar e administrar as regras de negócio gerando implicitamente códigos de fontes automaticamente.

Com mais uma aquisição, desta vez a Rollbase surge mais uma possibilidade no portfólio em 2013 esta permite a criação de aplicações SaaS com rapidez diretamente na nuvem, surgindo também nesta oportunidade a plataforma Pacific.

Para maiores informações, seguem os links do fornecedor, também colocamos os dados referentes ao fechamento fiscal de 2015.

<https://www.Progress.com>

<https://www.Progress.com.br>

<https://www.Progress.com.br/Corticon>

<https://www.Progress.com.br/Rollbase>

<https://www.Progress.com.br/Pacific>

<http://investors.Progress.com/releasedetail.cfm?ReleaseID=950042>

# HISTÓRICO VERSÕES

Uma das características da linguagem é sempre manter em funcionamento programas compilados em versões anteriores, ou seja, não perde funcionalidades ao longo do tempo e sim apenas incrementa novas e mais sofisticadas.

Segue um resumo dos principais acontecimentos no Progress® ao longo dos anos apenas como referência.

<b>1981</b> Fundação da Empresa	<b>1997</b> Versão 8.2B	<b>2003</b> Versão 10.0A	<b>2008</b> Versão 10.2A
<b>1984</b> Versão 2.1	<b>1998</b> Webspeed 2.0	Webservices IN	ABL GUI for .NET
<b>1984</b> Versão 2.2	<b>1998</b> Versão 8.2C	State-free AppServer	Windows 64bit
R-code	<b>1998</b> Versão 8.3A	ProDataSets	Structured error handling
<b>1985</b> Versão 3.0	Bistall/bithold	Type II Storage areas	TTY mode dynamic browse widget
Frames	More VST	OpenClient for .NET	Stream
ON & Apply	<b>1998</b> Versão 9.0A	Array parameters	OOABL static members
COLOR	ADM2	BLOB	OOABL garbage collection
<b>1987</b> Versão 4.0	PUB/SUB	CLOB	OOABL interface properties
Workfiles	AppBuilder	Mergeprop	OOABL array support
CHOOSE	Dynamic queries	<b>2004</b> Versão 9.1E	XML read/write enhancements
<b>1988</b> Versão 5.0	Superprocedures	Failover clusters	ProDataSet WHERE-STRING
Fastrack	Stateless mode	<b>2004</b> Versão 10.0B	<b>2009</b> Versão 10.2B
Client/Server	OpenClient for Java	Webservices OUT	OOABL abstract classes
SQL89	Type I storage areas	Ttmarshal	OOABL reflection updates
Federated DBs	<b>1999</b> Versão 8.3B	Encryption	Dynamic DLL/SO invocation
EDITING	Rereadnolock	DateTime	ProDataSet WRITE JSON
<b>1989</b> Versão 6.0	<b>1999</b> Versão 9.0B	Message digests	STOP-AFTER
DataServers	<b>1999</b> Versão 9.1A	Base64-encoding	<b>2011</b> Versão 11.0
Parametros	Integrated ABL & Webspeed	<b>2005</b> Versão 10.1A OO	Multi-tenancy
RESULTS (reports)	Dynamic Temp Table	ABL	Table-scan
<b>1991</b> Versão 6.3	Dynamics	SAX writer	JSON built-in objects
APW	1st large file support	Auditing	Temp-table blocking from funVSTs
Backup On-line	Async Appserver	Client-Principal	ProDataSet infer relations from XML
<b>1991</b> Versão 7.0	DOM XML parsing	By-reference parameters	Remove IO action
GUI (User Interface Graphic)	Memory mapped procedure library	Events in batch mode	OOABL interface inheritance
E-D model	Dynamic browser	OpenClient dynamic API	Fix 64-bit r-code
Temp Table	BLOBs	ProDataSet read/write xml	GUI for .NET everywhere
UIB (User Interface Builder)	Socket handling	Webservices-out w/temp-tables/PDS	OOABL dynamic property accessors
DLL calls	<b>2000</b> Versão 8.3C	<b>2006</b> Versão 9.1E04	<b>2012</b> Versão 11.1
Internal Procedures	<b>2000</b> Versão 9.1B	<b>2007</b> Versão 10.1B	<b>2013</b> Versão 11.2



Word indexes	SQL92	Major removing of limits on DB	<b>2013</b> Versão 11.3
Report Builder	Webclient	Double colon notation	REST and Mobile
<b>1993</b> Versão 7.2	JMS API	64 bit datatypes	Block-level undo
ODBC	AIA adapter	64-bit dbkeys	Throw
<b>1994</b> Versão 7.3A	<b>2001</b> Versão 8.3D	Indeterminate arrays	ABL single-run
<b>1995</b> Versão 7.3B	<b>2001</b> Versão 9.1C	XML xref	Dynamic access to built-ins
<b>1995</b> Versão 7.3C	WebClient Intellistream	Extend longchar support	<b>2013</b> Versão 10.2B08
<b>1995</b> Versão 8.0A	<b>2002</b> Versão 9.1D	OOABL strongly typed events	<b>2014</b> Versão 11.3.1
<b>1996</b> Versão 8.0B	New debugger	Log-manager "4glTrans"	<b>2014</b> Versão 11.3.2
<b>1996</b> Versão 8.1A	WinXP appearance	Browser column view-as	<b>2014</b> Versão 11.4
<b>1997</b> Webspeed 1.0	SAX reader	Browser sort arrows	Mobile (improved)
<b>1997</b> Versão 7.3E	WebClient over Internet	Color inheritance in GUI	Table Partitioning (improved)
<b>1997</b> Versão 8.2A		OOABL overloading	ABL Unit Testing
AppServer (state aware/reset)		<b>2008</b> Versão 10.1C	<b>2015</b> Versão 11.5
UDF		Client stack trace VST	<b>2015</b> Versão 11.6
VST		Throw/catch/finally	Webspeed (improved)
DB block size		Dynamic browse for ChUI	

## PROPOSTA

Este documento está descrito sobre a linguagem 4GL (Fourth Generation Language) que foi substituída na versão OpenEdge para ABL (Absolute Business Language) em modo texto (character), tendo vários exemplos com base no banco de dados nativo de documentação que vem junto com a instalação do PROGRESS® (banco sports).

As sintaxes dos comandos estão descritas conforme a versão 10.2B do Progress® (Open Edge®), nas situações que forem citados exemplos de algum comando/função que não exista em todas as versões, será indicado a partir de qual versão este comando/função começou a existir.

Este material possui a proposta de levar o conceito da linguagem como também do banco de dados para aqueles que não tiveram ainda contato com a tecnologia ou para quem está iniciando no desenvolvimento baseado em Progress®.

Acreditamos que com este material existirá um início com base sólida, o profissional poderá obter maiores e satisfatórios resultados no futuro.

Os dados aqui contidos foram retirados dos manuais do Progress®, de vários outros documentos acessíveis na internet, e de experiências pessoais e devidamente traduzido para português e tabulado para facilitar o aprendizado. Compilados em uma visão de treinamento empresarial por Sandro Carvalho, o qual se reserva de todos os direitos autorais do referido material.

## CONVENÇÕES DA APOSTILA

No decorrer da apostila, você vai se identificar com algumas convenções que a Progress® utiliza para identificar as SINTAXES de seus comandos, funções, atributos, parâmetros, etc. Conforme exemplo abaixo:

Exemplo:

```
TRIGGERS:
{ ON event-list [ ANYWHERE ]
  { trigger-block
    | PERSISTENT RUN procedure
      [ IN handle ]
      [ ( input-parameters ) ]
  }
} ...
END [ TRIGGERS ]
```

Segue explicação destas convenções para o melhor entendimento desta apostila, bem como o entendimento das leituras de manuais produzidos pela Progress®, para facilitar utilizaremos as mesmas características do material original, que se encontra disponível no site da Progress® escritos na língua inglesa.

Quando não existir símbolo nenhum, então o comando, parâmetro é obrigatório.

[ ] Colchetes, significa que o que estiver dentro, é opcional.

| Pipe, significa que você utiliza o comando anterior ao símbolo pipe ou utiliza o comando que está depois do símbolo pipe.

## A LINGUAGEM

Com dialeto proprietário, a PROGRESS SOFTWARE CORPORATION disponibiliza junto ao banco de dados uma poderosa linguagem de 4ª Geração, com inúmeros comandos, funções, pré-processadores, e uma infinidade de ferramentas que possibilita a construção de códigos robustos e enxutos, podendo-se fazer muita coisa com pouco código escrito. Uma das grandes vantagens desta linguagem é o seu poder de reaproveitamento do mesmo código para diversos outros sistemas operacionais, tais como:

- MS-DOS;
- SCO;
- Unix;
- HP - UX
- IBM - AIX;
- Linux;
- OS2
- WINDOWS 32 e 64 BITS
- Entre outros



## EDITORES

A Progress® possui duas versões de editores de texto, uma versão para a quem pretende trabalhar no modo CHARACTER e outra versão para quem trabalho no modo GRÁFICO.

Podemos ainda contar com a edição de programas Progress® em outros editores comuns, alguns até possuem artifícios para identificar comandos com as mesmas cores utilizadas no editor padrão da linguagem. Porém estes editores “não Progress®” não tem algumas facilidades que a linguagem oferece, entre elas a possibilidade de verificar erro de sintaxe em tempo real, ou até mesmo compilar o programa a partir do fonte aberto dentro do editor. Na sequência segue uma breve descrição sobre cada um dos dois modelos de editores do Progress® OpenEdge.

Existe ainda a possibilidade de se trabalhar no Open Edge® com o Architect que é uma versão Progress® em forma de Cockpit, para isto existe um plug-in que pode ser anexado ao Eclipse para aqueles que já trabalham em outras linguagens de mercado orientadas a objeto, facilitando assim o trabalho de se familiarizar com a ferramenta.

## CHARACTER

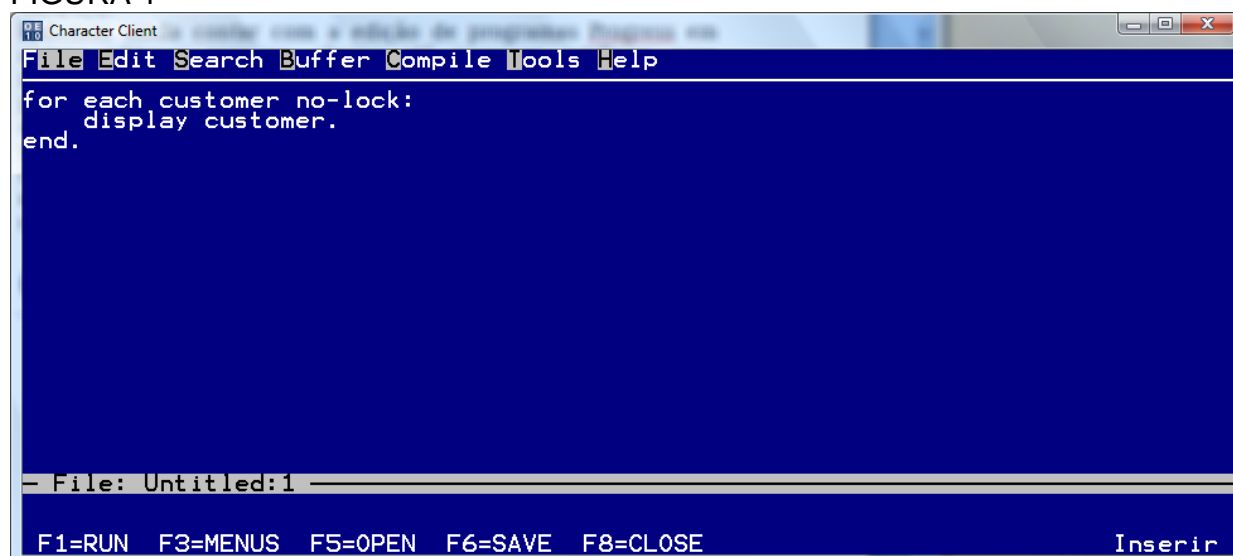
O Editor character tem o formato apresentado na Figura 1. É um editor extremamente funcional, remanescente das primeiras versões da linguagem, estritamente funcional, leve e objetivo para suprir quaisquer funcionalidades para atualização, ajustes, etc., de um determinado programa.

Este editor peca em usabilidade, suas funções consideradas corriqueiras em outros editores do mercado não são de simples percepção, citamos como exemplo um simples Copy + Paste, que passa a não ser algo tão simples como aparenta.

Outro fato que atrapalha e muito é que ele não possui UNDO, ou seja, se você errar em uma alteração vai ter que reconstituir o código manualmente, pois o mesmo não consegue voltar para a situação imediatamente anterior.

Porém para acessar diretamente servidores Linux/Unix é a única opção existente.

FIGURA 1

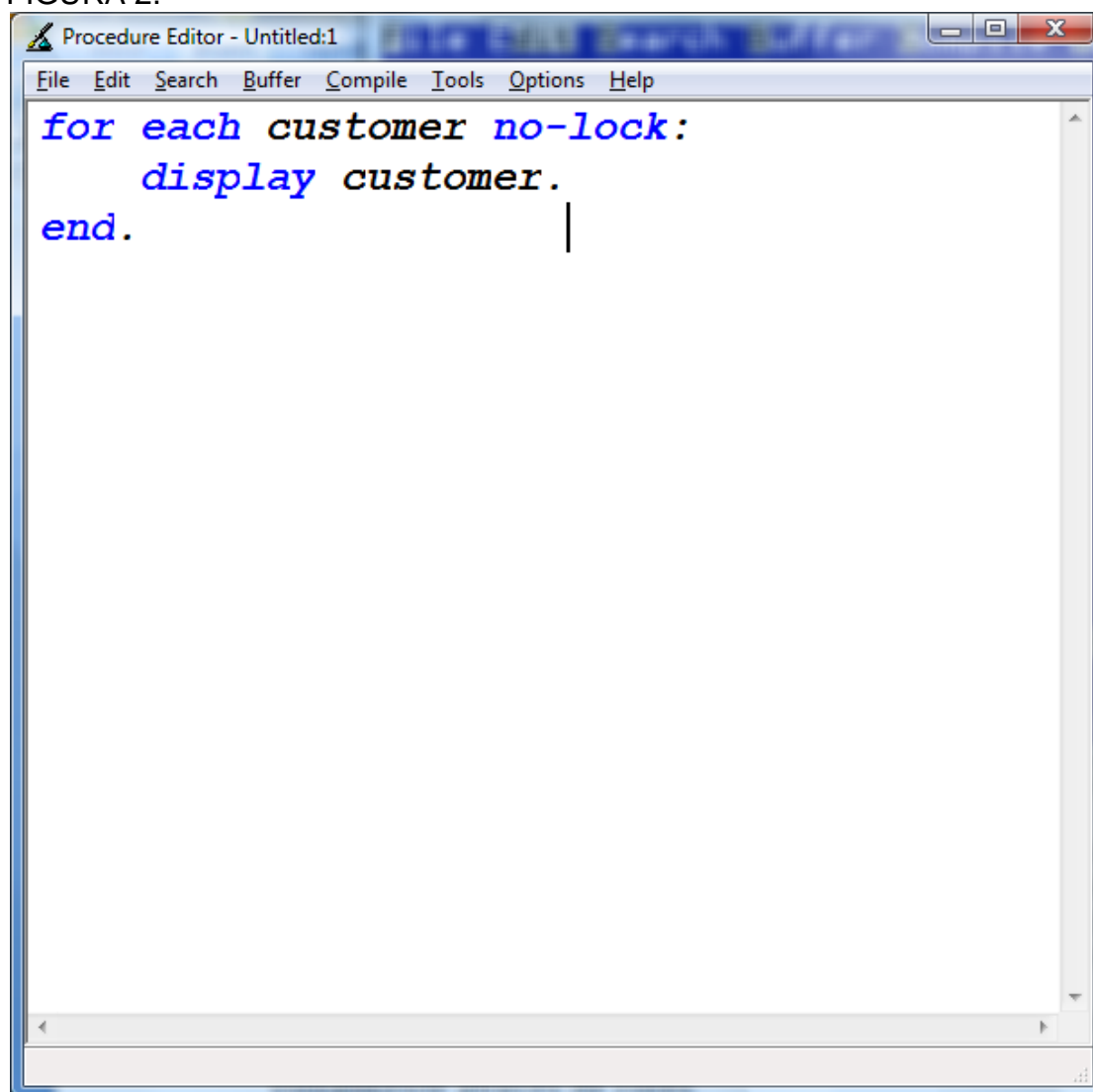


## GRÁFICO

O Editor Gráfico possui tudo o que um editor comercial tem, com um formato também familiar e mais atual, conforme apresentado na Figura 2. É um editor como a exemplo do editor character, que tem todas as funcionalidades da linguagem para sintaxe e compilação. Cabe ressaltar que permite configurações de alinhamento do fonte, bem como um acelerador de digitação de comandos, através da sessão de aliás.

Possui UNDO, que funciona nos mesmos moldes dos editores convencionais, permite também com que os comandos sejam identificados visualmente através de cores.

FIGURA 2.



## PRODUTOS PROGRESS

Neste material o foco é a linguagem ABL, que serve de base para vários produtos Progress®. Abaixo segue exemplo dos três mais utilizados comercialmente, onde qualquer comando que aprendermos neste curso, poderá ser utilizado nestas plataformas.

Existem outros produtos que podem ser citados ao longo do curso oportunamente ou de exemplo em algum cenário que não estão citados abaixo, grande parte dos produtos existentes fazem parte da lista da evolução da linguagem apresentada no tópico HISTÓRICO DAS VERSÕES deste material.

## CHARACTER

A versão character foi a pioneira da Progress® Software, todos os sistemas eram procedurais na época e com limitações de tecnologias, para a época nem o Windows era uma plataforma tão difundida e as linguagens de programação em geral seguiam uma certa característica peculiar que era a tela “preta”.

Hoje ainda a versão do Progress® character roda em muitos sistemas bons, pois é uma versão muito leve que não consome banda e nem rede para ser acessada em locais com dificuldades de comunicação/rede.

Inclusive a própria TOTVS até hoje mantém os agendadores de execução (RPW's) em versão character para quem tem a plataforma Linux.

Exemplo:

## CLIENT / SERVER

A versão Client Server foi a evolução natural da Progress® Software e das demais linguagens de programação que conseguiram se manter no mercado, que foi passar a

utilizar as características do Windows, em um primeiro momento orientada a eventos, aproveitando ao máximo o mouse, barras de rolagem e tudo que a plataforma Windows trazia consigo.

A maioria dos ERP's escritos em Progress®, estão na versão client server.

Exemplo:

Implantação Item - CD0204 - 2.00.00.051 -

Arquivo Ajuda

Item: 00 00

Descrição: .

Básico Complem Narrativa Com. Exterior

Grupo Estoque: 98 ITEM PADRAO P/COMPRA S/MOV EST

Família: 9999996 FAMÍLIA DÉBITO DIRETO

Família Comercial: GERAL FAMÍLIA GENÉRICA

Unid Medid: UM UNIDADE

Estabelecimento Padrão: PAD ESTABELECIMENTO PADRAO

Situação: Ativo

Data Implantação: 11/07/2000 Data Liberação: 11/07/2000

## WEBSPEED

Na sequência da evolução natural da tecnologia a Progress Software criou uma nova versão de produto que permitia que a linguagem ABL pudesse ser utilizada para fazer aplicações via WEB.

Foi mantida muita coisa, das duas versões de produtos anteriores, para fazer com que os amantes da Progress® não fossem tendenciosos a mudar de plataforma de desenvolvimento, já que o mundo web era irreversível.

A tecnologia que mantém as conexões de bancos e a geração das telas, é bem diferente e na época do lançamento por este motivo também era muito cara, fazendo deste motivo o principal empecilho para a o sucesso da nova plataforma.

Exemplo:



## PONTUAÇÃO

Esta seção contém entradas de referência que descrevem o idioma ABL. Ela começa com descrições da pontuação referenciadas pela Progress® e caracteres especiais. Segue com descrições das declarações, funções, etc...

: (Dois Pontos)

O dois pontos (:) símbolo que indica bloco, também rótulos de bloco para repetição (labels).

. (Ponto Final)

O ponto final (.) é o símbolo que termina todas as declarações, inclusive bloco declarações de cabeçalho. Também é um caminho de diretório ou separador de sufixo de arquivo na maioria das plataformas. Os laços de repetição podem terminar com um ponto final ou com os dois pontos.

, (Vírgula)

A vírgula (,) símbolo que separa especificações de arquivos múltiplas (ex: comando FOR), e argumentos múltiplos de uma função.

? (Ponto Interrogação)

O ponto de interrogação é um caráter especial que representa o valor desconhecido, sempre aparece quando o Progress® não consegue executar uma determinada função.

() Precedência de Expressão

Parênteses elevam precedência de expressão, assim como matemática, também, algumas funções lhe exigem que inclua argumentos em parênteses, por exemplos passagem de parâmetros.

### [ ] Referência de Array

Colchetes ([ ]) inclui subscrições de array ([1], [2], etc.) ou intervalos (como, [1 FOR 4]). Em um intervalo, você pode usar uma variável para o primeiro elemento, mas o segundo elemento deve ser uma constante. Colchetes também são usados ao especificar valores iniciais por uma ordem.

### { } Referência de argumento

Referências o valor de um argumento que um procedimento passa a um arquivo de procedimento externo chamado ou para um include.

### /\* \*/ Comentários

Toda a sentença escrita entre esta combinação de caracteres vai ser considerada um comentário e consequentemente desconsiderada pelo compilador do Progress®.

### + Sinal Adição

Este sinal pode ser usado em diversos casos, seja para fazer adição de dois números inteiros, decimais, datas e strings, conforme exemplos abaixo:

<b>Inteiro</b>	$1 + 1 = 2$
<b>Decimal</b>	$10.3 + 10.5 = 20,8$
<b>Datas</b>	$12/31/2015 + 1 = 01/01/2016$

### - Sinal Subtração

Este sinal como a exemplo do sinal de adição, pode ser usado em diversos casos, seja para fazer subtração de dois números inteiros ou decimais e datas, conforme exemplos abaixo:

<b>Inteiro</b>	$1 - 1 = 0$
<b>Decimal</b>	$10.3 - 10.5 = -0,2$
<b>Datas</b>	$12/31/2015 - 1 = 30/12/2015$

### / Sinal Divisão

Este sinal serve para fazer somente divisões de valores entre dois números inteiros ou decimais, conforme exemplos abaixo:

<b>Inteiro</b>	$2 / 2 = 1$
<b>Decimal</b>	$10.8 / 2.0 = 5,4$

### \* Sinal Multiplicação

Este sinal serve para fazer somente multiplicações de valores entre dois números inteiros ou decimais, conforme exemplos abaixo:

<b>Inteiro</b>	$2 * 2 = 4$
<b>Decimal</b>	$10.8 * 2.0 = 21,6$

## Capítulo 2: BANCO DE DADOS



## O BANCO DE DADOS

Banco de dados relacional com dicionário de dados ativo, com um enorme potencial de armazenamento e agilidade é o que a linguagem Progress® traz consigo.

Tem como prioridade a integridade dos dados que estão inseridos em suas tabelas, fazendo isto de forma ágil e transparente ao usuário final.

Bancos de dados (português brasileiro) ou bases de dados (português europeu) são coleções de informações que se relacionam de forma a criar um sentido. São de vital



importância para empresas, e há duas décadas se tornaram a principal peça dos sistemas de informação. Normalmente existem por vários anos sem alterações em sua estrutura. São operados pelos Sistemas Gerenciadores de Bancos de Dados (SGBD), que surgiram na década de 70. Antes destes, as aplicações usavam sistemas de arquivos do sistema operacional para armazenar suas informações. Na década de 80 a tecnologia de SGBD relacional passou a dominar o mercado, e atualmente utiliza-se praticamente apenas ele, que é onde se enquadra a estrutura RDBMS da Progress®. A principal aplicação de um Banco de Dados é o controle das operações empresariais.

## O MODELO DE DADOS

O modelo de base de dados Progress® possui as características elencadas abaixo, fazendo disso uma grande versatilidade para aplicação.

O modelo plano (ou tabular) consiste de matrizes simples, bidimensionais, compostas por elementos de dados: tipo inteiros, números reais, etc. Este modelo plano é muito próximo do desenho de uma planilha eletrônica.

Este modelo em rede permite que várias tabelas sejam usadas simultaneamente através do uso de apontadores (ou referências). Algumas colunas contêm apontadores para outras tabelas ao invés de dados. Assim, as tabelas são ligadas por referências, o que pode ser visto como uma rede. Uma variação particular deste modelo em rede, o modelo hierárquico, limita as relações a uma estrutura semelhante a uma árvore (hierarquia - tronco, galhos), ao invés do modelo mais geral direcionado por grafos.

Bases de dados relacionais consistem, principalmente de três componentes: uma coleção de estruturas de dados, nomeadamente relações, ou informalmente tabelas; uma coleção dos operadores, a álgebra e o cálculo relacionais; e uma coleção de restrições da integridade, definindo o conjunto consistente de estados de base de dados e de alterações de estados.

As bases de dados relacionais permitem aos utilizadores (incluindo programadores) escreverem consultas (queries) que não foram antecipadas por quem projetou a base de dados. Como resultado, bases de dados relacionais podem ser utilizadas por várias aplicações em formas que os projetistas originais não previram, o que é especialmente importante em bases de dados que podem ser utilizadas durante décadas. Isto tem tornado as bases de dados relacionais muito populares no meio empresarial.

O modelo relacional é uma teoria matemática desenvolvida por Edgar Frank Codd para descrever como as bases de dados devem funcionar. Embora esta teoria seja a base para o software de bases de dados relacionais, muito poucos sistemas de gestão de bases de dados seguem o modelo de forma restrita ou a pé da letra - lembre-se das 12 leis do modelo relacional - e todos têm funcionalidades que violam a teoria, desta forma variando a complexidade e o poder. A discussão se esses bancos de dados merecem ser chamados de relacional ficou esgotada com o tempo, com a evolução dos bancos existentes. Os bancos de dados hoje implementam o modelo definido como objeto-relacional.



# A TRANSAÇÃO

A transação é um conjunto de procedimentos que é executado em um banco de dados, que para o usuário é visto como uma única ação.

A integridade de uma transação depende de 4 propriedades, conhecidas como ACID.

## ATOMICIDADE

Todas as ações que compõem a unidade de trabalho da transação devem ser concluídas com sucesso, para que seja efetivada. Se durante a transação qualquer ação que constitui unidade de trabalho falhar, a transação inteira deve ser desfeita (rollback). Quando todas as ações são efetuadas com sucesso, a transação pode ser efetivada e persistida em banco (commit).

## CONSISTÊNCIA

Todas as regras e restrições definidas no banco de dados devem ser obedecidas. Relacionamentos por chaves estrangeiras, checagem de valores para campos restritos ou únicos devem ser obedecidos para que uma transação possa ser completada com sucesso.

## ISOLAMENTO

Cada transação funciona completamente à parte de outras estações. Todas as operações são parte de uma transação única. O princípio é que nenhuma outra transação, operando no mesmo sistema, possa interferir no funcionamento da transação corrente (é um mecanismo de controle). Outras transações não podem visualizar os resultados parciais das operações de uma transação em andamento (ainda em respeito à propriedade da atomicidade).

## DURABILIDADE

Significa que os resultados de uma transação são permanentes e podem ser desfeitos somente por uma transação subsequente. Por exemplo: todos os dados e status relativos a uma transação devem ser armazenados num repositório permanente, não sendo passíveis de falha por uma falha de hardware.

Podemos definir uma transação sendo um bloco de programação/código que altera e/ou cria dados no banco de dados ou ainda que faz leitura utilizando a cláusula EXCLUSIVE-LOCK.

A transação é que controla o fluxo de escrita (write) no banco de dados, isto é, a escrita no banco de dados só se dará por completa quanto a transação for terminada de forma normal, conforme o fluxo da programação. Se por ventura acontecer um erro qualquer, ou existir

intervenção do usuário, teclando ESC (gráfico), F4(character), CTRL-C, CTRL-BREAK, a transação termina de forma anormal, tendo como consequência desfazer todas as atualizações.

Trocando isso em miúdos, se você está atualizando uma tabela com 1500 registros e no 1498º registro você teclar CTRL-C, os 1497 registros atualizados nesta mesma transação irão se desfazer, voltando os valores anteriores a sua atualização, ocorrendo o chamado UNDO.

Tome cuidado com os blocos de transações grandes, pois se você trabalha em modo multiusuário, poderá haver LOCKS (travamentos por indisponibilidade de registros em uso exclusivo) e também tome cuidado com os blocos pequenos de transações, porque poderá haver inconsistências no banco de dados em casos em que uma informação depende única e exclusivamente de outra informação.

Tomamos como exemplo uma nota fiscal que possui o corpo da nota em uma transação e os produtos desta nota em outra transação. Como estão em transações pequenas e independentes, se ocorrer algum erro na hora de você digitar os produtos, o que você digitou a respeito dos produtos o PROGRESS® desfez e o que você digitou no corpo na nota, - como estava em outra transação independente – ficou gravado, o que torna esta informação inconsistente, pois existirá uma nota vagando em seu banco de dados, sem produtos.

Então cada caso deve ser analisado de forma a não trancar os outros usuários com LOCKS de registros e de forma a não deixar informações inconsistentes no banco de dados.

Para resolver este problema, bastaria criar uma única transação para o corpo da nota e para a digitação dos produtos.

Existem alguns comandos que “seguram” uma transação sendo eles:

- REPEAT
- FOR EACH
- DO ON ENDKEY
- DO ON ERROR
- DO TRANSACTION
- PROCEDURE

Existem comandos que acionam o início da transação, sendo eles comandos de acessos ao banco de dados, sendo estes:

- INSERT
- UPDATE <campo do banco de dados>
- CREATE
- DELETE
- ASSIGN <campo do banco de dados>
- SET <campo do banco de dados>
- FIND <utilizando a cláusula EXCLUSIVE-LOCK>

## Exemplo 1: Diferença de bloco sem transação e com transação

```

DEFINE VARIABLE vteste          AS INTEGER.

REPEAT:                          /* Segura a transação */
    UPDATE vteste.               /* Não altera dados no banco */
    LEAVE.                       /* Portanto não é transação */
END.

```

```

REPEAT:                          /* Segura a transação */
    FIND FIRST Customer EXCLUSIVE-LOCK.
    UPDATE Customer.             /* Altera dados no banco */
    LEAVE.
END.

```

## Exemplo 2: Exemplo de sub-transação

```

FOR EACH Customer               /* Segura a transação */
    EXCLUSIVE-LOCK:
        UPDATE Customer.        /* Altera dados banco */
        FOR EACH Order OF Customer /* Segura a sub-transação */
            EXCLUSIVE-LOCK:
                UPDATE order.    /* Altera dados banco */
            END.
        END.
    END.

```

Exemplo 3: Exemplo onde o primeiro bloco não é transação, somente o segundo

```
DEFINE VARIABLE vteste AS INTEGER.
```

```
REPEAT:                                     /* Segura a transação */
```

```
    UPDATE vteste.                        /* Não altera o banco */
```

```
    FOR EACH Customer
```

```
        EXCLUSIVE-LOCK: /* Segura a transação */
```

```
        UPDATE Customer.                /* Altera o banco */
```

```
    END.
```

```
LEAVE.
```

```
END.
```

## O CONTROLE DE CONCORRÊNCIA

O controle de concorrência é um método usado para garantir que as transações sejam executadas de uma forma segura e sigam as regras ACID. Os SGBD devem ser capazes de assegurar que nenhuma ação de transações completadas com sucesso (committed transactions) seja perdida ao desfazer transações abortadas (rollback).

Uma transação é uma unidade que preserva consistência. Requeremos, portanto, que qualquer escalonamento produzido ao se processar um conjunto de transações concorrentemente seja computacionalmente equivalente a um escalonamento produzido executando essas transações serialmente em alguma ordem. Diz-se que um sistema que garante esta propriedade assegura a seriabilidade ou também serialização.

## SEGURANÇA EM BANCO DE DADOS

O banco de dados é utilizado para armazenar diversos tipos de informações, desde dados sobre uma conta de e-mail até dados importantes da Receita Federal. A segurança do banco de dados herda as mesmas dificuldades que a segurança da informação enfrenta, que é garantir a integridade, a disponibilidade e a confidencialidade. Um Sistema gerenciador de banco de dados deve fornecer mecanismos que auxiliem nesta tarefa. Para evitar ataques, o desenvolvedor de aplicações deve garantir que nenhuma entrada possa alterar a estrutura da consulta enviada ao sistema.

## DICIONÁRIO DE DADOS

O dicionário de dados permite que você execute todas as tarefas de desenvolvimento necessárias para o banco de dados da sua aplicação.

Cria, modifica e deleta componentes do banco de dados, de forma interativa, pois na linguagem PROGRESS® o desenvolvimento e o banco de dados, convivem harmoniosamente, sendo que temos vários exemplos de um mesmo comando PROGRESS® manipular dados de tela, memória e banco de dados ao mesmo tempo, sem precisar da intervenção do usuário.

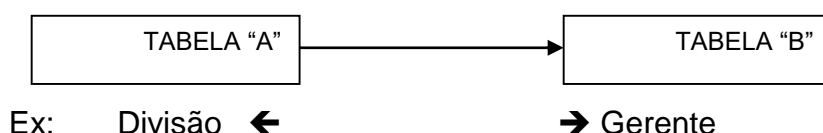
Segue abaixo um resumo prefácio de alguns termos utilizados no banco de dados PROGRESS®, e que por ser o Open Edge® uma linguagem que abre as fronteiras para outras aplicações e banco de dados, são termos que comumente veremos em outras linguagens também.

- **Banco de Dados** – um conjunto de tabelas relacionadas logicamente.
- **Tabelas** – um conjunto de registros relacionados logicamente organizados em linhas e colunas.
- **Campos** – um componente de um registro que armazena um valor de dado; também chamado de coluna.
- **Índices** – um campo ou grupo de campos que identifica os registros em uma tabela.
- **Sequences** – um objeto do banco de dados que fornece valores inteiros incrementais para uma aplicação.
- **Triggers** – uma procedure que é executada cada vez que um evento de banco de dados ocorrer, por exemplo, criar ou deletar registro.

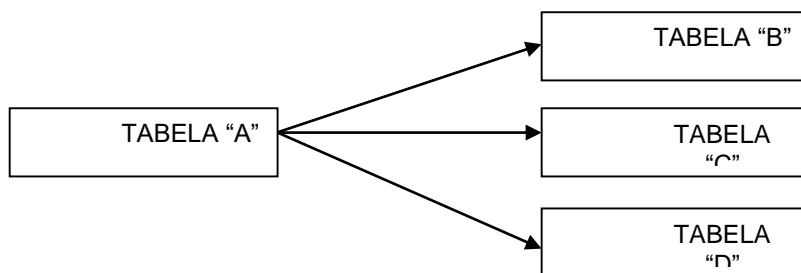
## RELACIONAMENTOS

Um banco de dados relacional fornece ligações entre os registros das mais variadas tabelas. Vamos abaixo citar os modelos de relacionamentos permitidos pelo Progress® Open Edge®, que seguem os preceitos e padrões estipulados nas melhores conjunturas de modelagem de dados.

Relacionamento “um-para-um” – um relacionamento de “um-par-um” existe quando um registro em uma tabela está diretamente relacionado com um registro em outra tabela.



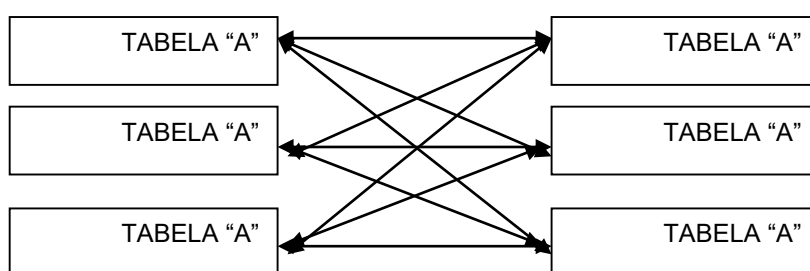
Um-para-vários – um relacionamento de um-para-vários existe quando um registro de uma tabela se relaciona com vários registros em uma outra tabela.



Ex: Pedido ←

→ Item do Pedido

Vários-para-vários – um relacionamento de vários-para-vários existe quando um registro em uma tabela possui relacionamento com vários registros em uma Segunda tabela. Além do mais, um registro na Segunda tabela possui um relacionamento com vários registros na primeira tabela.



Ex: Fornecedor ←

→ Produtos

## Capítulo 3: DEFININDO BANCO DADOS



## CRIANDO UM BANCO DE DADOS

Pode ser criado a partir da linha de comando ou através do menu Data Dictionary, primeiramente vamos abordar a forma mais rápida e depois a forma mais interativa.

Um banco é criado a partir de uma outra estrutura pré-definida, podendo ser:

Empty (cria um banco vazio);

Sports (cria um banco com as definições do banco de demonstração do Progress®);

Other database (Cópia fiel das definições de um outro banco existente).

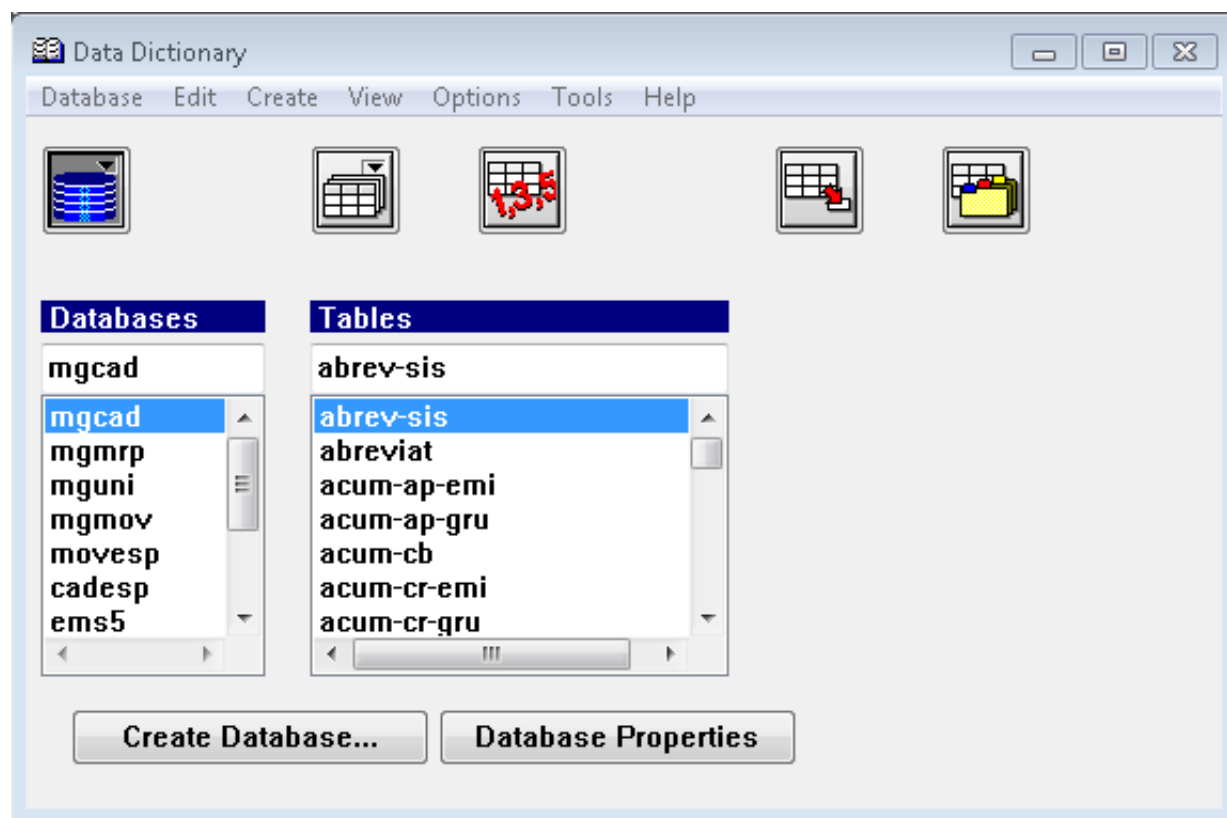
Criando um Banco de Dados via linha de comando:

**prodb <nome-do-banco-a-ser-criado> empty**

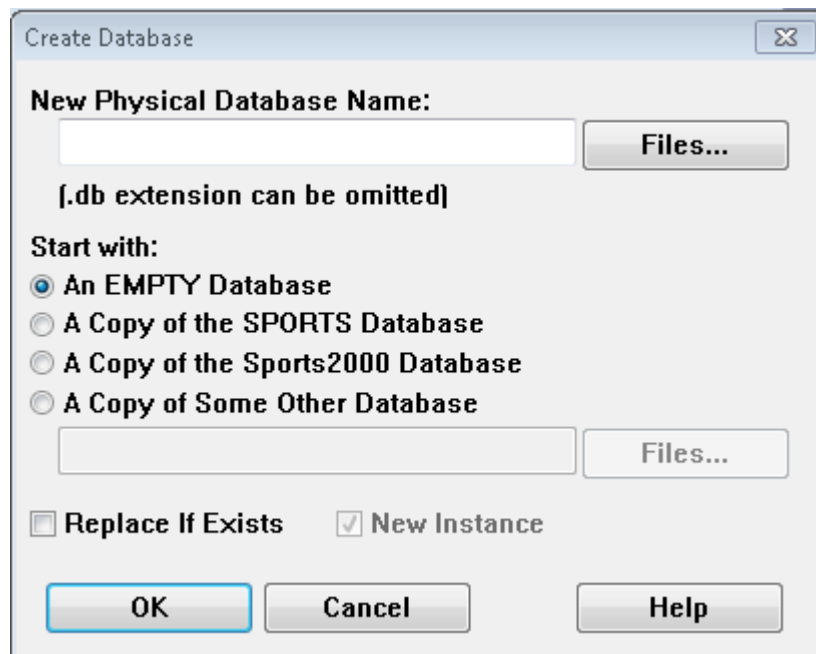
**prodb <nome-do-banco-a-ser-criado> sports**

**prodb <nome-do-banco-a-ser-criado> teste**

Os próximos exemplos serão abordados pela ferramenta de dicionário de dados da versão client do Progress® para facilitar o andamento do aprendizado, mas nada impede que este mesmo assunto seja abordado nas versões CHARACTER ou WEBSPEED.



Logo após clicar no ícone “Databases” no topo da tela, você deve selecionar a opção “Create Database”, botão localizado no rodapé da tela.

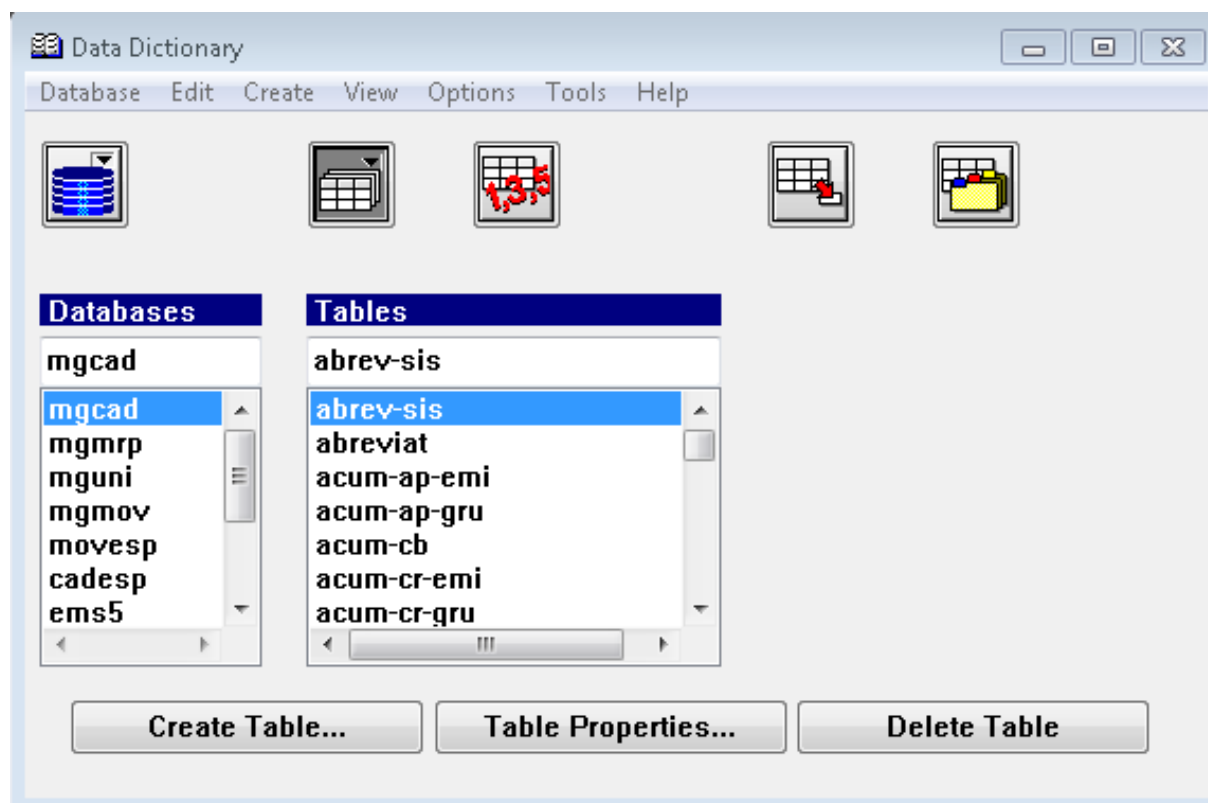


Digite o nome do Banco de Dados que deseja criar, omitindo a extensão (.db)  
Escolha a partir do que deseja criar o banco de dados (empty, sports, other database)  
Marque/Desmarque se deseja colocar o banco a ser criado em cima de um banco já existente com o mesmo nome.  
Pronto, seu banco já foi criado.

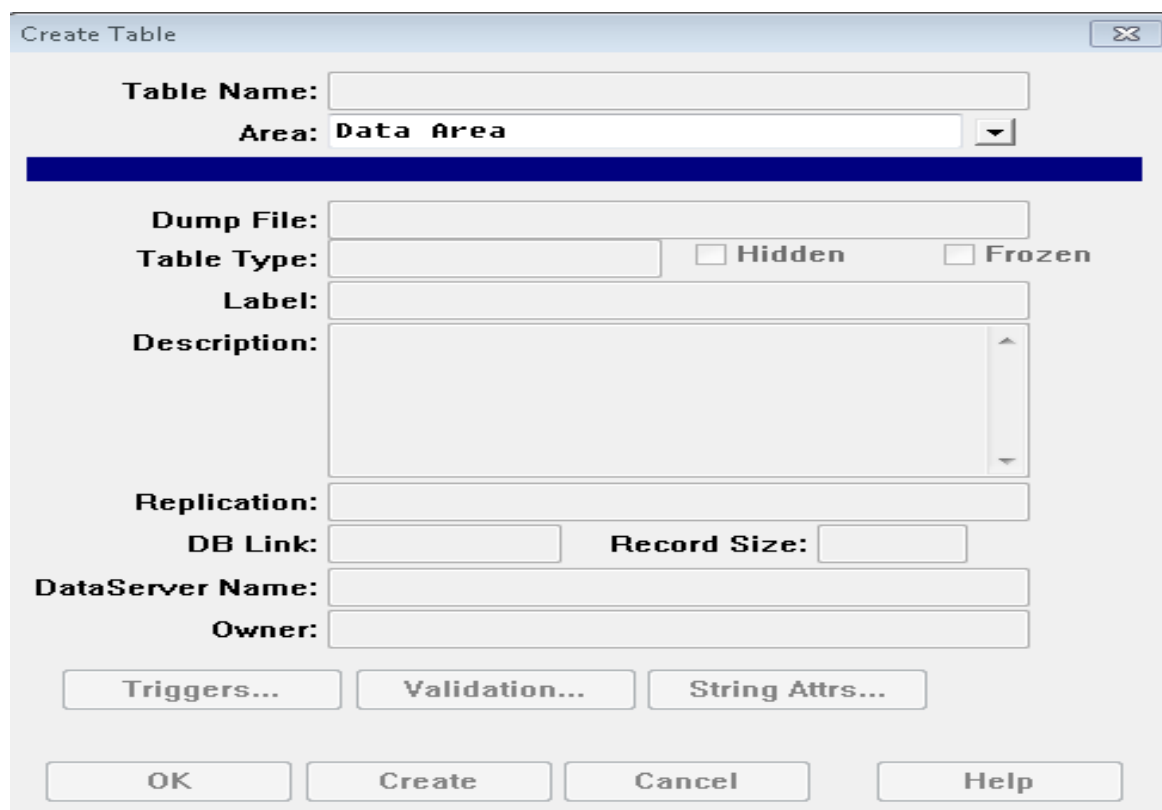
## CRIANDO UMA TABELA

Após criar o Banco de Dados, devemos criar suas estruturas, isto é, as tabelas (arquivos) onde vamos armazenar organizadamente os dados.





Logo após clicar no ícone “Tables” no topo da tela, você deve selecionar a opção “Create Table”, botão localizado no rodapé da tela.



Onde você deve preencher os seguintes campos:

**Table Nome** -Nome da tabela.

**Label** -Descrição abreviada da finalidade da tabela.

**Dump File** -Nome externo da tabela (arquivo.d).

Os demais campos não são os obrigatórios para a criação de uma tabela, segue o detalhamento de alguns deles.

**Área** -Schema Área onde a tabela vai ser criada

**Hidden** -Se é uma tabela oculta no bando de dados

**Description** -Descrição detalhada da finalidade da tabela

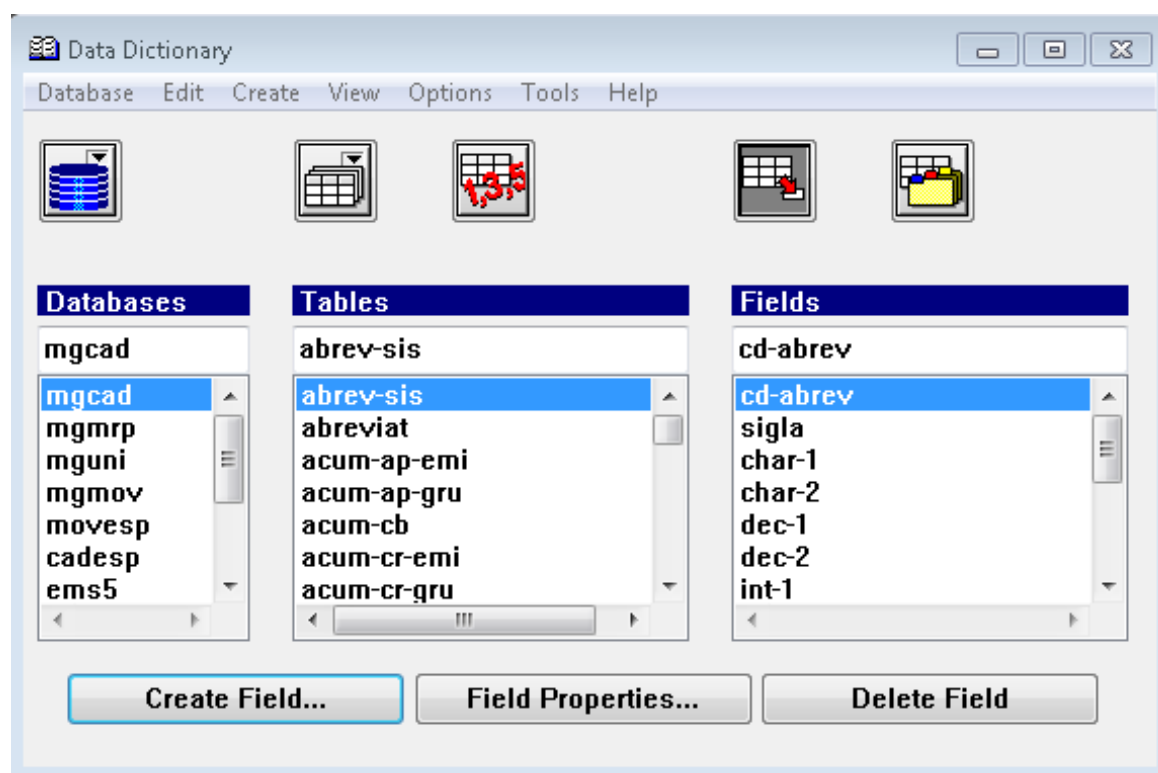
**Replication** -Se a tabela possui Replicação

**Record Size** -Tamanho do registro da tabela

**DataServer Name** -Se a tabela tem um DataServer atrelado

## CRIANDO CAMPOS NA TABELA

Os campos de uma tabela são na verdade, onde cada informação será armazenada de forma organizada, isto é, dentro de uma tabela como exemplo de Clientes, você terá os campos nome do cliente, endereço do cliente, etc. Onde cada campo identifica o tipo e qual dado este armazena.



Logo após clicar no ícone “Fields” no topo da tela, você deve selecionar a opção “Create Field”, botão localizado no rodapé da tela.

Onde você deve preencher os seguintes campos:

**Field Name** -Nome do campo criado.

**Data Type** -Tipo de registro do campo, conforme tipos de dados previstos pelo Progress®.

**Format** -Formato do campo para telas e relatórios, maneira em que se quer que o campo seja apresentado.

**Label** -Nome do campo para usuário, parecera como default na tela.

**Column Label** -Nome campo para usuário em caso de tratar tela ou relatório por colunas.

**Initial Value** -Serve para inicializar um campo com um determinado valor, toda a vez que for criado um novo registro para este campo será inicializado com o valor encontrado no campo inicial.

**Order** -Ordem sequencial de criação e ordenação de campos, seve também para os relatórios de tabelas do banco de dados.

**Decimals** -Indica a quantidade de casas decimais que o campo utiliza, em caso de ser um registro do tipo decimal.

**Description** -Basicamente com a mesma ideia do help, porém este campo está voltado para as pessoas que trabalham com o desenvolvimento de programas e não usuários finais, portanto pode ter informações mais técnicas.

**Help Text**-Campo destinado para colocar uma mensagem que vai servir de help para o usuário quando o mesmo estive fazendo alguma manipulação de dados no campo.

**Mandatory** -Diz que o campo será mandatório, isto é, não aceitará valores nulos, somente valores válidos conforme o formato definido para o campo.

**Case Sensitive** -Identifica se o campo deve fazer distinção de caracteres maiúsculos e minúsculos de uma mesma literal.

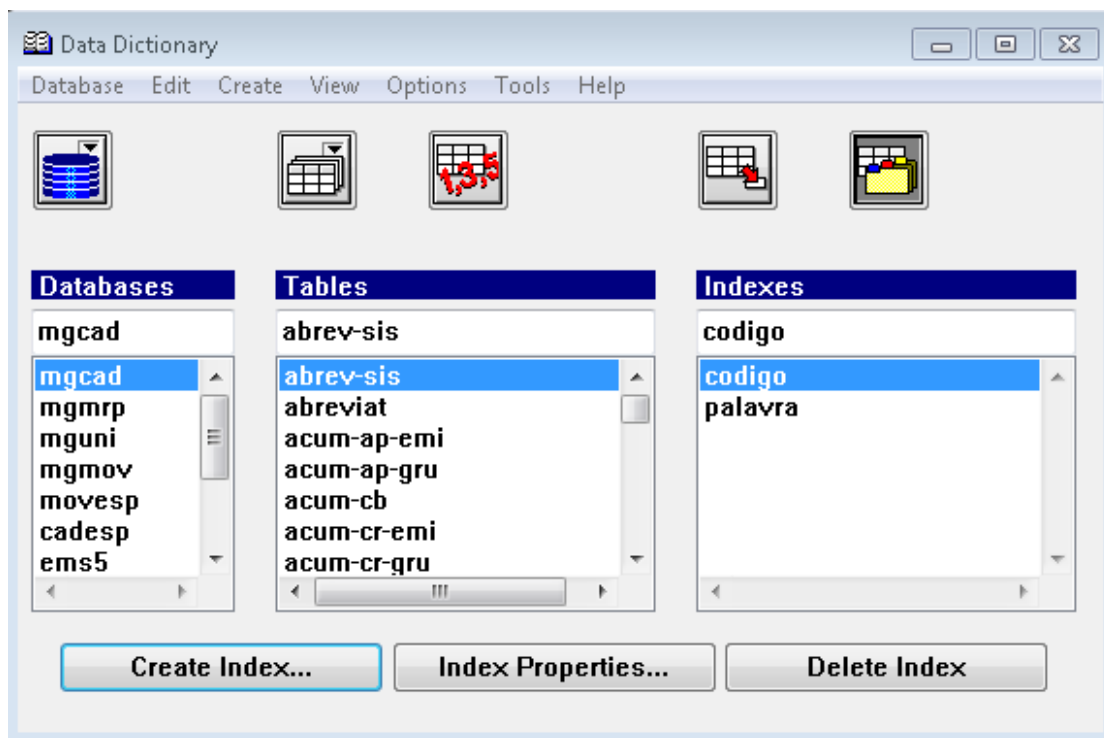
**Extent** -Quantidade de registros (posições) no caso de ser array.

Os demais campos da tela normalmente não são utilizados e são de relevância discutível para a característica de programas que utilizamos.

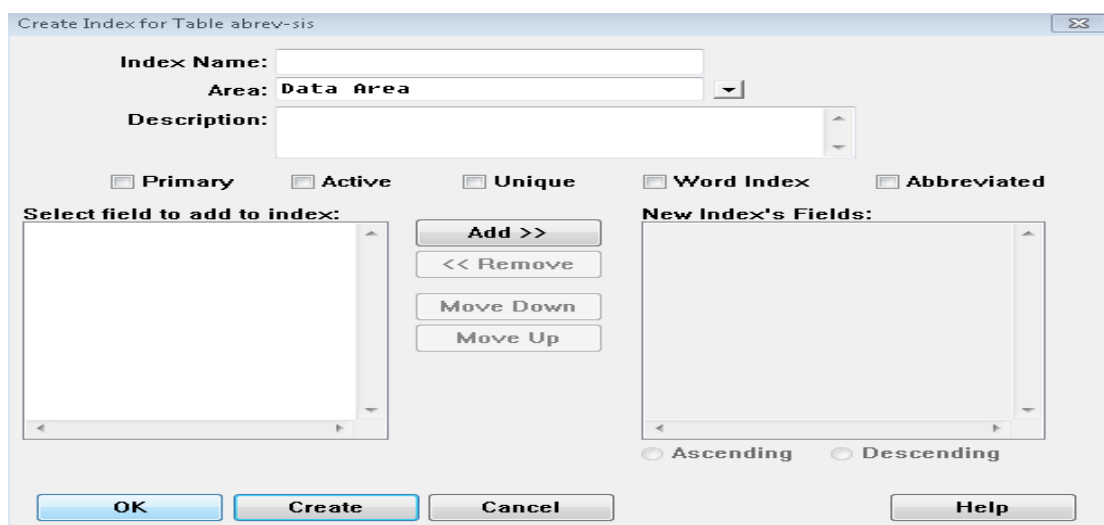
# CRIANDO ÍNDICES

Um índice é definido para uma tabela por vários motivos, dentre eles:

- ✓ Agilizar o processo de pesquisa;
- ✓ Não permitir a existência de duplicidade de dados em campos únicos;
- ✓ Ordenar de forma organizada a massa de dados.



Logo após clicar no ícone "Indexes" no topo da tela, você deve selecionar a opção "Create Index", botão localizado no rodapé da tela.



Onde você deve preencher os seguintes campos:

**Index Name** -Nome do índice criado.

**Área** -Nome do Schema Área onde vai ser criado o índice.

**Description** -Descrição detalhada do sentido da criação do índice.

Definição de alguns conceitos relacionados a índices:

**PRIMARY** -Indica que será o índice principal, padrão.

**ACTIVE** -Indica que o índice está ativo ou não.

**UNIQUE** -Indica que os valores que compõem o índice não podem ser duplicados, portando obrigatoriamente deve ser único.

**WORD INDEX** -Você pode atribuir uma string para ser vinculada a um índice por palavras, para facilitar buscas de palavras em um texto por exemplo.

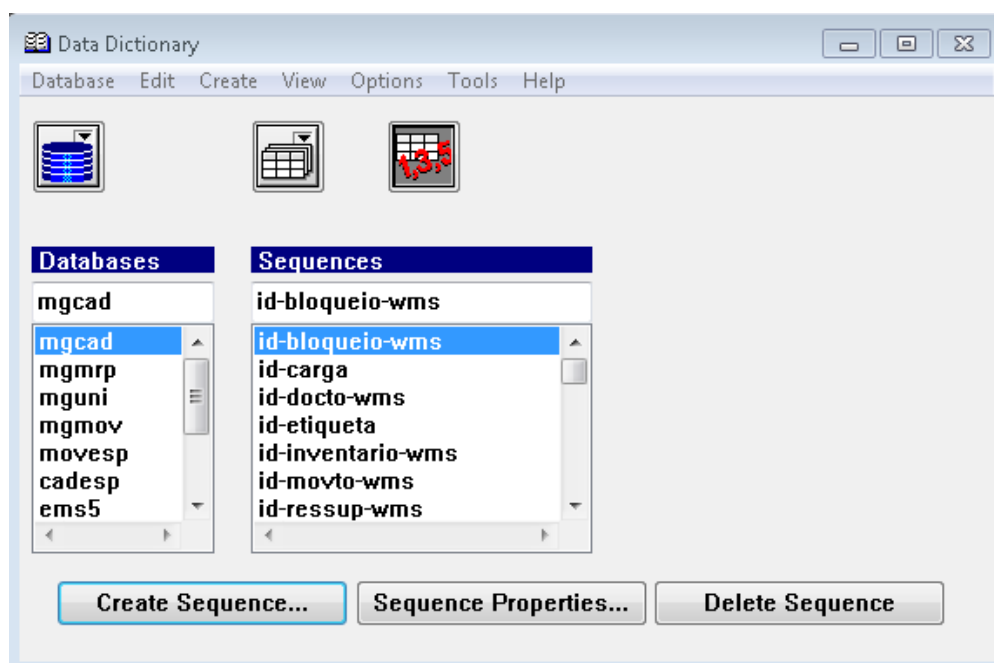
Algumas características relacionadas aos índices:

- ✓ Um índice pode conter vários campos;
- ✓ Cada campo integrante do índice pode ser ordenado em ordem crescente ou decrescente;
- ✓ Não se pode utilizar Array em índices.

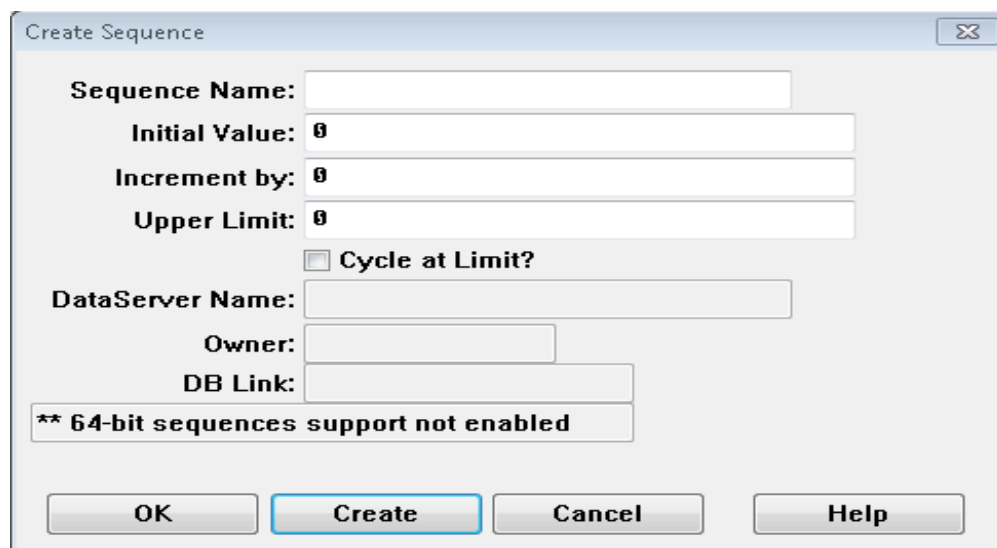
## CRIANDO SEQUENCES

Uma sequência tem a função de auto incremento de código. Geralmente usamos sequence para criar automaticamente o código/valor do campo que é o índice primário e único de uma tabela.

Também podemos usar sequência para controlar numerações importantes do sistema, como número de notas, de títulos, próximo código de cliente, de produto, etc.



Logo após clicar no ícone “Sequences” no topo da tela, você deve selecionar a opção “Create Sequence”, botão localizado no rodapé da tela.



Onde você deve preencher os seguintes campos:

**Sequence Name** -Nome da Sequence.

**Initial Value** -Valor inicial da Sequence.

**Increment by** -De quanto em quanto ela vai ser auto incrementada. Exemplo 10.

**Upper Limit** -Número máximo que pode conter a sequence.

**Cycle at Limit** -Indica se ao chegar no número máximo a sequence deve retornar a incrementar do initial value.

Algumas características relacionadas aos índices:

- ✓ Como boas práticas sugerimos que o nome da sequence seja o mesmo nome do campo que está sendo incrementado, ou algo que faça alusão a este campo, para o melhor entendimento do banco de dados.

## Capítulo 4: RECURSOS DA LINGUAGEM



## DECLARANDO VARIÁVEIS

Como em qualquer outra linguagem tradicional de programação, precisamos identificar os tipos de variáveis que vamos utilizar no programa.

O Progress® possui diversos tipos, dentre eles vamos citar os mais utilizados abaixo relacionados:

### SYNTAX

```
DEFINE { [ [ NEW [ GLOBAL ] ] SHARED ] | [ PRIVATE | PROTECTED | PUBLIC ] [
STATIC ] }
VARIABLE variable-name { { AS [ HANDLE TO ] primitive-type-name | AS [ CLASS ]
{ object-type-name } | LIKE field } [ EXTENT [ constant ] ] } [ BGCOLOR expression ]
[ COLUMN-LABEL label ] [ CONTEXT-HELP-ID expression ] [ DCOLOR expression ]
[ DECIMALS n ] [ DROP-TARGET ] [ FONT expression ] [ FGColor expression ]
[ FORMAT string ] [ INITIAL { constant | { [ constant [ , constant ] ... ] } } ]
[ LABEL string [ , string ] ... ] [ MOUSE-POINTER expression ] [ NO-UNDO ]
```

[ [ NOT ] CASE-SENSITIVE ] [ PFCOLOR *expression* ] { [ *view-as-phrase* ] } { [ *trigger-phrase* ] }

Exemplos:

Tipo	Default	
	Format	Initial
BLOB	<i>Não é exibido diretamente</i>	<i>N/A</i>
CHARACTER	<i>x(8)</i>	<i>branco</i>
CLASS	<i>Através método ToString()</i>	<i>?</i>
CLOB	<i>Não é exibido diretamente</i>	<i>N/A</i>
COM-HANDLE	<i>&gt;&gt;&gt;&gt;&gt;9</i>	<i>?</i>
DATE	<i>99/99/99</i>	<i>?</i>
DATETIME	<i>99/99/9999 HH:MM:SS.SSS</i>	<i>?</i>
DATETIME-TZ	<i>99/99/9999 HH:MM:SS.SSS+HH:MM</i>	<i>?</i>
DECIMAL	<i>-&gt;, &gt;&gt;9.99</i>	<i>0</i>
HANDLE	<i>&gt;&gt;&gt;&gt;&gt;9</i>	<i>?</i>
INT64	<i>-&gt;, &gt;&gt;&gt;, &gt;&gt;9</i>	<i>0</i>
INTEGER	<i>-&gt;, &gt;&gt;&gt;, &gt;&gt;9</i>	<i>0</i>
LOGICAL	<i>yes/no</i>	<i>no</i>
LONGCHAR	<i>Não é exibido diretamente</i>	<i>?</i>
MEMPTR	<i>Não é exibido diretamente</i>	<i>Tamanho Zero</i>
RAW	<i>Não é exibido diretamente</i>	<i>Tamanho Zero</i>
RECID	<i>&gt;&gt;&gt;&gt;&gt;9</i>	<i>?</i>
ROWID	<i>Não é exibido diretamente</i>	<i>?</i>

Posteriormente a termos definido que tipo de dado que nós iremos utilizar é que passamos a fase de definir (dar nomes) para as variáveis de nosso programa.

Recordamos que variável é um lugar alocado na memória do computador, lugar que podemos dar um nome a ele e também manipular o conteúdo guardado dentro dele. Podemos acrescentar ainda que variáveis, como seu nome sugere, além de conter um valor a cada execução, sempre que entrar em uma nova sessão não vai ter conteúdo algum, nós que temos que manipular o programa para alimentá-la e então somente pode utilizá-la para outros fins dentro de nossa lógica.

Dicas Importantes:

- ✓ O bom senso diz que devemos evitar o uso demasiado de variáveis no programa, programas com muitas variáveis tendem a se tornar complexos para manutenção, a pessoa que irá ajustá-lo perderá muito tempo tentando interpretar as variáveis e qual seu conteúdo e finalidade.
- ✓ Outro fato importante é dar nomes que tenham um significado para a variável, evitar nomes que não definam ela corretamente. Devemos dar nomes que identifiquem o tipo da variável ou o seu conteúdo.



- ✓ Também devemos evitar nomes demasiadamente grandes para variáveis, isso dificulta a digitação do código fonte, podendo onerar o seu trabalho com erros de digitação.

Para definirmos uma variável o comando é **DEFINE VARIABLE** ou simplesmente **DEF VAR**, conforme sintaxe abaixo:

Exemplo:

```
DEFINE VARIABLE vteste
DEFINE VARIABLE vteste1
DEFINE VARIABLE vteste3
```

```
AS INTEGER INITIAL 3.
AS CHARACTER FORMAT "x(30)".
AS LOGICAL FORMAT "Sim/Não".
```

## DECLARANDO UM FRAME

FRAME é o nome que se dá a uma definição de um conjunto de variáveis/campos que serão vistos de uma determinada forma, em um determinado lugar. Algo totalmente abstrato.

Quando não declaramos um frame, através das informações que possui, o próprio ABL declara um internamente, pois a linguagem é dependente de um “repositório” para as informações que vão ser mostradas em uma tela, arquivo ou relatório.

Este mesmo FRAME poderá ser identificado de forma local ou shared (compartilhada).

Vale lembrar que este comando apenas define a forma das coisas serem apresentadas, ele por si só, não mostra nada na tela. Para mostrarmos na tela utilizando as definições deste frame deveremos utilizar comandos de interação com a tela, como por exemplo o DISPLAY, UPDATE, PROMPT-FOR, etc...

### SINTAX

---

```
DEFINE { [ [ NEW ] SHARED ] | [ PRIVATE ] } FRAME frame [ form-item ... ]
[ { HEADER | BACKGROUND } head-item ... ] { [ frame-phrase ] }
```

---

Exemplo:

```
DEFINE FRAME f-teste.
```

Desta forma o FRAME foi definido somente com o nome.

Exemplo2:

```
DEFINE FRAME f-teste
customer.cust-num LABEL "Cliente" FORMAT "99999"
customer.name LABEL "Nome" FORMAT "x(20)"
WITH CENTERED ROW 3 SIDE-LABELS.
```

Este exemplo já utilizou alguns campos da tabela customer (tabela do banco Sports, base de dados de treinamento do Progress®) formatou o conteúdo a ser apresentado em cada campo na tela ou arquivo, conforme o definido no dispositivo de saída.

Vejamos agora como ficaria este frame na tela após utilizarmos o comando DISPLAY e o resultado em tela.

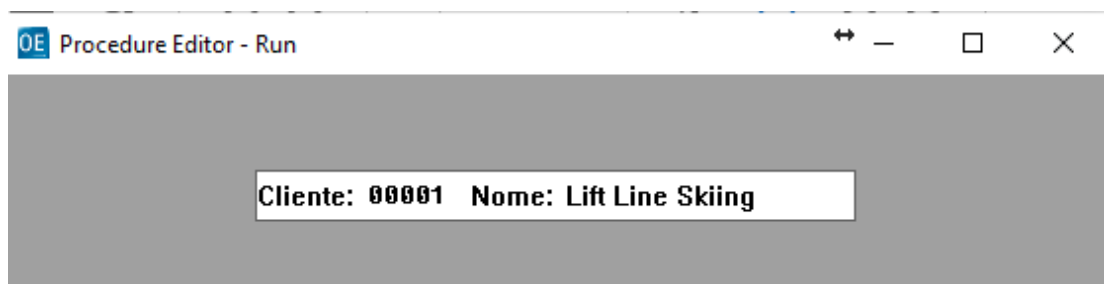
Exemplo3:

```
DEFINE FRAME f-teste
customer.cust-num LABEL "Cliente" FORMAT "99999"
customer.name LABEL "Nome" FORMAT "x(20)"
WITH CENTERED ROW 3 SIDE-LABELS.

FIND FIRST customer NO-LOCK NO-ERROR.

DISPLAY customer.cust-num
customer.name
WITH FRAME f-teste.
```

Resultado:



Verifique que não mais precisamos definir LABELS e nem FORMAT's para os campos no momento de utilizar, no caso no momento do comando display, pois apenas fazemos referência ao frame definido (WITH FRAME f-teste), e ele vai herdar todas as informações que foram definidas no momento da criação do frame.

# OPERADORES LÓGICOS

Operadores lógicos são os artefatos que nos ajudam a compor a lógica de nosso programa, são eles que impõem condições de leitura e mudança de escopo conforme as informações provenientes de um outro programa ou rotina.

Segue os operadores lógicos do Progress®, com exemplos pertinentes a sua utilização.

## AND

Este operador trata das condições de leituras e testes de lógicas, funcionando com a característica de adicionar uma condição, conforme abaixo:

Exemplo:

```
FOR EACH customer WHERE  
  customer.cust-num > 10 AND  
  customer.cust-num < 15  
NO-LOCK:  
  
  DISPLAY customer.cust-num  
  customer.name  
  WITH FRAME f-teste DOWN.  
END.
```

Resultado:

OE Procedure Editor - Run	
Cust-Num	Name
11	Keilailu ja Biljardi
12	Surf Lautaveikkoset
13	Biljardi ja tennis
14	Paris St Germain

## OR

Este operador trata das condições de leituras e testes de lógicas, funcionando com a característica de diversificar a leitura e as condições, conforme abaixo:

Exemplo:

```
FOR EACH customer WHERE
    customer.cust-num = 1 OR
    customer.cust-num = 2
NO-LOCK:

    DISPLAY customer.cust-num
    customer.name
    WITH FRAME f-teste DOWN.
END.
```

Resultado:

OE Procedure Editor - Run	
Cust-Num	Name
1	Lift Line Skiing
2	Urpon Frisbee

## NOT

Este operador tem a função de negar a sentença que se encontra estabelecida nas condições de leituras e testes de lógicas, conforme abaixo:

Exemplo:

```
FOR EACH customer WHERE
    NOT customer.country BEGINS "U" AND
    NOT customer.country BEGINS "F" AND
    NOT customer.country BEGINS "A"
NO-LOCK:

    DISPLAY customer.cust-num
    customer.name
    customer.country
    WITH FRAME f-teste DOWN.
END.
```

Resultado:

OE Procedure Editor - Run		
Cust-Num	Name	Country
8	Game Set Match	Sverige
45	SC Ren Je Rot	Nederland
53	Offside Hockey	Sverige
57	Golf Club Holland	Nederland
60	Hou Hoog die Bal	Nederland
65	Lagt Kort Ligger	Sverige



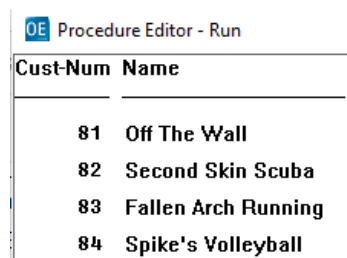
> ou GT

Este operador funciona basicamente como nos padrões matemáticos, seve para testar se uma sentença é maior que outra, conforme abaixo:

Exemplo:

```
FOR EACH customer WHERE  
    customer.cust-num > 80  
    NO-LOCK:  
  
    DISPLAY customer.cust-num  
    customer.name  
    WITH FRAME f-teste DOWN.  
END.
```

Resultado:



Cust-Num	Name
81	Off The Wall
82	Second Skin Scuba
83	Fallen Arch Running
84	Spike's Volleyball

< ou LT

Este operador funciona basicamente como nos padrões matemáticos, seve para testar se uma sentença é menor que outra, conforme exemplo abaixo:

Exemplo:

```
FOR EACH customer WHERE  
    customer.cust-num < 5  
    NO-LOCK:  
  
    DISPLAY customer.cust-num  
    customer.name  
    WITH FRAME f-teste DOWN.  
END.
```

Resultado:

OE Procedure Editor - Run	
Cust-Num	Name
1	Lift Line Skiing
2	Urpon Frisbee
3	Hoops Croquet Co.
4	Go Fishing Ltd

**>=** ou **GE**

Este operador funciona basicamente como nos padrões matemáticos, seve para testar se uma sentença é maior ou igual que outra, conforme exemplo abaixo:

Exemplo:

```
FOR EACH customer WHERE
    customer.cust-num >= 80
    NO-LOCK:

    DISPLAY customer.cust-num
    customer.name
    WITH FRAME f-teste DOWN.
END.
```

Resultado:

OE Procedure Editor - Run	
Cust-Num	Name
80	StickyWicket Cricket
81	Off The Wall
82	Second Skin Scuba
83	Fallen Arch Running
84	Spike's Volleyball

**<=** ou **LE**

Este operador funciona basicamente como nos padrões matemáticos, seve para testar se uma sentença é menor ou igual que outra, conforme abaixo:

Exemplo:

```
FOR EACH customer WHERE
    customer.cust-num <= 5
    NO-LOCK:
```

```

DISPLAY customer.cust-num
customer.name
WITH FRAME f-teste DOWN.
END.

```

Resultado:

OE Procedure Editor - Run

Cust-Num	Name
1	Lift Line Skiing
2	Urpon Frisbee
3	Hoops Croquet Co.
4	Go Fishing Ltd
5	Match Point Tennis

= ou **EQ**

Este operador funciona basicamente como nos padrões matemáticos, seve para testar se uma sentença é igual que outra, conforme abaixo:

Exemplo:

```

FOR EACH customer WHERE
customer.cust-num = 10
NO-LOCK:

DISPLAY customer
WITH FRAME f-teste 2 COL.
END.

```

Resultado:

OE Procedure Editor - Run

<b>Cust-Num:</b> 10	<b>Country:</b> United Kingdom
<b>Name:</b> Just Joggers Limited	<b>Address:</b> Fairwind Trading Est
<b>Address2:</b> Shoe Lane	<b>City:</b> Ramsbottom
<b>State:</b> Lancashire	<b>Postal-Code:</b> BL0 9ND
<b>Contact:</b> George Lacey	<b>Phone:</b> 070 682 2887
<b>Sales-Rep:</b> SLS	<b>Credit-Limit:</b> 22.000
<b>Balance:</b> 16.621,00	<b>Terms:</b> Net30
<b>Discount:</b> 20%	
<b>Comments:</b>	



**<> ou NE**

Este operador seve para testar se uma sentença é diferente que outra, conforme abaixo:

Exemplo:

```
FOR EACH customer WHERE
    customer.cust-num < 5 AND
    customer.cust-num <> 3
NO-LOCK:

DISPLAY customer.cust-num
        customer.name
        WITH FRAME f-teste DOWN.

END.
```

Resultado:

Cust-Num	Name
1	Lift Line Skiing
2	Urpon Frisbee
4	Go Fishing Ltd

Anotações:

This image shows a full page of blank white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page, providing a template for writing or drawing. There are no margins, text, or other markings present.



## Capítulo 5: COMANDOS



## COMANDOS DE REPETIÇÃO

Estes comandos servem para criar um laço de repetição, onde este laço só se quebra quanto satisfazer alguma condição especificada pelo código fonte. Em programas procedurais, são comandos vitais para a geração da lógica.

## REPEAT

### SYNTAX

```
[ label : ] REPEAT [ FOR record [ , record ] ... ] [ preselect-phrase ] [ query-tuning-phrase ]  
[ variable = expression1 TO expression2 [ BY k ] ][ WHILE expression ] [ TRANSACTION  
[ STOP-AFTER expression ] [ on-endkey-phrase ] [ on-error-phrase ] [ on-quit-phrase ]  
[ on-stop-phrase ] [ frame-phrase ] [ catch-block [ catch-block ... ] ] [ finally-block ] :
```

Exemplo:

**DEFINE VARIABLE** vcont **AS INTEGER.**

**REPEAT:**

**ASSIGN** vcont = vcont + 1.

**DISPLAY** "Passou no laço: " vcont **WITH FRAME** f-teste **DOWN.**

**IF** vcont = 5 **THEN**  
**LEAVE.**

**END.**

Resultado :

OE Procedure Editor - Run

	vcont
Passou no laço:	1
Passou no laço:	2
Passou no laço:	3
Passou no laço:	4
Passou no laço:	5

O comando **leave** serve para sair do laço de um repeatn neste exemplo estamos utilizando um contador **vcont** e após mostrarmos na tela o resultado, estamos verificando se o contador possui valor 5, se verdadeiro então quebra o laço.

## FOR

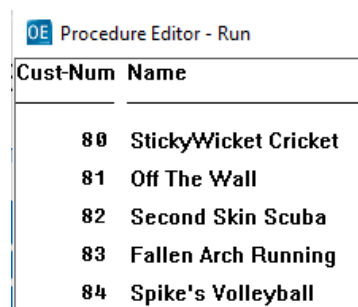
### SYNTAX

```
[ label: ] FOR [ EACH | FIRST | LAST ] record-phrase [ ,
[ EACH | FIRST | LAST ] record-phrase ]... [ query-tuning-phrase ] [ BREAK ]
[ BY expression [ DESCENDING ] | COLLATE ( string , strength [ , collation ] ) [
DESCENDING ] ] ...
[ variable = expression1 TO expression2 [ BY k ] ] [ WHILE expression ] [ TRANSACTION
]
[ STOP-AFTER expression ] [ on-error-phrase ] [ on-endkey-phrase ] [ on-quit-phrase ]
[ on-stop-phrase ] [ frame-phrase ] :
```

Exemplo:

```
FOR EACH customer WHERE  
    customer.cust-num >= 80  
    NO-LOCK:  
  
    DISPLAY customer.cust-num  
    customer.name  
    WITH FRAME f-teste DOWN.  
END.
```

Resultado:



OE Procedure Editor - Run

Cust-Num	Name
80	StickyWicket Cricket
81	Off The Wall
82	Second Skin Scuba
83	Fallen Arch Running
84	Spike's Volleyball

O comando **FOR EACH** é um comando de repetição e ao mesmo tempo um comando de leitura.

Neste exemplo estamos lendo toda a tabela de clientes e mostrando seu conteúdo na tela conforme a seleção que consta na cláusula **WHERE**.

Anotações:

---

---

---

---

---

---

## COMANDOS DE CONDIÇÃO

Estes comandos servem para desvio de código, isto é, conforme a condição especificada o programa desvia por um ou outro caminho respeitando a lógica gerada pelo programador.

# IF .. THEN .. ELSE

## SINTAX

IF *expression* THEN { *block* | *statement* } [ ELSE { *block* | *statement* } ]

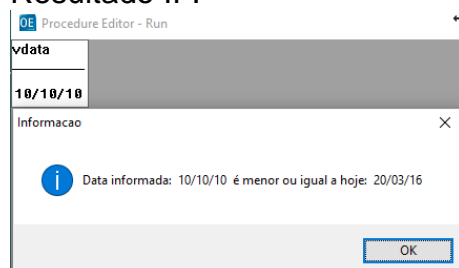
Exemplo:

**DEFINE VARIABLE** vdata **AS DATE**.

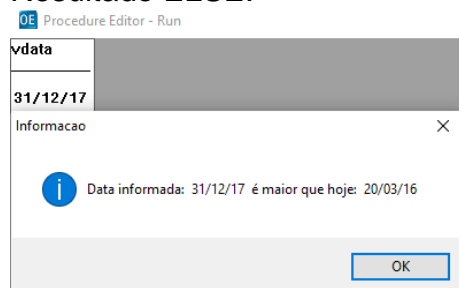
**UPDATE** vdata **WITH FRAME** f-teste.

```
IF vdata > TODAY THEN
  MESSAGE "Data informada: "
    vdata
    " é maior que hoje: "
    TODAY
  VIEW-AS ALERT-BOX INFORMATION.
ELSE
  MESSAGE "Data informada: "
    vdata
    " é menor ou igual a hoje: "
    TODAY
  VIEW-AS ALERT-BOX INFORMATION.
```

Resultado IF:



Resultado ELSE:



## CASE

Comando utilizado quando existe mais de duas condições para a lógica, até pode-se usar o IF encadeando todas as condições, porém o CASE deixa o programa mais legível tanto falando de questões visuais propriamente dita como também para uma possível futura interpretação do código.

### SINTAX

---

*CASE expression:*

```
{WHEN value [OR WHEN value]... THEN {block |statement }}...  
[OTHERWISE {block |statement }]  
END [CASE]
```

Exemplo:

```
DEFINE VARIABLE vopcao AS INTEGER FORMAT "9".
```

```
FORM "1-Inclusao" SKIP
```

```
    "2-Alteracao" SKIP
```

```
    "3-Exclusao" SKIP
```

```
    "4-Consulta" SKIP
```

```
    "5-Browse" SKIP(1)
```

```
    SPACE(05) vopcao LABEL "Digite a Opção"
```

```
    WITH FRAME f-menu SIDE-LABELS THREE-D CENTERED
```

```
        TITLE " Menu de Opções ".
```

```
REPEAT:
```

```
    UPDATE vopcao WITH FRAME f-menu.
```

```
    CASE vopcao:
```

```
        WHEN 1 THEN RUN inclusao.p.
```

```
        WHEN 2 THEN RUN alteracao.p.
```

```
        WHEN 3 THEN RUN exclusao.p.
```

```
        WHEN 4 THEN RUN consulta.p.
```

```
        WHEN 5 THEN RUN browse.p.
```

```
        OTHERWISE MESSAGE "Opção Inválida" VIEW-AS ALERT-BOX ERROR.
```

```
    END CASE.
```

```
END.
```

Resultado:

Menu de Opções	
1-Inclusao	
2-Alteracao	
3-Exclusao	
4-Consulta	
5-Browse	
Digite a Opção: <input type="text"/>	

## COMANDOS DE SELEÇÃO

### BEGINS

Este operador tem a função de mostrar todos os dados onde a sentença em questão iniciar com o valor informado, conforme exemplo abaixo:

### SINTAX

---

*expression1* BEGINS *expression2*

Exemplo:

```

FOR EACH customer WHERE
    customer.name BEGINS "A"
NO-LOCK:

    DISPLAY customer.cust-num
    customer.name
    WITH FRAME f-teste DOWN.
END.
  
```

Resultado:

OE Procedure Editor - Run

Cust-Num	Name
37	ABC Mountain Bikes
7	Aerobics valine KY
18	Antin Metsastysase
73	Auffi Bergausrustung

## MATCHES

Este operador tem a função de mostrar todos os dados onde a sentença em questão conter o valor informado, conforme exemplos abaixo:

### SINTAX

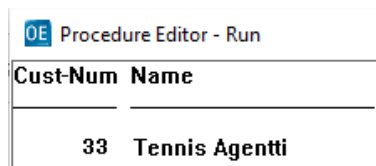
---

*expression MATCHES pattern*

Exemplo no início da sentença:

```
FOR EACH customer WHERE  
    customer.name MATCHES "Tennis*"  
    NO-LOCK:  
  
    DISPLAY customer.cust-num  
    customer.name  
    WITH FRAME f-teste DOWN.  
END.
```

Resultado:

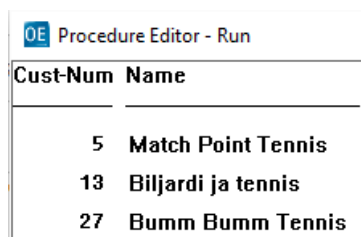


Cust-Num	Name
33	Tennis Agentti

Exemplo no final da sentença:

```
FOR EACH customer WHERE  
    customer.name MATCHES "*Tennis"  
    NO-LOCK:  
  
    DISPLAY customer.cust-num  
    customer.name  
    WITH FRAME f-teste DOWN.  
END.
```

Resultado:



Cust-Num	Name
5	Match Point Tennis
13	Biljardi ja tennis
27	Bumm Bumm Tennis

Exemplo em qualquer parte da sentença:

```
FOR EACH customer WHERE
    customer.name MATCHES "*"Tennis*"
    NO-LOCK:

    DISPLAY customer.cust-num
    customer.name
    WITH FRAME f-teste DOWN.

END.
```

Resultado:

OE Procedure Editor - Run

Cust-Num	Name
5	Match Point Tennis
13	Biljardi ja tennis
27	Bumm Bumm Tennis
33	Tennis Agentti

## LOOKUP

Esta função retorna à posição de um determinado valor em uma lista, caso não encontre nenhuma ocorrência retorna o valor 0, conforme exemplo abaixo:

### SINTAX

LOOKUP ( *expression* , *list* [ , *character* ] )

Exemplo:

```
DEFINE VARIABLE estados AS CHARACTER INITIAL "RS,SC,PR,SP,RJ".
DEFINE VARIABLE uf AS CHARACTER FORMAT "!(2)".
```

REPEAT:

```
UPDATE uf HELP "Informe UF de um estado Brasileiro".
```

```
IF LOOKUP(uf, estados) = 0 THEN
    MESSAGE "Estado informado não está na lista"
    VIEW-AS ALERT-BOX ERROR.
ELSE
    MESSAGE "Estado informado está na posição "
    LOOKUP(uf, estados)
    " da lista"
    VIEW-AS ALERT-BOX INFORMATION.
```

END.





Exemplo:

**DISPLAY** "Olá Amigo !!".

Exemplo2:

**FIND FIRST customer NO-LOCK NO-ERROR.**

**DISPLAY customer.**

Neste exemplo estamos pesquisando o primeiro registro da tabela customer e depois mostrando na tela todos os campos da referida tabela.

## COMANDO MESSAGE

O comando MESSAGE serve para mostrar a informação na tela. Pode apenas mostrar alguma informação como pode optar pela interação do usuário com as opções YES-NO, YES-NO-CANCEL, OK-CANCEL, RETRY-CANCEL ou apenas OK.

Quando utilizada a sintaxe view-as alert-box ele mostra a informação centralizada na tela em formato de uma caixa de texto, porém sem essa opção ele vai mostrar na primeira linha de mensagens disponível no rodapé da tela.

### SYNTAX

---

```
MESSAGE [COLOR color-phrase ] { expression | SKIP [ ( n ) ] } ...
[VIEW-AS ALERT-BOX [alert-type] [BUTTONS button-set] [ TITLE title-string ]] [ { SET |
UPDATE }
      field { AS datatype | LIKE field } [FORMAT string] [AUTO-RETURN]] [IN
WINDOW window]
```

---

Exemplos:

**MESSAGE "Olá Amigo !!" VIEW-AS ALERT-BOX.**  
**MESSAGE "Tudo Bem?" VIEW-AS ALERT-BOX QUESTION BUTTONS YES-NO**  
**UPDATE sresp AS LOGICAL.**  
**MESSAGE "Olá Amigo – 2 !!".**

## COMANDO PUT

O comando PUT é similar ao comando Display, com a diferença que ele não traz consigo a formatação de label ou column-label do campo ou variável, também não se preocupa com espaços entre as variáveis, quebras de linhas, etc... Porém esta versatilidade se enquadra justamente na criação de arquivos e relatórios posicionais, como integrações, EDI, XML e algum outro tipo de lay-out proprietário de alguma aplicação.

Com ele também é possível gerar comandos para quebra de linhas, negrito, itálico, etc... normalmente sequência de caracteres especiais ou ASC para interpretação de impressoras/ECF.

### SYNTAX

---

```
PUT [STREAM stream] [UNFORMATTED] [{expression [FORMAT string] [{AT|TO}
expression]}]
    {SKIP[(expression)]}{SPACE[(expression)]}...
```

Exemplos:

```
PUT "Olá Amigo ".
```

```
PUT UNFORMATTED CHR(27) + "N" + CHR(1).
```

Anotações:

---

---

---

---

---

---

---

## COMANDOS E FUNÇÕES DIVERSOS

### Função ABSOLUTE

Retorna o valor absoluto de um número. Mesmo recurso da matemática.

### SYNTAX

---

```
ABSOLUTE ( n )
```



Exemplos:

**DISPLAY ABSOLUTE**(10.32 – 12.33).

**DISPLAY ABS**(10.32 – 12.33).

**Função ACCUM**

## **SYNTAX**

---

*ACCUM aggregatephrase expression.*

Esta função tem o objetivo de mostrar o valor acumulado de uma sessão de leitura, para estas leituras ainda podemos ter a variação de possibilidades listada abaixo:

**AVERAGE** - Mostra a média de um campo lido no bloco leitura.

Exemplo:

```
FOR EACH customer WHERE
    customer.cust-num = 6 NO-LOCK,
EACH order OF customer NO-LOCK,
EACH order-line OF order WHERE
    order.order-num = 165 NO-LOCK:

DISPLAY order.order-num
    order-line.qty(AVERAGE) WITH FRAME X DOWN.
END.
```

Resultado:

Order-num	Qty
165	65
165	10
165	25
165	57
	-----
	39 AVG



**COUNT** - Mostra o total de registros lidos no bloco de leitura.

Exemplo:

```
FOR EACH customer WHERE
  customer.cust-num = 6 NO-LOCK,
  EACH order OF customer NO-LOCK,
  EACH order-line OF order WHERE
    order.order-num = 165 NO-LOCK:

  DISPLAY order.order-num
    order-line.qty(COUNT) WITH FRAME X DOWN.
END.
```

Resultado:

Order-num Qty

```
-----
165  65
165  10
165  25
165  57
-----
      4 COUNT
```

**MAXIMUM** - Mostra o valor máximo lido no bloco de leitura.

Exemplo:

```
FOR EACH customer WHERE
  customer.cust-num = 6 NO-LOCK,
  EACH order OF customer NO-LOCK,
  EACH order-line OF order WHERE
    order.order-num = 165 NO-LOCK:

  DISPLAY order.order-num
    order-line.qty(MAXIMUM) WITH FRAME X DOWN.
END.
```

Resultado:

Order-num Qty

```
-----
165  65
165  10
165  25
165  57
-----
      65 MAX
```



**MINIMUM** - Mostra o valor mínimo lido no bloco de leitura.

Exemplo:

```
FOR EACH customer WHERE
  customer.cust-num = 6 NO-LOCK,
  EACH order OF customer NO-LOCK,
  EACH order-line OF order WHERE
    order.order-num = 165 NO-LOCK:

  DISPLAY order.order-num
    order-line.qty(MINIMUM) WITH FRAME X DOWN.
END.
```

Resultado:

Order-num Qty

```
-----
165  65
165  10
165  25
165  57
-----
```

10 MIN

**TOTAL** - Mostra o valor total lido no bloco de leitura.

Exemplo:

```
FOR EACH customer WHERE
  customer.cust-num = 6 NO-LOCK,
  EACH order OF customer NO-LOCK,
  EACH order-line OF order WHERE
    order.order-num = 165 NO-LOCK:

  DISPLAY order.order-num
    order-line.qty(TOTAL) WITH FRAME X DOWN.
END.
```

Resultado:

Order-num Qty

```
-----
165  65
165  10
165  25
165  57
-----
```

157 TOTAL



Podemos ainda fazer a combinação de mais de uma situação da função **ACCUM** no mesmo bloco de leitura, veja o exemplo:

```
FOR EACH customer WHERE
    customer.cust-num = 6 NO-LOCK,
EACH order OF customer NO-LOCK,
EACH order-line OF order WHERE
    order.order-num = 165 NO-LOCK:

    DISPLAY order.order-num
        order-line.qty(AVERAGE)
        order-line.qty(COUNT)
        order-line.qty(MINIMUM)
        order-line.qty(MAXIMUM)
        order-line.qty(TOTAL)
WITH FRAME X DOWN.
END.
```

Resultado:

Order-num Qty

```
-----
165   65
165   10
165   25
165   57
-----
157 TOTAL
  4 COUNT
  65 MAX
  10 MIN
  39 AVG
```

As variações SUB-AVERAGE, SUB-COUNT, SUB-MAXIMUM, SUB-MINIMUM e SUB-TOTAL, como seus nomes sugerem servem como sub totais conforme a quebra (by) utilizado no bloco de leitura.

BY break-group - Este serve para dar totais para qualquer variável ou campo de tabela que seja alimentada no bloco de leitura, onde obrigatoriamente tem que ser declarada uma quebra de leitura (break by...).

## Comando ACCUMULATE

### SYNTAX

---

ACCUMULATE { *expression* ( *aggregate-phrase* ) } ...

Este comando faz uso da função ACCUM, a principal diferença entre os dois é bem conceitual, pois na utilização se confundem. O comando permite fazer cálculos, acumular

em memória e usar posteriormente o conteúdo como se fosse uma variável, porém sem precisar declarar uma variável.

Para armazenar o dado:

Accumulate <campo> (<tipo> BY <quebra>)

No exemplo abaixo estamos acumulando a quantidade de itens vendidos (todos os itens) por pedido:

**ACCUMULATE order-line.qty (TOTAL BY order.order-num)**

Para usar o dado armazenado:

Accumulate <tipo> BY <quebra> <campo>

Agora vamos fazer uso do que foi armazenado acima, ou seja, descarregar o conteúdo, na tela, relatório ou até mesmo jogar para uma variável ou campo de tabela.

**ACCUMULATE TOTAL BY order.order-num order-line.qty**

Exemplo completo de um programa usando o comando accumulate.

```
FOR EACH order          NO-LOCK,
  EACH order-line OF order NO-LOCK
  BREAK BY order.order-num:

  ACCUMULATE order-line.qty (TOTAL BY order.order-num).

  IF LAST-OF(order.order-num) THEN
    DO:
      DISPLAY order.order-num LABEL "Pedido"
      ACCUMULATE TOTAL BY order.order-num order-line.qty
      WITH FRAME X DOWN.
  END.
END.
```

## Função ADD-INTERVAL

Adiciona ou subtrai um determinado tempo em um campo do tipo DATE, DATETIME ou DATETIME-TZ, retornando um novo valor para tal considerando a atribuição feita.

## SYNTAX

---

ADD-INTERVAL (*datetime, interval-amount, interval-unit*)



Exemplos:

```
DISPLAY ADD-INTERVAL (TODAY, 2, "days").
DISPLAY ADD-INTERVAL (TODAY, -2, "days").
```

Esta função permite a utilização de vários tipos de unidades de intervalo "internal-unit", são elas:

- Ano = "years"
- Meses = "months"
- Semanas = "weeks"
- Dias = "days"
- Minutos = "minutes"
- Segundos = "seconds"
- Milisegundos = "milliseconds"

## Função AMBIGUOUS

Permite saber se existe mais de um registro compatível com a cláusula WHERE na utilização do comando FIND.

## SYNTAX

---

AMBIGUOUS *record*

Exemplo:

```
DEF VAR cName LIKE Customer.Name LABEL "Nome Cliente".
```

```
REPEAT:
```

```
  SET cName.
```

```
  FIND Customer WHERE
```

```
    Customer.Name BEGINS cName
```

```
  NO-LOCK NO-ERROR.
```

```
  IF AVAILABLE Customer THEN
```

```
    DISPLAY Customer.Cust-Num
```

```
      Customer.Address
```

```
      Customer.City
```

```
      Customer.State
```

```
    WITH FRAME X.
```

```
  ELSE
```

```
    IF AMBIGUOUS Customer THEN
```

```
      MESSAGE "Existe mais de um registro para a seleção".
```

```
    ELSE
```

```
      MESSAGE "Não Existe registro encontrado para a seleção".
```

```
END.
```

## Comando APPLY

Comando para aplicar um determinado evento no programa, pode ser um evento gerado pelo usuário no programa ou já previsto para ocorrer durante a lógica sem a intervenção do usuário.

### SYNTAX

---

APPLY *event* [ TO *widget-phrase* ]

Exemplo:

```
DEFINE BUTTON order-but LABEL "Pedido"
TRIGGERS:
ON CHOOSE
DO:
    FIND FIRST Order OF Customer NO-LOCK NO-ERROR.

    IF AVAILABLE Order THEN
        UPDATE Order
            WITH FRAME upd-dlg VIEW-AS DIALOG-BOX
            TITLE "Update Order" SIDE-LABELS.
    END.
END TRIGGERS.
```

```
FORM order-but Customer.Name WITH FRAME x.
```

```
ON F10 OF Customer.Name
DO:
    APPLY "CHOOSE" TO order-but IN FRAME x.
END.
```

```
FIND FIRST Customer NO-LOCK NO-ERROR.
DISPLAY order-but Customer.Name WITH FRAME x.
ENABLE ALL WITH FRAME x.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

## Função ASC

Função que converte uma expressão de caracteres que representa em um único character ASCII correspondente. O valor retornado é um dado do tipo base decimal.

### SYNTAX

---

ASC ( *expression* [ , *target-codepage* [ , *source-codepage* ] ] )

Exemplo:

```
DISPLAY ASC(STRING(1)).
```

Resultado:

49

Decimal	Binário	Hexadecimal	Referência
49	110001	31	1

*\*Informações retiradas da tabela ASCII*

Exemplo2: Tirar <enter> no final do campo

**FOR EACH** \_file:

```
IF ASC(SUBSTR(_dump-name,length(_dump-name),1)) = 10 AND
ASC(SUBSTR(_dump-name,(length(_dump-name) - 1),1)) = 13 THEN
ASSIGN _dump-name = SUBSTR(_dump-name,1,length(_dump-name) - 2).
```

**END.**

## Comando ASSIGN

Comando de atribuição do Progress®, serve para atribuir valores a qualquer tipo de variável, campos de tabelas, temp-tables, etc...

## SYNTAX

```
ASSIGN {[ [ INPUT ] FRAME frame | BROWSE browse ] { field [ = expression ] } [ WHEN expression ] } ... [ NO-ERROR ]
```

```
ASSIGN { record [ EXCEPT field ... ] } [ NO-ERROR ]
```

Exemplo:

**ASSIGN** wcontador = wcontador + 1.

Este comando ainda pode ser suprimido se o programador quiser fazer atribuições individuais para cada variável ou campo.

Exemplo2:

**wcontador = wcontador + 1.**

Porém se considerarmos que cada ponto final é um comando e irá executar processos em memória, disco e cpu, sempre que possível usar um único ASSIGN para várias sentenças.

Exemplo3:

**ASSIGN** wcontador = wcontador + 1



```
var1    = 10  
var2    = 3.
```

### Função AVAILABLE ou AVAIL

Comando que testa a existência de um determinado registro que foi feita uma tentativa de leitura, o Progress® trata esta situação por default, somente utilizamos este comando quando queremos interferir na mensagem para o usuário ou desviar a condição de leitura, que é altamente recomendável, pois o retorno do Progress® é o erro 91: \*\* No <file-name> record is available. (91).

O que nos permite fazer esta interferência no comportamento da linguagem e “tratar o erro” é a cláusula NO-ERROR no comando FIND.

### SYNTAX

---

AVAILABLE *record*  
AVAIL *record*

Exemplo com uso de operador conjuntamente:

```
FIND FIRST customer WHERE  
customer.cust-num = 1  
NO-LOCK NO-ERROR.
```

```
IF NOT AVAILABLE customer THEN  
MESSAGE “ERRO: Cliente não Cadastrado”  
VIEW-AS ALERT-BOX ERROR.
```

Exemplo sem uso de operador conjuntamente:

```
FIND FIRST customer WHERE  
customer.cust-num = 1  
NO-LOCK NO-ERROR.
```

```
IF AVAIL customer THEN  
MESSAGE “Cliente: ”  
customer.cust-num  
“ _ ”  
customer.name  
VIEW-AS ALERT-BOX ERROR.
```

### Comando BELL

Este comando emite um sinal sonoro no computador, serve para dar avisos ou chamar a atenção do usuário. Vem nativo em alguns comandos como o view-as alert-box **ERROR**.

## SYNTAX

---

BELL

Exemplo:

**DEF VAR** cont **AS INTEGER**.

**DO** cont = **1 TO 10**:

**BELL**.

**PAUSE 1**.

**END**.

## Comando BUFFER-COPY

Este comando permite copiar o conteúdo inteiro de um buffer, tabela ou temp-table para outro destino, desde que este destino tenha os mesmos campos. Assim sendo evita ter que utilizar o comando assign campo a campo.

## SYNTAX

---

**BUFFER-COPY** *source* [ { **EXCEPT** | **USING** } *field* ... ] **TO** *target* [ **ASSIGN** *assign-expression* ... ] [ **NO-LOBS** ] [ **NO-ERROR** ]

Exemplo:

**BUFFER-COPY** bf-customer **TO** customer.

Exemplo2:

**BUFFER-COPY** bf-customer **EXCEPT** bf-customer.cust-num  
    **TO** customer.

## Função CAN-FIND

Retorna se a existência do registro lido é verdadeira ou falsa. Função bastante utilizada na validação de conteúdo de campos pedidos na tela.

Outro ponto positivo desta função é que trafega apenas a informação se existe ou não existe, com isso não carrega desnecessariamente todos os campos da referida tabela do disco para a shared memory. É um ponto de atenção que pode ser explorado quando se procura melhorar performance de um programa.

## SYNTAX

---

**CAN-FIND** ( [ **FIRST** | **LAST** ] *record* [ *constant* ] [ **OF** *table* ] [ **WHERE** *expression* ] [ **USE-INDEX** *index* ] [ **USING** [ **FRAME** *frame* ] *field* [ **AND** [ **FRAME** *frame* ] *field* ] ... ] [ **SHARE-LOCK** | **NO-LOCK** ] [ **NO-WAIT** ] [ **NO-PREFETCH** ] )

Exemplo com uso de operador conjuntamente:

```
IF NOT CAN-FIND(FIRST customer WHERE
                customer.cust-num = 1) THEN
MESSAGE "ERRO: Não Existe o Registro".
```

Exemplo sem uso de operador conjuntamente:

```
IF CAN-FIND(FIRST customer WHERE
            customer.cust-num = 1) THEN
MESSAGE "Existe o Registro".
```

## CAPS

Comando que passa o valor de uma variável do tipo character para maiúscula (caixa alta), no caso de minúsculo para maiúsculo.

O comando que faz o sentido inverso é o LC (lower case)

## SYNTAX

---

CAPS ( *expression* )

Exemplo:

```
DISPLAY CAPS ("a").
```

Retorno:

A

## Comando CHOOSE

Comando que permite navegar em várias situações de uma lista, muito utilizado para montagem de menus de opções de sistemas e de programas com várias funcionalidades. Ainda é um comando remanescente da versão 4 do Progress®.

## SYNTAX

---

```
CHOOSE{ { ROW field [ HELP char-constant ] } | { FIELD { field [ HELP char-constant ] }
... } }
[ AUTO-RETURN ] [ COLOR color-phrase ] [ GO-ON ( key-label ... ) ] [ KEYS char-variable ]
[ NO-ERROR ] [ PAUSE expression ] { [ frame-phrase ] }
```

Exemplo:

```
DEF VAR wopcao AS CHAR FORMAT "x(14)" EXTENT 5 INITIAL
[" Inclusao ",
```

```
" Alteracao ",
" Exclusao ",
" Consulta ",
" Relatorio "].
```

```
FORM SPACE(01) /* Form Choose */
wopcao[1] FORMAT "x(10)" SPACE(06)
wopcao[2] FORMAT "x(11)" SPACE(06)
wopcao[3] FORMAT "x(10)" SPACE(06)
wopcao[4] FORMAT "x(10)" SPACE(06)
wopcao[5] FORMAT "x(11)"
WITH FRAME f-opcoes NO-LABELS ROW 4 WIDTH 80.
```

```
REPEAT:
MESSAGE "Escolha a Opcao Desejada".
```

```
DISPLAY wopcao WITH FRAME f-opcoes.
CHOOSE FIELD wopcao AUTO-RETURN WITH FRAME f-opcoes.
```

```
IF FRAME-INDEX = 1 THEN
MESSAGE "Entrou na Opção de Inclusão".
```

```
IF FRAME-INDEX = 2 THEN
MESSAGE "Entrou na Opção de Alteração".
```

```
IF FRAME-INDEX = 3 THEN
MESSAGE "Entrou na Opção de Exclusão".
```

```
IF FRAME-INDEX = 4 THEN
MESSAGE "Entrou na Opção de Consulta".
```

```
IF FRAME-INDEX = 5 THEN
MESSAGE "Entrou na Opção de Relatório".
END.
```

## Função CHR

Converte um valor inteiro em seu correspondente character respeitando o mapa de caracteres utilizado.

## SYNTAX

---

CHR ( *expression* [ , *target-codepage* [ , *source-codepage* ] ] )

Exemplo:

```
DEF VAR cont AS INTEGER.
DEF VAR letra AS CHARACTER FORMAT "X(1)" EXTENT 26.
```

```
DO cont = 1 TO 26:
  ASSIGN letra[cont] = CHR((ASC("A")) - 1 + cont).
END.
```

```
DISPLAY SKIP(1)
  letra WITH 2 COLUMNS NO-LABELS
  TITLE "A L F A B E T O" CENTERED ROW 4.
```

## Comando CLEAR

Comando que limpa todo o conteúdo de variáveis e campos de um determinado frame.

### SYNTAX

---

```
CLEAR [ FRAME frame ] [ ALL ] [ NO-PAUSE ]
```

Exemplo:

```
CLEAR FRAME teste ALL.
```

## Comando COLOR

Comando para trocar a cor de uma determinada variável ou conteúdo, conforme definição ou regra de negócio. Utilizada para demonstrar valores expressivos, em forma de destaque.

### SYNTAX

---

```
COLOR [ DISPLAY ] color-phrase [ PROMPT color-phrase ] { field ... } { [ frame-phrase ] }
```

Exemplo:

```
DEF VAR hilite AS CHARACTER EXTENT 3.
DEF VAR loop AS INTEGER.
```

```
ASSIGN hilite[1] = "NORMAL"
  hilite[2] = "INPUT"
  hilite[3] = "MESSAGES".
```

```
REPEAT WHILE loop <= 10:
```

```
  FORM bar AS CHARACTER WITH ROW(RANDOM(3,17))
    COLUMN(RANDOM(5,50)) NO-BOX NO-LABELS
  FRAME semposicao.
```

```
  COLOR DISPLAY VALUE(hilite[RANDOM(1,3)]) bar
  WITH FRAME semposicao.
```

```
  DISPLAY FILL(" ",RANDOM(1,8)) @ bar
  WITH FRAME semposicao.
```





**PAUSE 1 NO-MESSAGE.**  
**HIDE FRAME** semposicao **NO-PAUSE.**

**ASSIGN** loop = loop + 1.  
**END.**

## Comando CREATE

Comando para criação de registros em buffers, tabelas ou temp-tables.

### SYNTAX

---

CREATE Record [ USING { ROWID ( nrow ) | RECID (nrec) } ] [ NO-ERROR ]

Exemplo:

**CREATE** customer.

## Função CURRENT-VALUE

Esta função mostra o valor atual de uma sequence.

### SYNTAX

---

CURRENT-VALUE ( sequence [ , logical-dbname ] )

Exemplo:

**DISPLAY CURRENT-VALUE**(next-cust-num).

## Função DATE

Converte uma string, ou seqüência de string em formato data do Progress®, conforme exemplo:

### SYNTAX

---

DATE ( month , day , year )  
DATE ( string )  
DATE ( integer-expression )  
DATE ( datetime-expression )

Exemplo:

**DEFINE VARIABLE** vano **AS INTEGER INITIAL** "2015".  
**DEFINE VARIABLE** vmes **AS INTEGER INITIAL** "10".



```
DEFINE VARIABLE vdia AS INTEGER INITIAL "03".  
DEFINE VARIABLE vdata AS DATE.
```

```
ASSIGN vdata = DATE(vmes,vdia,vano).
```

```
DISPLAY vdata.
```

### Função DATETIME

Esta função retorna o dia, hora e segundos do equipamento no momento de sua execução.

#### SYNTAX

---

```
DATETIME (date-exp [, mtime-exp ] )  
DATETIME ( string )  
DATETIME (month, day, year, hours, minutes [,seconds [, milliseconds ] ] )
```

Exemplo:

```
DEFINE VARIABLE vdatahora AS DATETIME.
```

```
ASSIGN vdatahora = DATETIME(TODAY, MTIME).  
DISPLAY vdatahora.
```

### Função DAY

Retorna o dia da data informada.

#### SYNTAX

---

```
DAY ( date )  
DAY ( datetime-expression )
```

Exemplo:

```
DISPLAY DAY(10/03/2016).
```

### Função DBNAME

Esta função retorna o nome do banco de dados conectado no momento (banco selecionado).

#### SYNTAX

---

```
DBNAME
```

Exemplo:

```
DISPLAY DBNAME FORMAT "x(30)".
```

## Função DECIMAL

Retorna o valor decimal de uma string ou expressão.

### SYNTAX

---

DECIMAL (expressão)

Exemplo:

**DEFINE VARIABLE** valor **AS DECIMAL**.

**ASSIGN** valor = **DECIMAL**(STRING("10,15")).

*/\* ou \*/*

**ASSIGN** valor = **DEC**(STRING("10,15")).

## Função ENCODE

Este comando tem a função de codificar uma determinada string, ou seja, ele coloca outros valores no campo, que não permite a interpretação de seu conteúdo a olho nu. Algo semelhante a uma criptografia, porém em um padrão Progress®.

### SYNTAX

---

ENCONDE (expressão)

Exemplo:

**DEFINE VARIABLE** senha **AS CHARACTER FORMAT** "x(16)".

**DEFINE VARIABLE** usuario **AS CHARACTER FORMAT** "x(12)".

**DEFINE VARIABLE** resultado **AS CHARACTER FORMAT** "x(16)".

**SET** usuario **LABEL** "Usuario"

Senha **LABEL** "Senha" **BLANK WITH CENTERED SIDE-LABELS**.

**ASSIGN** resultado = **ENCODE**(senha).

**DISPLAY** resultado **LABEL** "Senha Codificada".

## Função ETIME

Esta função retorna o tempo em milissegundos, este tempo é inicializado a cada sessão aberta, portanto ele nunca repete na mesma sessão.

### SYNTAX

---

ETIME [ ( logical ) ]

Exemplo:

**DISPLAY ETIME.**

### Função FILL

Monta a repetição de um character tantas vezes quantas definidas e no formato também estabelecidos, utilizado basicamente para montar linhas de divisão de sessões em relatórios.

#### SYNTAX

---

FILL ( expressão, repetição )

Exemplo:

**DISPLAY FILL**("-",080) **FORMAT** "x(80)".

### Comando HIDE

Comando para esconder um campo ou um frame, necessário quando utilizamos diversos frames sobrepostos em uma mesma aplicação.

#### SYNTAX

---

HIDE [ STREAM stream ] [ widget-phrase | MESSAGE | ALL ] [ NO-PAUSE ] [ IN WINDOW window ]

Exemplo:

**HIDE FRAME** teste.

### Função INTEGER

Função que converte uma string para o formato inteiro do Progress®.

#### SYNTAX

---

INTEGER (expressão)

Exemplos:

**INTEGER**(**STRING**("2")).  
**INT**(**STRING**("2")).

## Função KBLABEL

Função que retorna o label da função do teclado.

### SYNTAX

---

KBLABEL ( *key-function* )

Exemplo:

**DISPLAY KBLABEL("GO").**

Resultado:

**F2**

## Função KEYCODE

Função que retorna o código da função executada via teclado.

### SYNTAX

---

KEYCODE ( *key-label* )

Exemplo:

**DISPLAY KEYCODE("F2").**

Resultado:

**302**

## Função KEYFUNCTION

Função que converte o código informado, na função de teclado correspondente. Processo inverso da KEYCODE.

### SYNTAX

---

KEYFUNCTION ( *expression* )

Exemplo:

**DISPLAY KEYFUNCTION(302).**

Resultado:

**GO**

## Função LASTKEY

Retorna o código da última função executada via teclado.

### SYNTAX

---

LASTKEY

Exemplo:

```
IF LASTKEY = KEYCODE("CURSOR-DOWN") THEN  
    MESSAGE "Teclou Setinha para cima" VIEW-AS ALERT-BOX.
```

## Função LC

Lower Case retorna o conteúdo de uma string para letras minúsculas, inverso do comando CAPS.

### SYNTAX

---

LC ( *expression* )

Exemplo:

```
DISPLAY STRING("AA").
```

Resultado:

aa

## Comando LEAVE

Comando para sair de um laço de repetição conforme uma condição escolhida pelo desenvolvedor.

### SYNTAX

---

LEAVE [ *label* ]

Exemplo:

```
FOR EACH customer NO-LOCK:
```

```
    DISPLAY customer.cust-num  
        customer.name  
WITH FRAME X DOWN.
```

```
    IF customer.cust-num = 3 THEN  
        LEAVE.  
END.
```

Resultado:

OE Procedure Editor - Run

Cust-Num	Name
1	Lift Line Skiing
2	Urpon Frisbee
3	Hoops Croquet Co.

## Função LEFT-TRIM

Comando para tirar espaços em branco localizados a esquerda de uma string.

### SYNTAX

LEFT-TRIM ( *expression* [ , *trim-chars* ] )

Exemplo:

```
DEFINE VARIABLE vteste AS CHARACTER INITIAL "      xxxx".
DISPLAY vteste FORMAT "x(20)" SKIP. /* Forma Normal */
DISPLAY LEFT-TRIM(vteste).          /* Sem espaços esquerda */
```

Resultado:

OE Procedure Editor - Run

teste
xxxx
xxxx

## Função LENGTH

Comando que retorna à quantidade de caracteres utilizados em uma string.

### SYNTAX

LENGTH ( *variable* ) = *expression*

Exemplo:

```
DEFINE VARIABLE vteste AS CHARACTER INITIAL "xxxx".
DISPLAY LENGTH(vteste).
```

Resultado:

## Função MONTH

Retorna o mês da data informada.

### SYNTAX

---

MONTH ( *date* )

Exemplo:

**DISPLAY MONTH(10/01/2016).**

## Comando NEXT

Executa o retorno para o princípio do laço de repetição da lógica.

### SYNTAX

---

NEXT [ *label* ]

Exemplo:

**DEFINE VARIABLE VCONT AS INTEGER.**

**DO vcont = 1 TO 5:**

**IF vcont = 2 OR  
vcont = 4 THEN  
NEXT.**

**DISPLAY "Passou no laço: " vcont  
WITH FRAME f-dados DOWN.**

**DOWN WITH FRAME f-dados.  
END.**

Resultado:

OE Procedure Editor - Run	
	VCONT
Passou no laço:	1
Passou no laço:	3
Passou no laço:	5

## Comando PAGE

Executa o salto de página em um relatório. Não possui parâmetros é aplicada em sua forma pura.





## SYNTAX

---

PAGE [ STREAM *stream* | STREAM-HANDLE *handle* ]

Exemplo:

**DEFINE VARIABLE** vestido **AS CHARACTER NO-UNDO.**

**OUTPUT TO PRINTER.**

**FOR EACH** Customer  
**BY** Customer.State  
**NO-LOCK:**

**IF** Customer.State <> vestido **THEN**

**DO:**

**IF** vestido <> "" **THEN PAGE.**

**ASSIGN** vestido = Customer.State.  
**END.**

**DISPLAY** Customer.Cust-Num

Customer.Name

Customer.Address

Customer.City

Customer.State.

**END.**

## Função PAGE-NUMBER

Retorna o número corrente da página de um relatório, serve para fazer exatamente a paginação (contador de páginas).

## SYNTAX

---

PAGE-NUMBER [ ( *stream* | STREAM-HANDLE *handle* ) ]

Exemplo:

**OUTPUT TO** c:\temp\pagina.txt **PAGED.**

**FOR EACH** Customer **NO-LOCK:**

**FORM HEADER**

"Relatorio de Clientes" AT 30

"Page:" AT 60 **PAGE-NUMBER FORMAT** "zz9" **SKIP(1).**

**DISPLAY** Customer.Cust-Num **COLUMN-LABEL** "Codigo"

Customer.Name **COLUMN-LABEL** "Nome"

Customer.City **COLUMN-LABEL** "Cidade"

Customer.State **COLUMN-LABEL** "Estado".



END.

Resultado:

**Relatório de Clientes**

**Page: 1**

<b>Codigo Nome</b>	<b>Cidade</b>	<b>Estado</b>
1 Lift Line Skiing	Boston	MA
2 Urpon Frisbee	Valkeala	Uusimaa
3 Hoops Croquet Co.	Hingham	MA
4 Go Fishing Ltd	Harrow	Middlesex

.....continua

### Função PAGE-SIZE

Retorna o número de linhas de uma página, serve para testar se o próximo conteúdo a ser exibido cabe na página em questão.

### SYNTAX

PAGE-SIZE [ ( *stream* | STREAM-HANDLE *handle* ) ]

Exemplo:

OUTPUT TO PRINTER.

FOR EACH Customer  
BREAK BY Customer.cust-num  
NO-LOCK:

DISPLAY Customer.cust-num  
Customer.Name.

IF LAST-OF(Customer.cust-num) THEN  
DO:  
IF LINE-COUNTER + 4 > PAGE-SIZE THEN  
PAGE.  
ELSE  
DOWN 1.  
END.  
END.

OUTPUT CLOSE.

## Comando PAUSE

Para o processo, até que seja pressionada a tecla barra de espaços. Pode também vincular-se o tempo de parada desejado, incluído após o comando um número (em segundos).

### SYNTAX

---

PAUSE [ *n* ] [ BEFORE-HIDE ] [ MESSAGE *message* | NO-MESSAGE ] [ IN WINDOW *window* ]

Exemplo:

**PAUSE 10.**

## Comando RELEASE

Libera o registro lido com exclusive-lock, tem a função de evitar lock entre usuários.

### SYNTAX

---

RELEASE *record* [ NO-ERROR ]

Exemplo:

**DEFINE VARIABLE vmetodo AS LOGICAL NO-UNDO.**  
**DEFINE VARIABLE vcli-atual AS ROWID NO-UNDO.**

**DEFINE BUTTON bt-cli-altera LABEL "Altera Cliente".**  
**DEFINE BUTTON bt-sai-aplicacao LABEL "Sair".**

**DEFINE QUERY q-cliente FOR Customer.**

**DEFINE BROWSE b-cliente QUERY q-cliente**  
**DISPLAY Customer.Cust-Num COLUMN-LABEL "Código"**  
**Customer.Name COLUMN-LABEL "Nome"**  
**WITH 10 DOWN.**

**FORM bt-cli-altera**  
**bt-sai-aplicacao SKIP(1)**  
**b-cliente**  
**WITH FRAME f-opcoes.**

**FORM Customer EXCEPT Customer.Comments**  
**WITH FRAME f-cli-atual COLUMN 32 1 COL WIDTH 49.**

**OPEN QUERY q-cliente FOR EACH Customer.**

**ON VALUE-CHANGED OF b-cliente**



```
DO:
  DISPLAY Customer EXCEPT Customer.Comments
  WITH FRAME f-cli-atual SIDE-LABELS.
```

```
  ASSIGN vcli-atual = ROWID(Customer).
END.
```

```
ON CHOOSE OF bt-cli-altera
```

```
DO:                                     /* TRANSACTION */
  FIND Customer WHERE
    ROWID(Customer) = vcli-atual
  EXCLUSIVE-LOCK NO-ERROR.
```

```
  UPDATE Customer
    WITH FRAME f-bt-cli-altera
    TITLE " Alteração Cliente "
    ROW 13 2 COL OVERLAY.
```

```
  ASSIGN vmetodo = b-cliente:REFRESH().
```

```
  DISPLAY Customer EXCEPT Customer.Comments
  WITH FRAME f-cli-atual SIDE-LABELS.
```

```
  RELEASE Customer.
END.
```

```
ENABLE ALL WITH FRAME f-opcoes.
APPLY "VALUE-CHANGED" TO b-cliente.
PAUSE 0 BEFORE-HIDE.
WAIT-FOR CHOOSE    OF bt-sai-aplicacao OR
  WINDOW-CLOSE OF DEFAULT-WINDOW.
```

## Função REPLACE

Este comando tem a função de substituir um character ou sentença dentro de uma string ou lista.

### SYNTAX

---

REPLACE ( *source-string* , *from-string* , *to-string* )

Exemplo:

```
  DEFINE VARIABLE vteste AS CHARACTER INITIAL "abacadae".
  ASSIGN vteste = REPLACE(vteste,"a","X").
  DISPLAY vteste.
```

Resultado:

**XbXcXdXe**

## Função RIGHT-TRIM

Comando para tirar espaços em branco localizados a direita de uma string.

## SYNTAX

---

RIGHT-TRIM ( *expression* [ , *trim-chars* ] )

Exemplo:

```
DEFINE VARIABLE vteste AS CHARACTER INITIAL "xxxx  ".
DISPLAY vteste FORMAT "x(20)" SKIP. /* Forma Normal */
DISPLAY RIGHT-TRIM(vteste). /* Sem espaços direita */
```

Resultado:

DE Procedure Editor - Run	
vteste	
	xxxx
	xxxx

## Função ROUND

Esta função tem o objetivo de arredondar valores de campo ou variável do tipo decimal, tantas quantas casas decimais forem informadas na chamada da função.

## SYNTAX

---

ROUND ( *expression* , *precision* )

Exemplo:

```
DEFINE VARIABLE vteste AS DECIMAL INITIAL 123.91734305.
ASSIGN vteste = ROUND(vteste,2).
DISPLAY vteste.
```

Resultado:

DE Procedure E	
vteste	
	123,92

## Função SEARCH

Esta função retorna o nome do arquivo informado caso este seja encontrado no caminho indicado, se não encontra retorna interrogação.

### SYNTAX

---

SEARCH ( *opsys-file* )

Exemplo:

```
IF SEARCH("c:\temp\pagina.txt") <> ? THEN
  MESSAGE "Arquivo existe neste local"
  VIEW-AS ALERT-BOX INFORMATION.
ELSE
  MESSAGE "Arquivo nao existe neste local"
  VIEW-AS ALERT-BOX ERROR.
```

## Função STRING

Esta função converte um character, variável ou campo inteiro, decimal ou data para string(texto).

### SYNTAX

---

STRING ( *source* [ , *format* ] )

Exemplo:

```
DEFINE VARIABLE vteste AS CHARACTER.
DEFINE VARIABLE vnum AS INTEGER INITIAL 99999.

ASSIGN vteste = STRING(vnum).
DISPLAY vteste.
```

## Função SUBSTRING

Esta função tem o objetivo de retornar parte do valor de uma string, serve para você desmembrar uma string em vários pedaços caso você saiba que ela tenha um conteúdo regular.

### SYNTAX

---

SUBSTRING ( *source* , *position* [ , *length* [ , *type* ] ] )

Exemplo:

```
DEFINE VARIABLE vteste AS CHARACTER INITIAL "31/12/2015".
```

```

DISPLAY "Dia: " SUBSTRING(vteste,1,2) SKIP
      "Mês: " SUBSTRING(vteste,4,2) SKIP
      "Ano: " SUBSTRING(vteste,7,4).

```

Resultado:

Dia:	31
Mês:	12
Ano:	2015

## Função TIME

Esta função retorna o número de segundos a partir da meia noite, sendo assim possibilita efetuar cálculos entre dois horários sem necessitar maiores tratamentos. Também pode ser utilizada com formatos específicos de string para retornar hora, minuto e segundos.

## SYNTAX

### TIME

Exemplo:

```

DEFINE VARIABLE vprimeiro AS INTEGER.
DEFINE VARIABLE vsegundo AS INTEGER.

```

```

ASSIGN vprimeiro = TIME.
PAUSE 10.
ASSIGN vsegundo = TIME.

```

```

DISPLAY vsegundo - vprimeiro WITH FRAME x.
DISPLAY STRING(vsegundo - vprimeiro,"HH:MM:SS")
      WITH FRAME y.
DISPLAY STRING(TIME,"HH:MM:SS") WITH FRAME z.

```

Resultado:

<b>OE</b> Procedure E
10
00:00:10
14:42:30

## Função TODAY

Esta função retorna a data do dia.

## SYNTAX

---

TODAY

Exemplo:

```
DISPLAY TODAY.
```

### Função TRIM

Função para retirar espaços em branco localizados em uma string.

## SYNTAX

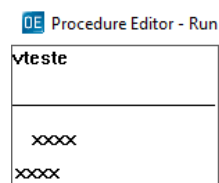
---

TRIM ( *expression* [ , *trim-chars* ] )

Exemplo:

```
DEFINE VARIABLE vteste AS CHARACTER INITIAL " xxxx ".
DISPLAY vteste FORMAT "x(20)" SKIP. /* Forma Normal */
DISPLAY TRIM(vteste).             /* Sem espaços */
```

Resultado:



### Função TRUNCATE

Função para truncar a quantidade de casas decimais após a vírgula, tantas decimais quantas forem informadas.

## SYNTAX

---

TRUNCATE ( *expression* , *decimal-places* )

Exemplo:

```
DEFINE VARIABLE vteste AS DECIMAL INITIAL 123.99856980.
DISPLAY TRUNCATE(vteste,2).
```



Resultado:

OE Procedure E
123,99

## Função USERID

Comando que retorna o nome do usuário logado na base de dados conectada.

### SYNTAX

---

USERID [ ( *logical-dbname* ) ]

Exemplo:

**DISPLAY USERID**("sports").

Resultado:

sandroac
----------

## Comando VIEW

Comando para mostrar um campo ou um frame, necessário quando utilizamos diversos frames sobrepostos em uma mesma aplicação.

### SYNTAX

---

VIEW [ STREAM *stream* | STREAM-HANDLE *handle* ] [ *widget-phrase* ] [ IN WINDOW *window* ]

Exemplo:

**VIEW FRAME** f-teste.

## Função WEEKDAY

Função que retorna o número do dia na semana.

### SYNTAX

---

WEEKDAY ( *date* )

Exemplo:

**DISPLAY (WEEKDAY(TODAY)).**

### Função YEAR

Retorna o ano da data informada.

### SYNTAX

---

YEAR ( *date* )

Exemplo:

**DISPLAY YEAR(TODAY).**

## Capítulo 6: MANIPULAÇÃO DE DADOS



## CRIANDO UM REGISTRO

Para criar um registro em uma tabela no PROGRESS® é muito fácil, basta utilizar o comando CREATE ou INSERT.

Vejamos através de exemplos como fazer isso:

Exemplo 1: Criando um registro na tabela Customer.

**INSERT Customer WITH 1 COL.**

Neste exemplo, estamos criando um registro e ao mesmo tempo estamos pedindo para que o usuário entre com os dados neste registro, utilizando para isso o comando INSERT.

Exemplo 2: Criando um registro na tabela Customer.

**CREATE Customer.**  
**UPDATE Customer WITH 1 COL.**

Neste exemplo, estamos criando um registro em branco com o comando CREATE e depois estamos pedindo para que o usuário entre com os dados utilizando o comando UPDATE.

Nos dois casos o resultado será idêntico, mas a forma da construção do código é diferente, conforme podemos observar nos exemplos acima.

Anotações:

---

---

---

---

---

---

---

---

---

---

---

---

## LENDO UM REGISTRO

Para podermos pesquisar, alterar, gravar ou mostrar um determinado registro de uma tabela ou TEMP-TABLE, devemos utilizar o comando FIND.

Este comando pesquisa em uma determinada tabela dados conforme o especificado na condição WHERE, deixando disponível o registro inteiro para a sua utilização (todos os campos).

### SYNTAX

```
FIND [FIRST|LAST|NEXT|PREV|CURRENT] record [constant] [OF table] [WHERE
expression]
[USE-INDEX index] [USING [FRAME frame] field [AND [FRAME frame] field]...]
[SHARE-LOCK|EXCLUSIVE-LOCK|NO-LOCK] [NO-WAIT] [NO-PREFETCH] [NO-
ERROR]
```

Exemplo 1: Lendo o registro tendo a chave única em mãos

```
FIND Customer 1.
/* ou */
FIND Customer WHERE
    Customer.cust-num = 1.
```

**DISPLAY** Customer.name.

Resultado:

Name
Lift Line Skiing

Exemplo 2: Lendo o primeiro registro da condição escolhida

```
FIND FIRST Customer WHERE
    Customer.name BEGINS "A".
```

**DISPLAY** Customer.name.

Resultado:

Name
ABC Mountain Bikes

Exemplo 3: Lendo o último registro da condição escolhida

```
FIND LAST Customer WHERE
    Customer.name BEGINS "A".
```



## DELETANDO UM REGISTRO

Para excluirmos um registro no banco de dados, inicialmente teremos que ler este registro e deixá-lo disponível na memória para depois apagá-lo.

**Atenção !!** Após excluir um registro do banco, este se torna irrecuperável, pois o PROGRESS® não marca e oculta os registros excluídos como algumas linguagens de mercado. Ele literalmente mata o registro, salvo as transações ainda em execução, conforme o explicado no item TRANSAÇÕES deste documento.

Verifique os exemplos abaixo:

Exemplo 1: Eliminando apenas um registro

```
FIND Customer WHERE  
Customer.cust-num = 10  
EXCLUSIVE-LOCK NO-ERROR.
```

```
IF AVAIL Customer THEN  
DELETE Customer.
```

Este exemplo procura no banco de dados um registro com o código 10, na tabela Customer e depois verifica se esta pesquisa foi bem sucedida, para então somente efetuar a deleção do registro.

Exemplo 2: Eliminando vários registros

```
FOR EACH Customer WHERE  
Customer.cust-num > 80  
EXCLUSIVE-LOCK:
```

```
DELETE Customer.  
END.
```

Este exemplo procura por TODOS os registros da tabela Customer com código acima de 80 e elimina do banco de dados.

Anotações:

---

---

---

---

---

---

---

## Capítulo 7: MANIPULAÇÃO DE TABELAS



### TEMP-TABLE

Até a versão 6 do Progress® existia as figuras de tabelas temporárias que eram as chamadas WORKFILE e WORK-TABLE, que foram obviamente a base para a criação da atual TEMP-TABLE.

O conceito da TEMP-TABLE é criar uma estrutura de definições e dados em memória com as característica de uma tabela criada em banco de dados, esta similaridade apesar dos meios físicos onde elas estão de fato estão criadas serem diferentes é o que permite usar os mesmos comandos nas duas situações.

Uma das principais características que difere as arquiteturas antigas (que ainda funcionam em versões atuais) em relação a nova é a possibilidade de criar índices também para uma tabela temporária, podendo assim gerar um diferencial de performance em leituras mesmo para uma temporária LIKE uma tabela qualquer pelo fato da possibilidade de criação de um índice adequando para uma determinada leitura.

Alguns motivos para usar:

Ganho de performance através de índices específicos para o programa em questão;  
Armazenar cálculos e outras combinações que tornam o programa mais complexo de se ler, separando a parte de criação da parte que mostra os dados, desta forma a segunda

ficaria apenas com os comandos de formatação de tela e/ou relatório e por consequência mais fácil de se dar manutenção;

Se pode trabalhar com apenas uma faixa (que interessa) de uma grande tabela, porém com a mesma estrutura física da tabela, dando maior performance ao programa;

Se usada de forma compartilhada, pode ser criada em um determinado programa e ser usada em diversos outros a partir deste, ficando assim desnecessária a leitura das tabelas originais novamente, literalmente estaria tudo em memória;

Se você tem uma tabela que tem “quase” tudo o que precisa para seu programa, pode ser usada uma temp-table LIKE esta tabela e adicionar somente os campos necessários e terá tudo em uma única estrutura, facilitando assim seu desenvolvimento.

## SYNTAX

```
[DEFINE [ [ NEW [ GLOBAL ] ] SHARED ] TEMP-TABLE temp-table-name [ NO-UNDO ]
[ LIKE table-name [ VALIDATE ] [ USE-INDEX index-name [ AS PRIMARY ] ] ... ]
[ RCODE-INFORMATION ] [ FIELD field-name { AS data-type | LIKE field [ VALIDATE ]
}
[ field-options ] ] ...
[ INDEX index-name
[ IS [ UNIQUE ] [ PRIMARY ] [ WORD-INDEX ] ]
{ index-field [ ASCENDING | DESCENDING ] } ... ] ...
```

Exemplo 1: Definindo uma TEMP-TABLE com definições de campos, tabela e índices iguais a uma tabela existente no banco de dados.

```
DEFINE TEMP-TABLE tt-cliente LIKE Customer.
```

Exemplo 2: Definindo uma TEMP-TABLE com seus próprios campos.

```
DEFINE TEMP-TABLE tt-dados
FIELD tt-nome AS CHARACTER
FIELD tt-idade AS INTEGER.
```

Exemplo 3: Definindo uma TEMP-TABLE igual a uma tabela existente e com campos adicionais.

```
DEFINE TEMP-TABLE tt-cliente LIKE Customer
FIELD tt-nome AS CHARACTER
FIELD tt-idade AS INTEGER.
```

Exemplo 4: Definindo índices para uma TEMP-TABLE.

```
DEFINE TEMP-TABLE tt-cliente
FIELD tt-codigo AS INTEGER
FIELD tt-nome AS CHARACTER
FIELD tt-idade AS INTEGER
INDEX primario IS PRIMARY tt-codigo
```



**INDEX** nome                      **tt-nome.**

A TEMP-TABLE é utilizada normalmente como se fosse uma tabela do banco de dados.

Exemplo 5: Utilização completa de uma TEMP-TABLE (passo a passo)

Normalmente dividimos em quatro etapas:

#### 1) Criação da TEMP-TABLE

Onde é definida as características desejadas para a utilização, basicamente o que vimos nos quatro exemplos anteriores.

#### 2) Limpeza da TEMP-TABLE

É comum esquecer de limpar/zerar a TEMP-TABLE durante a utilização e principalmente quando a sua alimentação se dá em um laço de repetição, fazendo com que o programa dobre os valores a cada execução sem sair da mesma sessão. Então a sugestão é ter a limpeza como sendo um passo da utilização que nunca terá esta preocupação.

#### 3) Alimentação da TEMP-TABLE

O local onde acontece a lógica que gera os dados na TEMP-TABLE que vai servir de base para a sua utilização no próximo passo, importante pois se nascer os dados errados o resultado final de toda a aplicação também não será o adequado.

#### 4) Utilização da TEMP-TABLE

É neste momento que se faz jus a criação de toda a arquitetura da TEMP-TABLE é onde você mostra ou usa para um cálculo o lógica aquilo que gravou no passo 3.

Segue então um programa que lista os 10 maiores clientes, considerando pedidos criados no sistema, identificando as 4 etapas citadas acima.

**DEFINE VARIABLE** vqtd\_cli **AS INTEGER INITIAL** 0.

**/\* FASE 1 - Criação TEMP-TABLE \*/**

**DEFINE TEMP-TABLE** tt-dados

**FIELD** cod\_cliente **LIKE** Customer.Cust-Num **LABEL** "Código"

**FIELD** nom\_cliente **LIKE** Customer.Name **LABEL** "Nome"

**FIELD** vlr\_pedidos **AS DECIMAL** **LABEL** "Pedidos".

**/\* FASE 2 - Limpeza TEMP-TABLE \*/**

**EMPTY TEMP-TABLE** tt-dados.

**/\* FASE 3 - Alimentação TEMP-TABLE \*/**

**FOR EACH** Customer **NO-LOCK,**

EACH Order OF Customer NO-LOCK,  
EACH Order-Line OF Order NO-LOCK:

FIND tt-dados WHERE

tt-dados.cod\_cliente = Customer.cust-num  
NO-LOCK NO-ERROR.

IF NOT AVAIL tt-dados THEN

DO:

CREATE tt-dados.

ASSIGN tt-dados.cod\_cliente = Customer.cust-num

tt-dados.nom\_cliente = Customer.name.

END.

ASSIGN tt-dados.vlr\_pedidos = tt-dados.vlr\_pedidos  
+ (Order-Line.qty  
\* Order-Line.price).

END.

/\* FASE 4 - Utilização TEMP-TABLE \*/

FOR EACH tt-dados

BY tt-dados.vlr\_pedidos DESCENDING:

ASSIGN vqtd\_cli = vqtd\_cli + 1.

IF vqtd\_cli > 10 THEN

LEAVE.

DISPLAY tt-dados.cod\_cliente

tt-dados.nom\_cliente

tt-dados.vlr\_pedidos

WITH FRAME f-dados DOWN.

END.

Importante: TEMP-TABLE LIKE <TABELA> não herda triggers da tabela e uma TEMP-TABLE também não pode ter triggers, essa é a diferença de tratamento em relação a uma tabela normal do banco de dados. Reforçando: Os demais comandos podem ser utilizados normalmente em uma TEMP-TABLE.

# BUFFER

Você deseja utilizar mais de um registro de uma mesma tabela no mesmo programa, porém não quer ter que ler novamente a tabela a cada vez que precisar utilizar algum dado.

O BUFFER nada mais é do que uma imagem da tabela, ele cria esta imagem tantas quantas vezes você quiser dentro do fonte e pode usar isoladamente cada uma destas instâncias, mantendo assim um ponteiro exclusivo para cada um dos registros que você precisa sem interferir na outra leitura que você precisa da mesma tabela.

E no momento que você cria um registro novo na tabela, o BUFFER é automaticamente atualizado também então a imagem da tabela se mantém.

O BUFFER pode ser identificado como local (utilização no próprio programa) ou shared (podendo ser compartilhado em outros programas chamados pelo mesmo).

## SYNTAX

```
DEFINE { [ [ NEW ] SHARED ] } [ PRIVATE | PROTECTED ] [ STATIC ] } BUFFER buffer-
name
FOR [ TEMP-TABLE ] table-name [ PRESELECT ] [ LABEL label-name ]
[ NAMESPACE-URI namespace ] [ NAMESPACE-PREFIX prefix ] [ XML-NODE-NAME
node-name ]
```

Exemplo 1: No trecho de código abaixo, vamos simular uma situação que não existe no banco de dados, ou seja, o banco não foi modelado para ter este tipo de artifício, porém didaticamente daremos o exemplo como se fossem mais de uma leitura de registros sobre a mesma tabela, apresentando os dados isoladamente sem perder o ponteiro do registro lido justamente utilizando a questão apresentada acima que é o BUFFER de uma tabela do banco de dados.

```
DEFINE BUFFER b-fiscal FOR Customer.
DEFINE BUFFER b-entrega FOR Customer.
DEFINE BUFFER b-cobranca FOR Customer.
```

```
FIND Customer WHERE
    Customer.cust-num = 1
NO-LOCK NO-ERROR.
```

```
DISPLAY Customer.cust-num
    Customer.NAME
    WITH FRAME f-dados SIDE-LABELS WIDTH 80
TITLE " Dados Cliente ".
```

```
FIND b-fiscal WHERE
    b-fiscal.cust-num = 2
NO-LOCK NO-ERROR.
```

```
DISPLAY b-fiscal.city    FORMAT "X(10)" SKIP
    b-fiscal.state    FORMAT "X(10)" SKIP
    b-fiscal.country  FORMAT "X(10)" SKIP
    WITH FRAME f-fiscal SIDE-LABELS
```



# ORDENAÇÃO

Para ordenarmos uma tabela não se faz necessário a criação de um índice definido para tal, basta mandar o programa ordenar a pesquisa por um determinado campo.

Funciona bem, porém sempre é interessante basear uma pesquisa em índices pois nem sempre é uma tarefa fácil prever o crescimento de uma tabela e isso pode ao longo do tempo trazer lentidão para a rotina que se está criando.

Exemplo1:

```
FOR EACH Customer NO-LOCK  
BY Customer.Credit-limit:
```

```
DISPLAY Customer WITH 2 COL 1 DOWN.
```

END.

Neste exemplo estamos lendo a tabela Customer e criando uma ordenação pelo campo de limite de crédito, utilizando para isto “apenas” argumento BY.

Se observar a tabela Customer não temos um índice definido para o campo acima utilizado (Credit-limit), mas mesmo assim a ordenação foi executada conforme projetada no fonte.

Exemplo2:

```
FOR EACH Customer NO-LOCK  
BY Customer.Credit-limit  
BY Customer.name:
```

```
DISPLAY Customer WITH 2 COL 1 DOWN.
```

END.

Neste exemplo estamos lendo a mesma tabela do Exemplo 1, porém além da ordenação do campo Limite de Crédito adicionamos mais uma ordenação por nome do cliente, portanto se houverem dois clientes com o mesmo limite de crédito ele ainda vai deixar em ordem alfabética baseado no nome do cliente. Posso incrementar tantos quantos BY, claro que dependendo do tamanho da tabela e da ausência de índices para a nossa escolha a possibilidade de trazer lentidão se torna progressiva.

Anotações:

---

---

---

## Capítulo 8: DISPOSITIVOS DE SAÍDA



### OUTPUT TO

O Progress® trabalha com diversos tipos de saídas para mostrar os dados, que podem ser perfeitamente manipuladas pelo desenvolvedor. A saída default ao abrir o editor sempre vai ser a saída em tela, internamente setada como OUTPUT TO TERMINAL.

Sempre que for aberta uma saída no programa, independente de qual o formato a mesma deverá ser fechada para que as interações com o usuário possam voltar a acontecer em tela e isso obviamente vai depender sempre da lógica idealizada para cada programa. Para fechar a saída e voltar a “tela”, usamos o comando OUTPUT CLOSE.

Em relação ao OUTPUT CLOSE, vale salientar que não tem a necessidade do “TO”, diferentemente de quando se abre saída que neste caso sim exige que tenha o “TO”, ao usar indevidamente esta cláusula para fechar o arquivo/saída o que vai acontecer é que de fato não estaremos fechando como idealizado e sim abrindo um outro arquivo/saída chamado CLOSE. É comum acharmos no sistema operacional de grandes sistemas/ERP's arquivos chamado CLOSE nas áreas de trabalho.

## SYNTAX

```
OUTPUT [ STREAM stream | STREAM-HANDLE handle ] TO { PRINTER [ printer-name ]
| opsys-file
| opsys-device | TERMINAL | VALUE ( expression ) | "CLIPBOARD" } [ LOB-DIR { constant
|
VALUE ( expression ) } ] [ NUM-COPIES { constant | VALUE ( expression ) } ] [ COLLATE
]
[ LANDSCAPE | PORTRAIT ] [ APPEND ] [ BINARY ] [ ECHO | NO-ECHO ]
[ KEEP-MESSAGES ] [ NO-MAP | MAP protermcap-entry ] [ PAGED ] [ PAGE-SIZE {
constant | VALUE ( expression ) } ] [ UNBUFFERED ] [ NO-CONVERT | { CONVERT
[ TARGET target-codepage ] [ SOURCE source-codepage ] }
```

Anotações:

---

---

---

---

---

---

---

---

---

---

## EM TELA

Para as saídas em tela não é necessário direcionar nada no programa, pois como citado acima o default do Progress® é a saída em tela, então é somente utilizar os comando e poder interagir diretamente com ele através do ABL.

Exemplo 1:

**DISPLAY** "Olá Amigo".

Exemplo 2:

**DISPLAY** "Olá Amigo" **WITH FRAME** f-teste **WIDTH 40 CENTERED**.

Exemplo 3:

**MESSAGE** "Olá Amigo" **VIEW-AS ALERT-BOX INFORMATION**.



Anotações:

[illegible]

## NA IMPRESSORA

Para imprimir no Progress® basta direcionar a saída para tal, isto é, dizer onde serão mostrados os dados e utilizar os mesmos comandos de saída para a tela como citado nos exemplos anteriores (DISPLAY, MESSAGE...).

Você pode também antes de tudo, definir um FRAME ajustando o label, o formato e a posição dos campos e só depois utilizar o DISPLAY referenciando o FRAME criado, para imprimir. Isto trará um resultado estético mais interessantes das informações a serem impressas.

A omissão do nome da impressora fará com que saia diretamente na impressora padrão setada para a sessão que você está usando, no Windows na impressora padrão e no Linux na impressora que estiver cadastrada como default (caso exista parametrização).

Exemplo 1:

**OUTPUT TO PRINTER** <nome\_impressora>.  
**DISPLAY** “Esta mensagem saiu na impressora !!” **WITH NO-LABEL**.  
**OUTPUT CLOSE**.





Exemplo 2:

```
DEFINE FRAME f-imprime  
  Customer.cust-num LABEL "Codigo"  
  Customer.cust-name LABEL "Nome do Cliente"  
  WITH CENTERED.
```

```
OUTPUT TO PRINTER <nome_imprensa>.
```

```
FIND FIRST Customer NO-LOCK NO-ERROR.
```

```
DISPLAY Customer.cust-num  
  Customer.cust-name  
  WITH FRAME f-imprime.
```

```
OUTPUT CLOSE.
```

Anotações:

---

---

---

---

---

---

## EM ARQUIVO

Como a exemplo das demais saídas para o Progress®, para você gerar em arquivo basta direcionar o caminho e nome para o mesmo que todo o conteúdo que originalmente apareceria na tela, vai ser gravado neste arquivo.

Neste caso podemos ter variações, para gerar em formato XML e também para gerar em formato Excel, ou até mesmo com character separador.

Exemplo 1: Simples geração direta no sistema operacional (Windows) de um arquivo texto.

```
OUTPUT TO c:\temp\arquivo.txt.
```

```
FIND FIRST Customer NO-LOCK NO-ERROR.
```

```
DISPLAY Customer.cust-num  
  Customer.cust-name  
  WITH FRAME f-imprime.
```

```
OUTPUT CLOSE.
```

Exemplo 2: Mesmo exemplo 1, porém gerando em extensão CSV, que é a extensão conhecida do Excel para separador “;”(ponto e vírgula) para os campos, neste caso abrirá corretamente os campos abaixo cada um em sua devida coluna sem misturar todo o texto na mesma célula fato que ocorreria na abertura de um arquivo texto normal.

Acrescentamos que neste exemplo não estamos trabalhando a formatação das células, pode sim ser feito este tipo de tratamento via ABL, porém não com este tipo de saída que estamos tratando neste capítulo.

**OUTPUT TO** c:\temp\arquivo.csv.

**FIND FIRST** Customer **NO-LOCK NO-ERROR**.

**PUT** Customer.cust-num “;”  
Customer.cust-name “;” **SKIP**.

**OUTPUT CLOSE**.

Anotações:

---

---

---

---

---

## NA ÁREA DE TRANSFERÊNCIA

Assim como podemos direcionar textos e conteúdo para diversas saídas previamente definidas, também podemos pegar estes mesmos conteúdos e colocar na área de transferência do sistema operacional, neste caso Windows para que seja “colada” onde o usuário desejar, seja no pacote Office ou qualquer outra aplicativo compatível com famoso “CTRL+V”.

Exemplo 1: Direcionamento simples para área de transferência.

**OUTPUT TO** "clipboard".  
**DISPLAY** "Olá Amigo !!" **WITH CENTERED NO-LABEL**.  
**OUTPUT CLOSE**.

Exemplo 2: Trocando o direcionamento durante a lógica do programa.

**DISPLAY** “Isto saiu na tela !!” **WITH NO-LABEL CENTERED**.

**OUTPUT TO** "clipboard".  
**DISPLAY** “Isto foi p/área de transferência!”  
**WITH NO-LABEL CENTERED**.



**OUTPUT CLOSE.**

**DISPLAY “Saiu novamente na tela !!” WITH NO-LABEL CENTERED.**

Anotações:

This image shows a single sheet of white paper with horizontal blue ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

## EM STREAM

Em casos mais complexos onde precisamos setar mais de uma saída em arquivo simultaneamente ainda temos a possibilidade de uma STREAM. Podemos no mesmo bloco ou laço de repetição estar direcionando dados a diversos arquivos, sem perder a integridade da transação como a de cada um dos arquivos, bastante utilizado em geração de arquivos de EDI e outras integrações proprietárias.

Assim como o direcionamento de outra saída qualquer, também temos/devemos que fechar as tantas STREAM's utilizadas durante a lógica de meu programa.

Exemplo:

**DEFINE STREAM** st-cliente.

**DEFINE STREAM** st-item.

**OUTPUT STREAM** st-cliente **TO VALUE**("c:\temp\cliente.txt") **PAGED**.

**OUTPUT STREAM** st-item **TO VALUE**("c:\temp\item.txt").

**FOR EACH** Customer **NO-LOCK**:

**DISPLAY STREAM** st-cliente Customer.cust-num.

**END**.

**FOR EACH** item **NO-LOCK**:

**DISPLAY STREAM** st-item item.item-num.

**END**.

**OUTPUT STREAM** st-cliente **CLOSE**.

**OUTPUT STREAM** st-item **CLOSE**.

Anotações:

---

---

---

---

---

---

---

---

---

---

---

---

# Capítulo 9: INTERAGINDO COM OUTROS PROGRAMAS



## PROCEDURES EXTERNAS

Procedures externas nada mais são que chamadas de outros programas/fontes e isso é uma prática comum em todas as linguagens de programação.

Por manter um padrão por assuntos, ou até mesmo por reaproveitamento de códigos, utilizamos este artifício, que é a qualquer momento invocar um programa externo para executar, calcular ou buscar dados para complementar uma determinada rotina de nossa lógica.

Por fim ainda por questões de tamanho e de dificuldade de manutenção dos sistemas, mantemos eles o mais granular possível.

Exemplo 1:

```
comando 1
comando 2
comando 3
.
.
.
run pro_teste.p.
```

comando 4

comando 5

·  
·

A chamada acima no MEIO de nosso programa, vai executar uma outra rotina que não faz parte do fonte que estamos trabalhando, o Progress® vai executar esta rotina, voltar para a linha imediatamente abaixo, e dar sequencia ao meu programa.

Anotações:

---

---

---

---

---

---

---

---

---

---

## PROCEDURES INTERNAS

Já no caso das procedures internas, a situação é bem semelhante para a sua execução, porém bem distintas em relação a sua e forma de acesso.

As procedures internas NÃO são vistas pelos demais programas do sistema, assim sendo não podem ser chamadas por estes, ficando assim restritas somente para a utilização do próprio código fonte onde ela reside.

Outro detalhe é que ela tem que necessariamente estar identificada como "PROCEDURE" dentro do próprio fonte.

A falta desta identificação pode fazer com que o programa depois de não ter encontrado a referência internamente saia para o sistema operacional para buscar uma procedure externa (programa) com aquele nome em questão.

O principal motivo de uma procedure externa é a possibilidade de ser chamada diversas vezes a mesma lógica em vários pontos do programa, com isso o trecho não precisa ser repetido, fazendo com que o fonte fique mais enxuto e facilita também a sua manutenção futura.

Exemplo 1:

comando 1

comando 2

·  
·

```
RUN pro_teste
comando 3
comando 4
.
.

PROCEDURE pro-teste:

    comando 1
    comando 2
    comando 3

END PROCEDURE.
```

Anotações:

---

---

---

---

---

---

## PASSAGEM PARÂMETROS

Quando falamos de um sistema a necessidade de um programa se comunicar com outro é eminente, as chamadas de outras rotinas existem desde a montagem do menu do sistema, algumas vezes passando informações outras recebendo dependendo da lógica e no ABL isto não é diferente.

Antes vamos detalhar as melhores práticas e regras para não termos erros de execução ou resultados inesperados na passagem de parâmetros, para isso vamos imaginar uma situação onde você está no programa A e este precisa se comunicar com o programa B através de passagem de parâmetros, para facilitar o entendimento vamos denominar então didaticamente que o programa A é o programa **CHAMADOR** e o programa B é o programa **CHAMADO**, seguem então as recomendações para a melhor utilização do recurso da linguagem:

- 1) O programa **CHAMADO** deve estar preparado para receber os parâmetros oriundos do programa **CHAMADOR**;
- 2) A quantidade de parâmetros utilizados pelo programa **CHAMADOR** deve ser a mesma declarada no programa **CHAMADO**;
- 3) O tipo de dado dos parâmetros utilizados pelo programa **CHAMADOR** deve ser os mesmos declarados pelo programa **CHAMADO**;
- 4) A sequência dos parâmetros no programa **CHAMADOR** tem que ser a mesma declarada no programa **CHAMADO**;

- 5) O tipo de parâmetro declarado no momento em que o programa **CHAMADO** é referenciado no programa **CHAMADOR** tem que ser respeitado em ambos, ou seja, não posso dizer que estou enviando um parâmetro para o **CHAMADO** se o **CHAMADO** está também querendo enviar um parâmetro, ele deve sim estar preparado para receber um parâmetro que daí sim o ciclo se completa;
- 6) O nome das variáveis ou campos utilizados no programa **CHAMADOR**, não tem a necessidade de ser os mesmos nomes declarados no programa **CHAMADO**, desde que respeite os 5 itens anteriores.

## SYNTAX

```
DEFINE {INPUT|OUTPUT|INPUT-OUTPUT|RETURN} PARAMETER parameter
{{LIKE field}}{{AS [HANDLE [TO]] datatype}} [[NOT] CASE-SENSITIVE]
[FORMAT string] [DECIMALS n] [INITIAL constant] [COLUMN-LABEL label] [LABEL
string] [NO-UNDO]
```

```
DEFINE PARAMETER BUFFER buffer FOR table [PRESELECT] [LABEL label]
```

```
DEFINE { INPUT | OUTPUT | INPUT-OUTPUT } PARAMETER TABLE FOR temp-table-
name [APPEND]
```

Exemplo 1: Passando parâmetro do programa **CHAMADO** para o programa **CHAMADOR**.

```
/* ===== Programa CHAMADOR.p ===== */

RUN chamado.p (INPUT 10).

/* ===== Programa CHAMADO.p ===== */

DEFINE INPUT PARAMETER vparam AS INTEGER.

DISPLAY vparam LABEL "Parâmetro recebido" WITH SIDE-LABELS.
```

Exemplo 2: Recebendo parâmetro do programa **CHAMADOR** para ser usado no programa **CHAMADO**.

```
/* ===== Programa CHAMADOR.p ===== */

    DEFINE VARIABLE vretorna AS CHARACTER.

RUN chamado.p (OUTPUT vretorna).

DISPLAY vretorna LABEL "Parâmetro recebido" WITH SIDE-LABELS.

/* ===== Programa CHAMADO.p ===== */

DEFINE OUTPUT PARAMETER vretorna AS CHARACTER.
```



**ASSIGN** vretorna = "TESTE".

Exemplo 3: Enviando e recebendo o mesmo parâmetro do programa **CHAMADO**.

```
/* ===== Programa CHAMADOR.p ===== */

    DEFINE VARIABLE venv_rec AS CHARACTER.

    ASSIGN venv_rec = "".

    DISPLAY venv_rec LABEL "Parâmetro enviado" WITH SIDE-LABELS.

    RUN chamado.p (INPUT-OUTPUT venv_rec).

    DISPLAY venv_rec LABEL "Parâmetro recebido" WITH SIDE-LABELS.

/* ===== Programa CHAMADO.p ===== */

    DEFINE INPUT-OUTPUT PARAMETER venv_rec AS CHARACTER.

    ASSIGN venv_rec = "TESTE".
```

Exemplo 4: Enviando e recebendo mais de um parâmetro do programa **CHAMADO**.

```
/* ===== Programa CHAMADOR.p ===== */

    DEFINE VARIABLE venvia AS CHARACTER INITIAL "ABCDE".
    DEFINE VARIABLE vrecebe AS CHARACTER.
    DEFINE VARIABLE venv_rec AS CHARACTER INITIAL "XYZ".

    RUN chamado.p (INPUT      venvia,
                   OUTPUT      vrecebe,
                   INPUT-OUTPUT venv_rec).

    DISPLAY venvia LABEL "Parâmetro Enviado" SKIP
           vrecebe LABEL "Primeiro Parâmetro Recebido" SKIP
           venv_rec LABEL "Segundo Parâmetro Recebido"
           WITH SIDE-LABELS.

/* ===== Programa CHAMADO.p ===== */

    DEFINE INPUT      PARAMETER var1 AS CHARACTER.
    DEFINE OUTPUT      PARAMETER var2 AS CHARACTER.
    DEFINE INPUT-OUTPUT PARAMETER var3 AS CHARACTER.

    DISPLAY var1 LABEL "Parâmetro Recebido" WITH SIDE-LABELS.
```

```
ASSIGN var2 = "TESTE2"  
var3 = "TESTE3".
```

Exemplo 5: Enviando uma TEMP-TABLE

```
/* ===== Programa CHAMADOR.p ===== */
```

```
DEFINE TEMP-TABLE tt-dados  
  FIELD codigo AS INTEGER  
  FIELD nome AS CHARACTER.
```

```
CREATE tt-dados.  
ASSIGN tt-dados.codigo = 1  
tt-dados.nome = "Teste".
```

```
CREATE tt-dados.  
ASSIGN tt-dados.codigo = 2  
tt-dados.nome = "Outro Teste".
```

```
RUN chamado.p(INPUT TABLE tt-dados).
```

```
/* ===== Programa CHAMADO.p ===== */
```

```
DEFINE INPUT PARAMETER TABLE FOR tt-dados.
```

```
FOR EACH tt-dados:  
  DISPLAY tt-dados.  
END.
```

## INCLUDES

Includes são normalmente pequenos trechos de códigos fontes em ABL, que podem ser utilizados dezenas de vezes em diversos programas, ou até mesmo diversas vezes no mesmo programa.

Normalmente uma include nasce de uma rotina repetitiva que tem que ser escritas diversas vezes no sistema, assim sendo fica de uso comum a todos os programas que precisarem fazer uso desta referida rotina.

A vantagem é que quando precisar ser alterada ou ajustada uma include isto será feito uma única vez em apenas um único lugar, porém aqui tem um pequeno inconveniente que é o fato de que para os programas que usam a includes assimilarem os ajustes feitos precisam ser compilados novamente e somente aí terão o benefício da alteração feita. Neste contexto uma procedure externa pode ter vantagem em relação a include, pois a mesma pode ser alterada e compilada isoladamente e todos os programas que utilizam podem fazer a

chamada imediata que já estará valendo a nova versão compilada, independentemente de quantos programas a utilizam.

No momento da compilação de um programa que utiliza uma include é como se o código da include fosse copiado para dentro do fonte do programa tantas quantas forem as vezes em que for chamada na lógica sendo assim para o executável como se o programador tivesse escrito de fato aquele código todas estas vezes, porém visualmente no editor aparece só a linha da chamada da include ficando assim um fonte mais limpo e de melhor entendimento para quem desenvolve.

A extensão de uma include normalmente é “.i”, porém o ABL permite que seja criada outras extensões, pois o que caracteriza uma include de fato é a forma de sua chamada que se dá entre chaves, como por exemplo {include.i}.

Exemplo: A include é chamada mais de uma vez no programa.

**FOR EACH Customer NO-LOCK:**

{mostra\_cliente.i}

**END.**

**FIND FIRST Customer NO-LOCK NO-ERROR.**

{mostra\_cliente.i} **WHEN AVAIL Customer.**

**/\* Include Mostra Dados Cliente \*/**

**DISPLAY Customer WITH 2 COL 1 DOWN.**

## EXTENSÕES PROGRESS

A linguagem Progress® tem suas extensões próprias, segue abaixo detalhada as principais delas:

- P    Procedures e triggers de banco de dados;
- I    Includes;
- R    Programas compilados, o executável do Progress®;
- DB   Base de dados;
- BI   Before Image da base de dados;
- AI   After image da base de dados;
- LK   Lock da base de dados, normalmente indicador se a base está no ar ou não, porém podem existir situações em que isso não é uma verdade absoluta;
- LG   Registro de log das ocorrências relativas a uma base de dados;
- W    Programas da versão Windows do Progress®, Smart Windows, Smart Dialog, Smart Frames, Smart Browsers, Smart Viewers, Smart Panels e Smart Querys;
- WX   Templates;
- WBX Compilação com objetos VBX/OCX;



## Capítulo 10: AMBIENTE



## SISTEMA OPERACIONAL

O PROGRESS® possui um comando para identificar em qual Sistema Operacional se está trabalhando no momento.

Este comando é muito útil para empresas que possuem em seu parque de máquinas, sistemas operacionais diversos, tais como MS-Windows, Unix, Linux, MS-DOS, BTOS, OS2, VMS.

Como já foi dito anteriormente, no PROGRESS® podemos reaproveitar os programas fontes para executar sobre os diversos Sistemas Operacionais suportados, sem mexer em uma linha sequer de comandos PROGRESS®, salvo os comandos de interação com o sistema operacional propriamente dito.

Por esta razão existe este comando, que indica em qual Sistema Operacional você está rodando o programa.

### SYNTAX

OPSYS

Exemplo:

```
CASE OPSYS:  
  WHEN "UNIX" THEN UNIX Is.
```

```
WHEN "WIN32" THEN DOS dir.  
  WHEN "MSDOS" THEN DOS dir.  
    WHEN "OS2" THEN OS2 dir.  
    WHEN "VMS" THEN VMS directory.  
    OTHERWISE MESSAGE "Sistema Operacional não Suportado"  
  VIEW-AS ALERT-BOX ERROR.  
END CASE.
```

Neste exemplo, o programa quer mostrar os arquivos no diretório corrente, mas para saber em qual Sistema Operacional está sendo utilizado, ele se baseou no comando OPSYS, que uma vez identificado o sistema operacional mostrou os dados com o comando pertinente e ainda trata o fato de estar sendo em um sistema operacional não tratado no fonte acima.

Anotações:

---

---

---

---

---

---

## MANIPULANDO O SO

No ABL é possível executar qualquer binário do Sistema Operacional, vamos nesta sessão abordar os mais usuais.

OS-COMMAND  
OS-COPY  
OS-CREATE-DIR  
OS-DELETE  
OS-DRIVES

OS-COMMAND

Este comando vai até o Sistema Operacional e executa um comando definido pelo programa que você escreveu, independente de Sistema Operacional que você estiver rodando o Progress®.

Por esta razão que este comando é mais versátil e útil do que o OPSYS, visto anteriormente.

### SYNTAX

OS-COMMAND[ SILENT ][ NO-WAIT ] [ NO-CONSOLE ] [ command-token | VALUE ( expression ) ] ...

Exemplo:

### OS-COMMAND dir.

Neste exemplo mesmo estando no Windows como estando no Linux é o Progress® que vai converter em tempo de execução se deve usar o comando DIR ou comando LS no caso do Linux e nesta situação o meu programa com o mesmo código estaria compatível para as duas plataformas.

Anotações:

---

---

---

---

---

---

---

---

---

---

## COMPILAÇÃO

A compilação se faz necessário no Progress® por diversos motivos, seguem detalhados alguns deles:

- 1) Maior performance para a execução do programa.
  - a. Uma vez compilado o programa não precisará mais ser interpretado e compilado em tempo de execução.
  - b. O .r já carrega todas as características do ambiente para que este rode satisfatoriamente, assim sendo não precisa angariar mais informações no momento da execução.
- 2) Segurança do sistema.
  - a. Um programa compilado está gravado em código binário, assim sendo não pode ser lido, e pessoas que não possuem ligação com o desenvolvimento do sistema não conseguirão entender o que o programa faz através da leitura de seu código fonte.
  - b. Pode ser instalado no cliente sem a necessidade do código fonte estar presente.
- 3) Run timer.
  - a. O Progress® vende uma licença de Progress® chamada run timer, que é mais barata e esta versão somente executa .r, não tem o poder de interpretar o código e gerar o executável em tempo de execução.

## SYNTAX

```

COMPILE { procedure-pathname | class-pathname | VALUE ( expression ) }
[ ATTR-SPACE [ = logical-expression ] ] [ SAVE [ = logical-expression ] ]
[ INTO { directory | VALUE ( expression ) } ] [ LISTING { listfile | VALUE ( expression ) } ]
[ APPEND [ = logical-expression ] | PAGE-SIZE integer-expression |
PAGE-WIDTH integer-expression ] [ XCODE expression ] [ XREF { xreffile | VALUE (
expression ) } ]
[ APPEND [ = logical-expression ] ] [ XREF-XML { directory | filename | VALUE (
expression ) } ]
[ STRING-XREF { sxreffile | VALUE ( expression ) } [ APPEND [ = logical-expression ] ] ]
[ STREAM-IO [ = logical-expression ] ] [ LANGUAGES ( { language-list | VALUE (
expression ) } ) ]
[ TEXT-SEG-GROW = growth-factor ] [ DEBUG-LIST { debugfile | VALUE ( expression )
} ]
[ PREPROCESS { preprocessfile | VALUE ( expression ) } ] [ NO-ERROR ]
[ V6FRAME [ = logical-expression ] ] [ USE-REVVIDEO | USE-UNDERLINE ] ]
[ MIN-SIZE [ = logical-expression ] ] [ GENERATE-MD5 [ = logical-expression ] ]

```

Exemplo 1: Compila e salva um novo .r para o programa teste.p

**COMPILE teste.p SAVE.**

Exemplo 2: Compila e salva um novo .r para o programa teste.p em um outro local definido no momento da compilação

**COMPILE teste.p SAVE INTO c:\temp.**

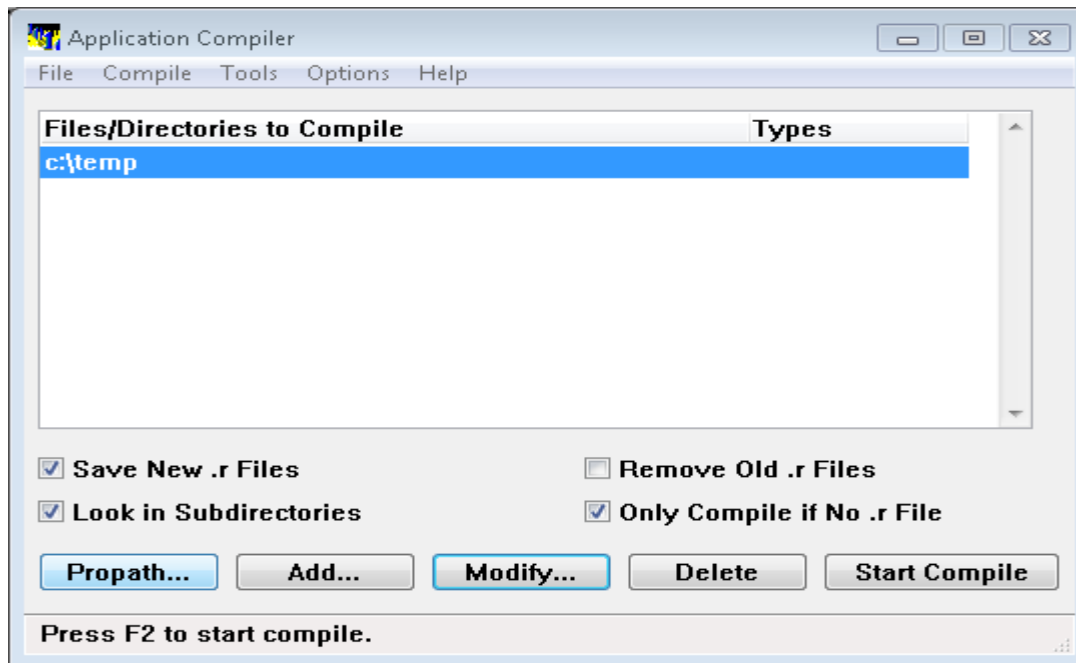
Exemplo 3: Compila e salva um grupo de programas

**COMPILE \*.p SAVE.**

Também podemos compilar programas a partir da editor gráfico do progress, faz tudo que a linha de comando faz, porém com algumas facilidades, segue a tela exemplo para compilar todos os fontes localizados no c:\temp.

Tela Gráfica – Application Compiler



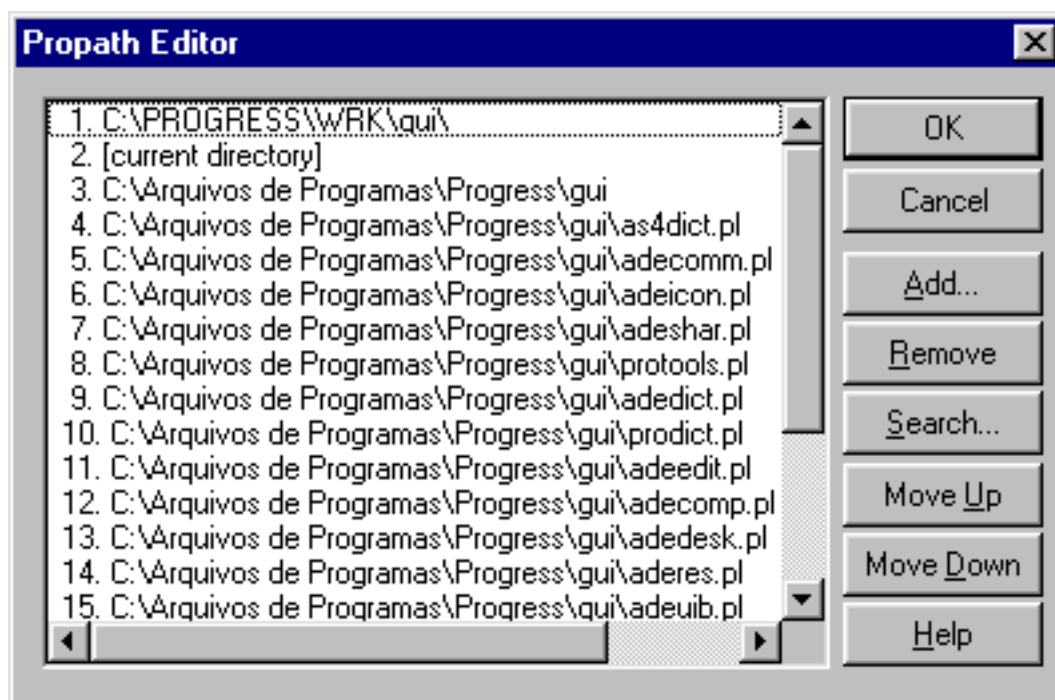


## PROPATH

Propath é o caminho interno do Progress®, caminho que ele respeita para a execução dos programas, seja compilados ou não. Concluimos que o Progress® não vai executar um programa que não esteja em uma das pastas especificadas no PROPATH, a não ser que internamente no programa haja uma chamada fixa para um outro caminho qualquer o que não é usual e recomendado em qualquer linguagem de programação, pois o sistema assim perde a sua portabilidade.

Este caminho é seguido exatamente na ordem em que os objetos estão dispostos, ou seja, se por acaso inadvertidamente um programa qualquer esteja presente em mais de uma pasta contida no PROPATH o sistema vai rodar/utilizar o primeiro que achar de cima para baixo, simplesmente desconsiderando a existência de um outro caminho com o mesmo programa.

Exemplo:



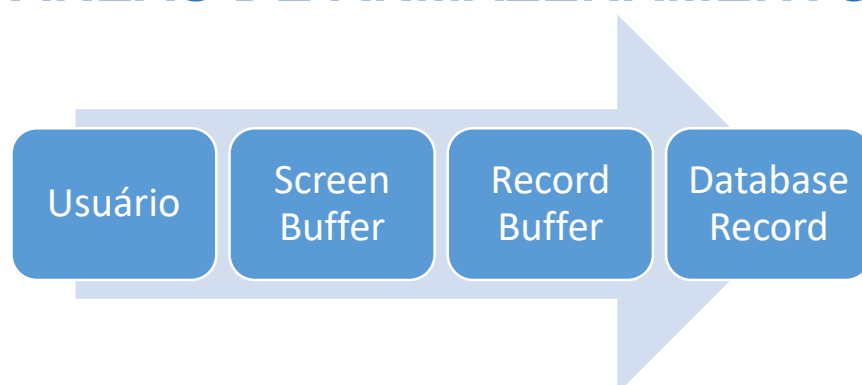
Anotações:

---

---

---

## ÁREAS DE ARMAZENAMENTO



O PROGRESS® possui, várias áreas de armazenamentos de dados, o DATABASE RECORD, O RECORD BUFFER, SCREEN-BUFFER.

Os dados no DATABASE RECORD são aqueles que já estão fisicamente armazenados no banco de dados, no disco. Já o RECORD BUFFER armazena dados temporariamente

enquanto um determinado procedimento está acessando os dados e armazena os valores das variáveis utilizadas pelo procedimento.

O SCREEN BUFFER armazena os dados que estão sendo exibidos na tela ou quando está sendo enviado o outro tipo de saída (output).

O comando ASSIGN, por exemplo, possui a origem dos dados no SCREEN BUFFER e o destino dos dados no RECORD BUFFER. Com isso podemos concluir que o comando ASSIGN pega os valores que estão sendo exibidos na tela, ou em qualquer outra saída e joga estes dados para o RECORD BUFFER. Consequentemente estes dados serão armazenados no DATABASE ao término da transação (RELEASE).

Já o comando DISPLAY, possui sua origem no RECORD BUFFER e seu destino no SCREEN BUFFER. Então podemos concluir que este comando pega os dados que estão disponíveis no RECORD BUFFER e joga isso na tela ou em qualquer outra saída, SCREEN BUFFER.

Agora vejamos o comando FIND. Observe que sua origem é a DATABASE e seu destino o RECORD BUFFER. Com isso, podemos concluir que para o comando DISPLAY poder mostrar a informação na tela, precisa dos dados disponíveis no RECORD BUFFER, não é mesmo ?? Pois então, o FIND é um dos comandos que pega a informação no DATABASE e deixa disponível no RECORD BUFFER.

Observando mais profundamente, podemos notar que o comando mais completo neste aspecto é o INSERT, pois ele vai até o DATABASE, pega a informação, deixando-a disponível (RECORD BUFFER), joga na tela para exibição (SCREEN BUFFER) e ainda deixa o usuário alterar o conteúdo das informações e depois faz o caminho inverso, parando no RECORD BUFFER. Porque só o RELEASE possui a tarefa de levar os dados do RECORD BUFFER para o DATABASE.

Segue abaixo um quadro onde exemplifica os comandos que fazem leitura de dados e qual o seu comportamento em relação a questão das áreas de armazenamento.

Os comandos são:

- **ASSING**
- **CREATE**
- **DELETE**
- **DISPLAY**
- **ENABLE**
- **FIND**
- **FOR EACH**
- **GET**
- **INSERT**
- **PROMPT-FOR**
- **RELEASE**
- **SET**
- **UPDATE**

Comando	Banco de Dados	Buffer de Registro	Buffer de Tela	Usuário
ASSIGN			←●	
CREATE	●→			
DELETE	←●			
DISPLAY		●→		
ENABLE			←●	
FIND	●→			
FOR EACH	●→			
GET	●→			
INSERT	●→	●→	←●	←●
PROMPT-FOR			←●	←●
RELEASE	←●			
SET		←●	←●	←●
UPDATE		●→	←●	←●

## Capítulo 11: OTIMIZAÇÃO

Capítulo 11 – Otimização

- 1 • Utilização do IF
- 2 • Utilização de Blocos
- 3 • Utilização do Case
- 4 • Utilização do Can-Find
- 5 • Utilização de Temp-Tables
- 6 • Variáveis Lógicas
- 7 • Redução de Transações
- 8 • Parâmetro No-Undo
- 9 • Respeite o Índice

LHC

Neste capítulo segue algumas dicas rápidas de performance que podem ser aplicadas no dia-a-dia. As regras são simples e seus efeitos, isoladamente, podem não apresentar

grandes resultados, entretanto, seguindo a uma maior quantidade de regras, a melhoria de performance é considerável.

Muitos dos problemas envolvem uso de comandos que causam impactos, enquanto outros problemas são causados por má utilização de comandos existentes, isto é, usar um comando que se substituído por outro melhora performance, podendo até diminuir consumo de memória.

Um fator importante a ressaltar, é que a performance não é resultado apenas de um comando mal aplicado, mas, também, por um consumo elevado de memória. Dependendo do tipo de programação, o arquivo compilado poderá crescer, aumentando então, o consumo de memória. A seguir, serão descritas as dicas para melhoria de performance:

## UTILIZAÇÃO DO IF

O comando IF é um dos piores no ponto de vista de performance, entretanto, sua utilização é inevitável. Porém, é possível, em alguns pontos, fazer um melhor uso deste comando. Caso não seja possível substituir o comando IF por outro comando (CASE por exemplo), por motivos de lógica de programação, o uso adequado já auxilia na performance. A substituição dos comandos IF'S colocados um após o outro por uma encadeação do comando pode melhorar de maneira considerável a performance, isto é:

Exemplo 1:

### Trocar:

```
IF <condição 1> THEN  
DO:  
    COMANDOS...  
END.
```

```
IF <condição 2> THEN  
DO:  
    COMANDOS...  
END.
```

```
IF <condição 3> THEN  
DO:  
    COMANDOS...  
END.
```

```
IF <condição 4> THEN  
DO:  
    COMANDOS...  
END.
```

```
IF <condição 5> THEN  
DO:  
    COMANDOS...
```

END.

Por:

```
IF <condição 1> THEN
  DO:
    COMANDOS...
  END.
ELSE
  IF <condição 2> THEN
    DO:
      COMANDOS...
    END.
  ELSE
    IF <condição 3> THEN
      DO:
        COMANDOS...
      END.
    ELSE
      IF <condição 4> THEN
        DO:
          COMANDOS...
        END.
      ELSE
        IF <condição 5> THEN
          DO:
            COMANDOS...
          END.
```

Desta maneira, as condições 2, 3, 4 e 5 só serão executadas caso necessário, enquanto no primeiro caso, todas as condições são executadas.

É bom levar em consideração que este exemplo depende, logicamente, da necessidade do programa. Se for possível fazer a troca deste IF por um comando CASE, a performance pode ser melhorada.

## UTILIZAÇÃO DE BLOCOS

A utilização de blocos para agrupar comandos, pode aumentar desnecessariamente o arquivo compilado (RCODE), diminuindo assim a performance do produto.

Exemplo 1:

**Trocar:**

```
IF <condição> THEN
DO:
    ASSIGN VAR1 = <valor 1>.
    ASSIGN VAR2 = <valor 2>.
    ASSIGN VAR3 = <valor 3>.
END.
ELSE
DO:
    ASSIGN VAR1 = <valor 4>.
    ASSIGN VAR2 = <valor 5>.
    ASSIGN VAR3 = <valor 6>.
END.
```

**Por:**

```
IF <condição> THEN
    ASSIGN VAR1 = <valor 1>
    VAR2 = <valor 2>
    VAR3 = <valor 3>.
ELSE
    ASSIGN VAR1 = <valor 4>
    VAR2 = <valor 5>
    VAR3 = <valor 6>.
```

É importante ressaltar que neste caso, existe apenas um comando após o THEN e/ou ELSE e podemos omitir o DO: END.

Outra coisa que podemos fazer é aglutinar em um único ASSIGN as três atribuições que existiam no programa, com isso é um único processamento e não a operação de três comandos.

Entretanto, é necessário tomar cuidado, quando utilizar uma INCLUDE como código dentro de um possível bloco. Como nem sempre é possível saber o que há dentro da INCLUDE, por precaução, é aconselhável utilizar o bloco DO/END.

Exemplo 2:

**Trocar:**

```
DO WITH FRAME {&FRAME-NAME}:
    ASSIGN VAR1 = <valor 1>
    VAR2 = <valor 2>
    VAR3 = <valor 3>.
END.
```

**Por:**

```
ASSIGN VAR1 IN FRAME {&FRAME-NAME} = <valor 1>
```

**VAR2 IN FRAME {&FRAME-NAME} = <valor 2>**  
**VAR3 IN FRAME {&FRAME-NAME} = <valor 3>.**

O motivo para esta troca é o mesmo do anterior, diminuição de arquivo compilado (RCODE), diminuindo então utilização de memória. Neste exemplo, é comum encontrar programas escritos desta maneira para “economizar” a digitação do parâmetro **IN FRAME {&FRAME-NAME}**.

## UTILIZAÇÃO DO CAN-FIND

A utilização do comando CAN-FIND retorna TRUE ou FALSE quando utilizado, isto é, existe apenas a verificação se o registro existe ou não. O registro pesquisado não fica habilitado para nenhuma operação, nem mesmo para leitura. Ao contrário do comando FIND que pode tornar o registro habilitado até mesmo para escrita.

O principal ganho para a melhoria da performance, é que o comando CAN-FIND retorna apenas um valor lógico, enquanto o comando FIND busca todo o registro, ou apenas os campos especificados para o CLIENT, isto é, aumenta o tráfego de dados na rede fazendo muito I/O e, várias vezes, desnecessariamente.

A principal funcionalidade do comando CAN-FIND é a necessidade de saber se um registro existe ou não, como por exemplo, em um cadastro. Ao verificar se a chave informada já existe, não é necessário trazer todo o registro, somente a informação de sua existência.

Exemplo 1:

### Trocar:

```
FIND <tabela> WHERE
    <tabela>.<campo-chave> = <VALOR>
no-lock.
```

```
IF NOT AVAILABLE <tabela> THEN
DO:
    CREATE <tabela>.
    ASSIGN <tabela>.<campo-chave>.
END.
```

```
ASSIGN <tabela>.<demais-campos>.
```

Exemplo 2:

### Por:

```
IF NOT CAN-FIND(<tabela> WHERE
    <tabela>.<campo-chave> = <VALOR>) THEN
DO:
    CREATE <tabela>.
    ASSIGN <tabela>.<campo-chave>.
END.
```



ASSIGN <tabela>.<demais-campos>.

## VARIÁVEIS LÓGICAS

As comparações realizadas com variáveis lógicas em comandos, como por exemplo, as condições em um comando IF, são mais rápidas se não utilizam o sinal de igualdade para expressar a condição desejada.

Exemplo 1:

### Trocar:

```
IF <variável ou campo lógico> = YES THEN DO:  
  Comandos ...  
END.
```

Exemplo 2:

### Por:

```
IF <variável ou campo lógico> THEN DO:  
  Comandos ...  
END.
```

## REDUÇÃO DAS TRANSAÇÕES

A redução de transações significa mais memória livre, pois a quantidade de registros bloqueados é menor. Entretanto, para diminuir as transações, existe um aumento na complexidade da codificação dos programas.

Será descrito a seguir algumas dicas para diminuir transações:

1) Criar sub-transações:

Para cada agrupamento de comandos que façam alteração de informações em um banco de dados, é criado uma transação no escopo do bloco atual que pode ser o programa como um todo ou um laço(DO TRANSACTION: ... END.).

Exemplos:

### Uma grande transação:

```
... /* Início do programa – Não acessa o banco de dados */
```

```
DO: <var1> = 1 TO 1000:
```

```
  CREATE <tabela>.
```

```
  ASSIGN <tabela>.<campo> = <VALOR>.
```

```
END.
```

```
... /* Finalização do programa – Não acessa o banco de dados */
```

**Observação:** A transação deste programa fica no escopo do programa (DO: ... END.) e bloqueia 1000 registros. Para isto é consumida uma quantia de memória. É importante ressaltar que um simples comando DO não é suficiente para gerar uma transação. Caso só exista ele, como é o caso, o escopo da transação é o programa como um todo.

**Uma pequena transação:**

```
... /* Início do programa – Não acessa o banco de dados */
DO <var1> = 1 TO 1000:
  DO TRANSACTION:
    CREATE <tabela>.
    ASSIGN <tabela>.<campo> = <VALOR>.
  END.
END.
... /* Finalização do programa – Não acessa o banco de dados */
```

**Observação:** A transação deste programa é menor, pois fica no escopo da sub-transação (DO TRANSACTION: ... END.) e bloqueia apenas 1 registro. Para isto é consumido menos memória do que no exemplo anterior.

2) Utilizar o comando DO para laços (LOOP) simples ao invés do comando REPEAT:

É considerado um laço simples, aquele que não tiver envolvimento com transações. O comando REPEAT cria por default uma transação:

**Transação desnecessária:**

```
... /* Início do programa */
REPEAT <VAR> = 1 TO 100:
  ASSIGN <variável> = <VALOR>.
  DISPLAY ...
END.
... /* Finalização do programa */
```

**Observações:** Foi criado desnecessariamente transações, ocupando então memória.

**Correção:**

```
... /* Início do programa */
DO <VAR> = 1 TO 100:
  ASSIGN <variável> = <VALOR>.
  DISPLAY ...
END.
... /* Finalização do programa */
```

**Observação:** Não foi criada nenhuma transação, consumindo menos memória.

## PARÂMETRO NO-UNDO

Variáveis declaradas como NO-UNDO, não necessitam ser controladas pelo PROGRESS®, isto é, quando não é utilizado o parâmetro NO-UNDO, o arquivo \*.LBI (local before image – este arquivo está no Client) necessita controlar os valores armazenados pela variável. Isto aumenta I/O diminuindo a performance.

Normalmente, as variáveis não precisam ser controladas caso haja problemas com o sistema, sendo assim, não é necessário ficar armazenando seus valores.

## RESPEITE O ÍNDICE

Ao fazer qualquer leitura no banco, sempre respeite o índice, porque este simples fato, se ignorado pode acarretar em uma enorme perda de performance em seu programa.

O PROGRESS® possui algoritmos bastante eficientes para encontrar o melhor índice a ser utilizado, mas se você puder, dê uma mãozinha. Você pode utilizar o USE-INDEX <nome\_do\_índice> na hora da pesquisa.

Antes de fazer suas pesquisas, entre no Data Dictionary e veja os campos que fazem parte do índice, observe a ordem dos campos, se o índice é único ou não, se é primário ou não, se a ordem em que foi ordenado é crescente ou decrescente. Todos estes fatores fazem com que sua pesquisa se torne muito rápida.

Não crie índices demais para suas tabelas, pois pense que, a cada atualização de registros, o PROGRESS® terá que atualizar, também, sua tabela de índices, e quanto mais índices definidos, mais tempo para atualizar um registro.

Muitos campos no índice também podem atrapalhar e só servir para um determinado tipo de pesquisa, tente analisar, na hora de criar um índice, pois muitas vezes é necessário criar um campo-chave sequencial na tabela para facilitar o relacionamento, nesta hora podemos ter a ajuda das SEQUENCES.