Talisman

Versão 4.40

Manual de programação

Arndt von Staa

LES/DI/PUC-Rio

Novembro 1996

Ambiente de Engenharia de Software Assistido por Computador Talisman

Manual de Programação

Versão 4.4

Outubro 1996

LES/DI/PUC-Rio

Rio de Janeiro, RJ, Brasil

Sobre este manual

Este manual tem por objetivo apresentar como programar o ambiente Talisman. Através dos programas de formulários pode-se configurar Talisman de modo a atender a maior parte das necessidades dos desenvolvedores de software.

Sumário

1. APRESENTAÇÃO

Este manual descreve a linguagem de programação utilizada internamente por Talisman. Os programas escritos nesta linguagem são chamados *programas de formulário*. Eles são utilizados, entre outros, para definir a organização e o conteúdo das especificações dos diversos objetos manipulados.

Uma das características marcantes de Talisman é a sua flexibilidade. Esta flexibilidade é alcançada através da possibilidade de se programar as ações realizadas com o apoio do ambiente. As seguintes ações de usuários são processadas através de programas de formulários redigidos pelo usuário do ambiente:

- *criar* objetos contidos em dicionários de classes de objetos.
- *criar* ou *instanciar* elementos gráficos ao editar diagramas.
- *editar* e/ou *exibir* especificações de objetos contidas na base de software.
- *explorar* e/ou *inspecionar* o conteúdo da base de software.
- *validar* o conteúdo da base de software.
- *imprimir* relatórios.
- *linearizar* e/ou *prototipar*, gerando arquivos utilizáveis por outros processadores, por exemplo, compiladores.
- *transformar* o conteúdo da base de software, gerando novas especificações, novos objetos, novos atributos de objeto, e/ou complementando as informações relativas a objetos já existentes.
- *exportar* especificações, gerando arquivos utilizáveis por outros ambientes e/ou outros membros da equipe de desenvolvimento.
- *complementar* ações internas. Diversas ações, por exemplo, a edição do corpo de macros, também são realizadas por programas de formulários.

Talisman é configurável o que torna variável a relação de palavras reservadas, nomes de classes de objetos, nomes de dicionários, nomes de relações, propriedades de relações etc. O arquivo `LISTAREL.LST` é gerado a partir do conteúdo da base de conhecimento. Consulte este arquivo para obter as informações mais recentes sobre todos os elementos configuráveis de Talisman.

2. ASPECTOS CONCEITUAIS E OPERAÇÃO DO COMPILADOR

Nesta seção serão apresentados diversos aspectos conceituais. Serão descritos ainda os diversos procedimentos de operação do ambiente Talisman tanto para programar como para utilizar estes programas de formulários.

2.1 O Administrador de Ambiente

O uso indisciplinado da programação de formulários pode trazer problemas. Por exemplo, se os programas de formulários forem alterados sem obedecer a um padrão válido para toda a instituição, podem surgir conflitos de comunicação entre os membros da equipe. Para evitar estas dificuldades, designe uma pessoa para a função de *Administrador do Ambiente*.

O Administrador de Ambiente será o responsável pela programação de formulários. Conseqüentemente, será o responsável, e deverá ter a necessária autoridade, pelo estabelecimento dos padrões técnicos utilizados por todas as equipes usuárias de Talisman. Desta forma serão eliminadas as dificuldades decorrentes de bases de software com organizações conflitantes. Serão reduzidas, ainda, as dificuldades de comunicação decorrentes da falta de padronização na documentação técnica gerada.

O Administrador de Ambiente deverá ter um bom conhecimento de metodologias, de normas técnicas e de processos de desenvolvimento, além, é claro, de ter bom conhecimento de Talisman.

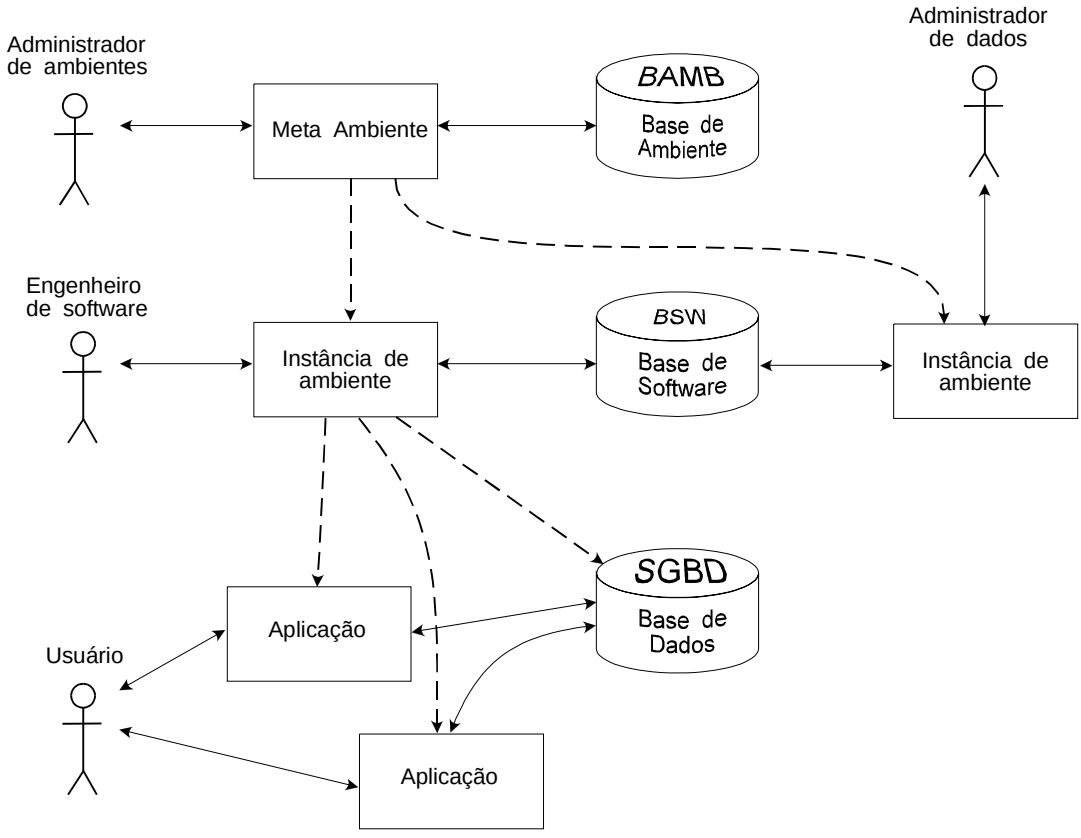

Figura 2.1. Hierarquia de funções

Na figura 2.1 ilustramos a hierarquia de funções das pessoas que direta ou indiretamente interagem com o ambiente Talisman. O *Administrador de Ambiente* interage com o ambiente criando e/ou editando os programas de formulário fonte a serem compilados e, posteriormente, utilizados pelo Engenheiro de Software (analistas, projetistas, programadores, líderes de projeto, etc.). Desta forma o *Administrador de Ambiente* estabelece os padrões a serem obedecidos por toda a equipe.

Ao *Administrador de Dados* cabe a responsabilidade de padronizar as definições e os usos dos diferentes dados, arquivos e bancos de dados. Estas definições podem ser registradas em uma base de software raiz. Esta base de software é distribuída a todas as equipes quando do início de um novo desenvolvimento. Através da importação, estas definições podem ser passadas para as bases dos diversos sistemas a serem desenvolvidos. O *Engenheiro de Software* interage com o ambiente já instanciado, criando e editando a base de software. Esta edição é controlada

por programas de formulário já compilados. Ao editar a base de software, ele poderá lançar mão das especificações de dados definidas pelo *Administrador de Dados*. A partir da base de software, o *Engenheiro de Software* gera os diversos aplicativos. Estes, finalmente, são utilizados pelo usuário.

Pode-se ampliar a função do *Administrador de Dados*, passando a ele o controle sobre fragmentos de código. Estes podem ser rotinas de biblioteca, bem como, texto fonte a ser incorporado aos programas gerados por intermédio de programas de formulários.

Resumindo, temos as seguintes atribuições:

Administrador de Ambiente — define e padroniza o processo de desenvolvimento, inclusive o formato e composição das representações.

Administrador de Dados — define e padroniza os dados, inclusive a forma de manipulá-los.

2.2 Glossário específico do manual

Ação de usuário — é uma operação do ambiente selecionada pelo usuário. Diversas ações de usuário são efetuadas por intermédio de programas de formulários. São exemplos de tais ações: criar um novo objeto, editar a especificação de um objeto, validar um objeto, imprimir um objeto, etc.

Atributo — é um valor associado a um objeto. Atributos são organizados em *categorias de atributos*. Estas categorias são, entre outras: *nomes, aliases, fragmentos de texto* e *relações* com outros objetos. Os programas de formulário permitem criar, alterar, formatar, editar e excluir os atributos dos diferentes objetos.

Campo — é uma unidade de exibição e, possivelmente, edição. A edição de um formulário se dá através da edição dos campos que o constituem. Campos podem ser: um texto constante: *título*, um valor associado a um objeto: *atributo*, ou uma *variável*.

Elemento — é um objeto ou uma instância de objeto.

Elemento corrente — é o objeto, ou instância de objeto, sobre o qual se está operando em um determinado ponto do programa de formulários.

Elemento origem — é o objeto, ou instância de objeto, corrente quando da ativação de uma ação de usuário. O elemento origem é o elemento a partir do qual a base de software é *explorada* para gerar o formulário editável, o texto gerado, o relatório impresso, ou o arquivo linearizado.

Exploração — é o resultado de percorrer parte da base de software, operando sobre os atributos dos objetos e instâncias de objetos contidos neste percurso. Ao explorar a base de software, são examinados formulários e diagramas. A partir destes, pode-se *navegar* para outros diagramas e/ou formulários.

Formulário — é uma unidade elementar de um programa de formulário. Intuitivamente, formulários correspondem a sub-programas (pacotes, *procedures*, etc.). Formulários contém *campos* que permitem *editar*, *criar*, *explorar*, *exibir*, *manipular* e *transformar* os atributos contidos na base de software, *manipular* variáveis e, finalmente, definir como *formatar* os atributos e as variáveis a serem editados, exibidos, gravados ou impressos. Formulários são redigidos na linguagem de programação descrita neste capítulo.

Formulário editável — exibe e torna editáveis os campos resultantes de uma extração da base de software. É o resultado da aplicação do *formulário origem* a um *elemento origem* em uma ação de edição ou criação de objetos, sendo o resultado da execução exibida no vídeo. É o formulário com o qual o usuário interage ao editar o conteúdo da base de software. Um formulário editável é, em geral, uma representação da especificação ou da implementação (código fonte) de um determinado elemento. Formulários editáveis são editados pelo **Editor de Formulários**.

Formulário origem — é o formulário inicial ativado em resposta a uma ação de usuário.

Instância de objeto — é a ocorrência de uma referência a um objeto, onde esta referência está contida em um diagrama. Um determinado objeto pode ser instanciado em vários diagramas. Cada uma destas instâncias referencia este mesmo objeto. Algumas classes de objetos gráficos não possuem especificação. Por exemplo, a cardinalidade de uma relação entre entida-

des (linguagem Entidades e Relacionamentos) não possui especificação. Ou seja, instâncias puras pertencem a classes não dicionarizadas. Conseqüentemente, para estas classes de objetos existirão somente as instâncias de objeto.

Navegação — é a operação de caminhar do objeto corrente para outro. Através da navegação pode-se, inclusive, caminhar de um objeto para uma instância e vice-versa. Em geral a navegação é realizada caminhando-se sobre relações entre objetos e/ou instâncias de objetos. A navegação pode ser realizada, ainda, explorando o conteúdo de todo um dicionário, ou caminhando diretamente para um objeto identificado através de uma variável, lista ou de uma função.

Objeto — é um elemento de uma classe de componentes de uma linguagem de representação. Por exemplo, o objeto "Cadastro de Clientes" pertence à classe *Depósito de Dados*. Objetos são agrupados em dicionários. Para cada classe de objetos existe um dicionário. Evidentemente, excetuam-se as classes de objeto não dicionarizadas, cujos elementos aparecem, exclusivamente, sob a forma de instâncias de objeto.

Programa de formulário — é um conjunto de formulários apoiando uma ou mais ações do usuário.

Texto gerado — é uma extração da base de software dirigida para um arquivo, uma impressora, um atributo **Texto**, ou uma variável tipo **ListaTexto**. Um texto gerado é idêntico a um formulário editável, diferenciando-se somente quanto ao destino. Esta propriedade permite utilizar programas de formulário de forma intercambiável tanto para gerar formulários editáveis como para produzir textos gerados.

2.3 Interação entre programas e base de software

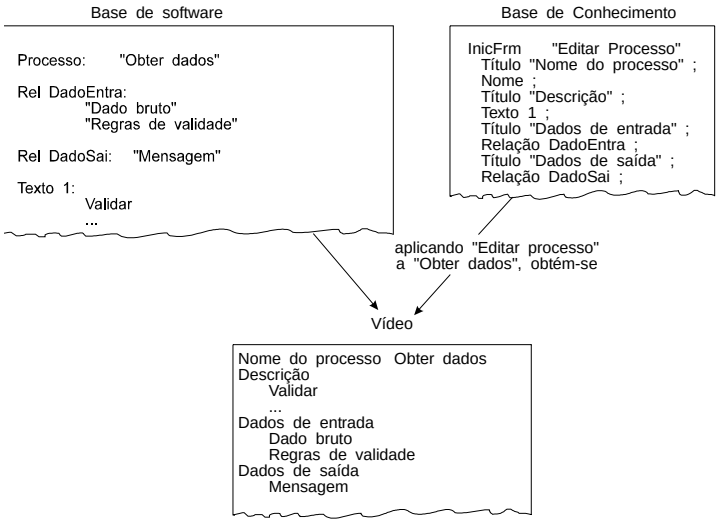

Figura 2.2. Ilustração da composição de um formulário editável

A figura 2.2 ilustra a composição de um formulário editável a partir de um programa de formulário (o formulário *Editar Processo*) e de um objeto (o processo *Obter dados*). O programa de formulários reside na base de conhecimento. Os atributos do objeto residem na base de software. O correspondente formulário editável é exibido no vídeo, podendo ser editado pelo usuário. Todas as edições realizadas serão transferidas para a base de software ao terminar a edição do formulário editável (comandos: [F3], [^F3], [F4], [F6] e [^F6]). Para maiores detalhes quanto à edição de formulários, consulte o capítulo **Editor de Formulários**.

2.4 Ambiente de programação

Nesta seção será descrita a interação dos programas de formulários e o ambiente Talisman. Será descrita ainda a seqüência de trabalho a ser utilizada para criar, editar e configurar o uso de programas de formulários.

Programas de formulários existem em dois formatos:

programas fonte — são arquivos seqüenciais criados e mantidos através de algum editor de texto, usualmente o próprio ambiente Talisman.

programas executáveis — são armazenados na base de conhecimento e são gerados pelo compilador de programas de formulário.

Talisman contém um *ambiente de programação* de formulários composto por um *editor*, um *compilador* e um *interpretador* de programas de formulário, ver figura 2.3. Com o editor são gerados e mantidos os arquivos de programas fonte. Estes arquivos são apresentados ao editor via o compilador. Com o compilador são gerados, a partir dos programas fonte, os programas executáveis contidos na base de conhecimento. Os programas executáveis operam na *máquina virtual* Talisman. Esta máquina virtual é reificada pelo interpretador. Estes três componentes formam um todo integrado, assegurando que erros sintáticos e/ou de execução sejam explicitamente vinculados ao programa fonte.

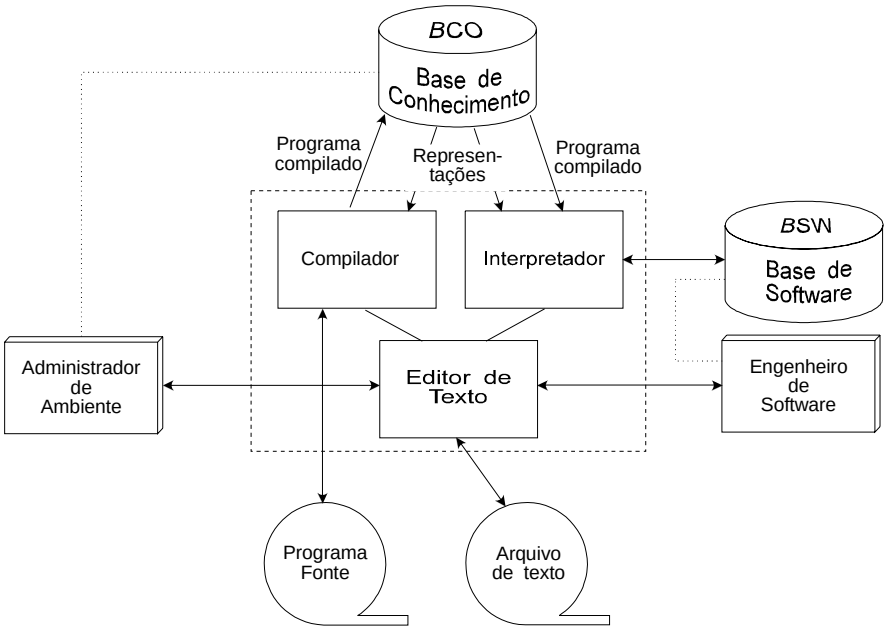

Figura 2.3. Componentes do ambiente de programação

Os programas fonte podem ser editados, alternativamente, usando editores quaisquer, podendo, inclusive, ser gerados, por intermédio de linearizadores a partir de bases de software contendo projetos de programas de formulários. Por exemplo, os programas de formulários `TALISESP.FRM` e `TALISLIN.FRM` permitem editar e, posteriormente, linearizar, bases de software contendo programas de formulários. Para compilar programas de formulários gerados, basta referenciar os arquivos correspondentes ao ativar o ambiente de programação. Isto permite a criação de bibliotecas de programas de formulários que serão utilizados por todos os membros das diversas equipes, assegurando, assim, a uniformidade e a padronização dos diversos ambientes em uso.

Para cada linguagem de representação são definidos os programas de formulários:

Especificação — contém todos os formulários utilizados para criar e editar as diversas classes de objetos da correspondente linguagem de representação.

Validação — contém todos os formulários utilizados para validar os diversas classes de objetos da correspondente linguagem de representação.

Impressão — contém os formulários utilizados para gerar relatórios.

Transformação — contém os formulários utilizados para gerar e/ou alterar o conteúdo da base de software a partir de representações já definidas.

Linearização — contém os formulários utilizados para gerar código e/ou protótipos a partir do conteúdo da base de software.

Exportação — contém os formulários utilizados para gerar arquivos ASCII de exportação para outros ambientes, ou outras estações Talisman.

Em adição a estes programas, está definido, ainda, o programa Serviços. Na atual versão, este se destina a editar o corpo de macros de teclado.

Os programas de formulário originais são distribuídos sob a forma de arquivos ASCII (nome de extensão `.FRM`). A maioria destes programas já vem instalados “de fábrica”. No caso de algum programa não ter sido instalado,

ou caso queira retornar aos programas originais, utilize o serviço Instalação de formulários no menu **Serviços**.

2.5 Procedimento de configurar programas de formulário

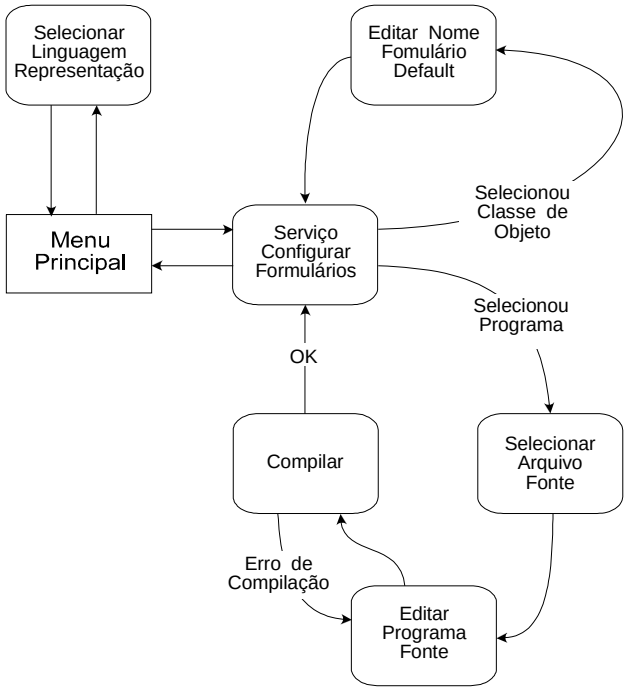

Figura 2.4. Procedimento de configuração de programas de formulário

A figura 2.4 ilustra o procedimento (seqüência de ações) a ser efetuado para editar e compilar um programa de formulário. A figura ilustra ainda como associar nomes de formulário a ações de usuário.

Cada programa de formulário está associado a um conjunto de uma ou mais ações que podem ser ativadas pelo usuário. Por exemplo, ao operar com a linguagem de representação *Documentação* e estando posicionado sobre um elemento da classe *parágrafo*, as ações [F4], *Editar especificação*, e [F5], *Criar objeto*, ativarão, respectivamente, os formulários *Editar parágrafo* e *Criar parágrafo*, ambos contidos no programa de formulário *Especificação* da linguagem de representação *Documentação* (programa `DOCESP.FRM`). Já a ação [F6 V], *Validar especificação*, ativará o formulário *Validar parágrafo* contido no programa *Validação* da linguagem de representação *Documentação* (programa `DOCVAL.FRM`).

Para cada ação do usuário, para cada classe de objetos e para cada linguagem de representação suportada, são definidos *formulários default*. Estes formulários serão ativados sempre que a ação correspondente venha a ser ativada pelo usuário. Por exemplo, se, ao editar uma estrutura de dados, for ativada a ação Editar especificação (comando [F4]), será ativado o formulário associado com a linguagem de representação corrente e a classe de objetos estrutura de dados. A associação de ações com formulários é efetuada por intermédio do serviço configurar Formulários. Este mesmo serviço permite a edição e compilação dos diversos programas de formulário.

Para associar e/ou programar formulários, selecione primeiro a linguagem de representação. Isto é realizado no Menu Principal. Após, ative o serviço configurar Formulários. No serviço de configuração de formulários podem ser selecionadas as ações relativas às quais se deseja reassociar os formulários *default*. Selecionada uma das ações, são exibidas as associações definidas para cada uma das classes de objetos da linguagem de representação corrente. Na classe de objeto desejada, digite o nome do formulário *default* a ser utilizado. Neste mesmo serviço pode-se, ainda, ativar o ambiente de programação relativo aos diversos programas de formulários da linguagem de representação corrente.

Ao redigir os programas de formulário ou ao associar formulários *default* com ações, deve-se assegurar que os correspondentes formulários *default* estejam definidos em ambos os lugares. A correspondência se dá pelo nome do formulário. Ao procurar o nome do formulário *default* no programa, letras maiúsculas e minúsculas, acentuadas ou não, serão consideradas iguais.

Ao executar uma ação de usuário a partir do Editor de Dicionário, pode-se utilizar o formulário *default* ou, um formulário selecionado. Ao ativar uma ação selecionando formulários, são exibidos todos os programas de for-

mulário selecionáveis no programa de formulário associado à ação na linguagem de representação corrente. Os formulários selecionáveis são definidos no programa de formulário, utilizando-se a declaração **InicExterno** ao invés da declaração **InicFrm**.

2.6 Processo de seleção do formulário origem

Ao efetuar uma ação de usuário, será executado primeiro o formulário global, caso exista no programa. Após será executado o formulário origem. Como visto na seção anterior, este poderá ser o formulário *default* associado com a ação, ou, então, será um formulário selecionado.

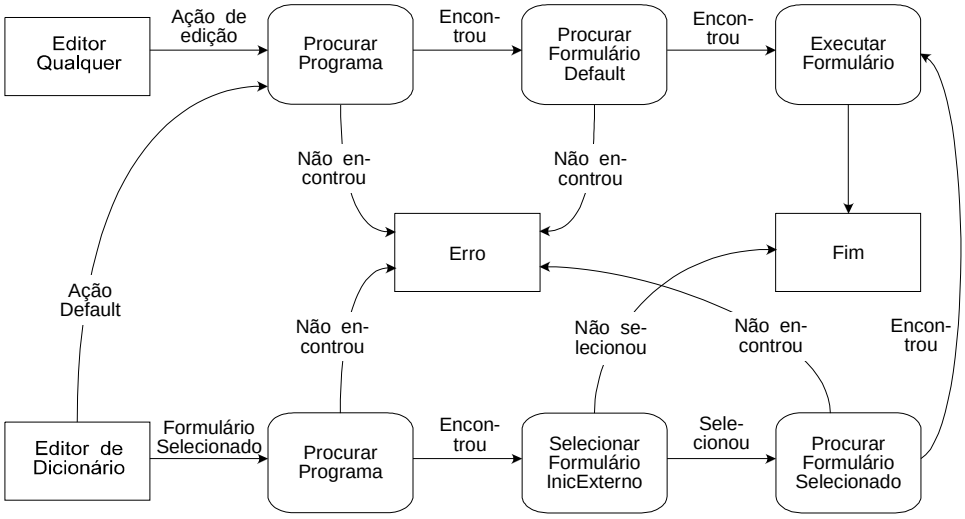

Figura 2.5. Processo de seleção do formulário origem

A figura 2.5 ilustra o processo de ativação do formulário origem. Sempre que um programa de formulário for ativado a partir de um editor que não seja o Editor de Dicionários, será procurado o formulário *default* correspondente ao objeto corrente naquele editor. Caso a ativação se dê a partir do Editor de Dicionários, pode-se optar, ou por utilizar o formulário *default* ou um formulário selecionado (cabeçalho definido com **InicExterno**).

Mais precisamente, ao efetuar uma ação de usuário, são realizados os seguintes passos:

- busca do programa de formulário da linguagem de representação corrente associado com esta ação. Será emitida uma mensagem de erro caso este programa não esteja definido.
- busca do formulário origem (*default* ou selecionado, conforme o caso) neste programa de formulário. Procura-se no programa de formulários pelo nome simbólico do formulário origem. Caso não seja encontrado, será emitida uma mensagem de erro. A associação de formulários com ações é feita conforme descrito na seção anterior.
- ativação do formulário global, caso exista.
- ativação do formulário origem.

Por exemplo, se a ação Editar Especificação de Documento estiver associada ao formulário *Editar documento*, ao ativar esta ação com relação a um determinado documento, será procurado o formulário de nome *Editar documento* no programa de *Especificação*. Caso não seja encontrado, será emitida uma mensagem de erro. A associação de nomes de formulários a ações de usuário poder ser realizada a qualquer momento e é independente da compilação do correspondente programa de formulários.

3. COMPOSIÇÃO DAS LINGUAGENS DE REPRESENTAÇÃO

3.1 Componentes gerais

Cada linguagem de representação apoiada por Talisman possui:

Definição da representação — estabelecida através de *tabelas*, descreve todas as classes de objetos, regras de validade estrutural básicas, relações entre objetos e os aspectos gráficos da linguagem de representação. Embora possam ser criadas novas linguagens, relações, classes de objeto editando umas poucas tabelas, na atual versão, a definição de linguagens de representação não é editável pelo usuário.

Definição de especificações — estabelecida através de *programas de formulários*, descreve a composição da especificação de cada classe de objetos da linguagem. Cabe observar que os formulários de especificação podem navegar para objetos criados em outras linguagens de representação.

Regras de validação — definidas através de *programas de formulários*, descrevem como controlar a qualidade das representações redigidas nesta linguagem. A validação pode levar em conta atributos de objetos definidos em outras linguagens de representação.

Linearizadores — definidos através de *programas de formulários*, descrevem como gerar texto seqüencial para uso por algum outro processador (por exemplo compilador). Os linearizadores podem ser utilizados para gerar programas, protótipos, documentos, etc.

Transformadores — definidos através de *programas de formulários*, descrevem como transformar o conteúdo da base de software, gerando e/ou alterando representações a partir das informações já contidas na base de software.

Geradores de relatórios — definidos através de *programas de formulários*, descrevem como produzir texto impresso (documentação do usuário, ou técnica).

Exportadores — definidos através de *programas de formulários*, descrevem como produzir arquivos seqüenciais a serem utilizados em outros ambientes. Através da exportação e importação pode-se transferir informações entre bases de software Talisman, bem como trocar informações com outros ambientes de desenvolvimento. Na prática exportadores são uma forma alternativa de linearizadores.

3.2 Visão geral de programas de formulários

Formulários descrevem como explorar a base de software a partir de um elemento origem para gerar um:

- formulário editável a ser editado pelo usuário.
- texto impresso.
- arquivo seqüencial de texto.
- fragmento de texto a ser atribuído a uma variável ou a um atributo de objeto.
- transformação da base de software.

A interpretação do formulário é realizada pelo interpretador Talisman . Do ponto de vista do interpretador, um formulário é composto por:

Campos — buscam, exibem, editam e/ou guardam valores contidos na base de software.

Variáveis — registram valores simples e conjuntos (listas) necessários para a execução do programa de formulários. Variáveis podem ser armazenadas no programa ou nas diversas bases utilizadas por Talisman.

Expressões — definem procedimentos de cálculo.

Quantificadores — realizam uma seqüência de operações repetidas vezes para todos os elementos de um conjunto, efetuando a navegação para estes elementos.

chamadas de formulário — ativam outros formulários contidos no programa de formulários. Correspondem a chamadas de sub-rotinas. Os formulários podem ser ativados recursivamente.

controles de execução — controlam a seqüência da execução (repetição e seleção) de instruções dos programas de formulário.

controles de formato — definem como organizar a apresentação do formulário editável.

O texto a seguir ilustra um fragmento de programa de formulário:

```
/*************************************
* Ação: Criar/Editar nome de processo
*************************************/
   InicFrm "Criar processo"
      Titulo "Processo: " ;
      NãoAvLin ;
      Nome ;
   FimFrm

/*************************************
* Ação: Especificar processo
*************************************/
   InicFrm "Editar processo"
      Frm "Criar processo" ;
      Título "Descrição informal" ;
      Texto TxtDescr ;
      Título "Especificação formal" ;
      Texto 10 ;
      Título "Agentes responsáveis pelo processo" ;
      Relação Agentes ;
   FimFrm
```

O formulário *Criar processo* é utilizado para editar ou criar o nome de um processo. O formulário *Editar processo* é utilizado para a edição da especificação de um processo.

A instrução **InicFrm** "Criar processo" inicia a definição do código do formulário *Criar processo*. O formulário termina na primeira instrução **FimFrm** a seguir.

As instruções **Título** definem textos constantes que aparecerão no formulário editável. Os títulos explicam ao usuário o significado de cada um dos campos do formulário editável. Cabe salientar que, em geral, os atributos contidos na base de software não possuem uma interpretação pré definida. Ou seja, é o uso que se faz dos atributos que define a sua semântica. Para estabelecer uma comunicação correta com o usuário, utilize títulos que denotem precisamente o significado do campo editável a seguir.

A instrução **NãoAvLin** determina que o próximo campo a ser exibido deve ficar na mesma linha que o último campo exibido.

A instrução **Nome** busca, exibe, edita e guarda o nome do objeto corrente.

A instrução **Frm** "Criar processo" ativa o formulário *Criar processo*. A utilização de um único formulário para a edição do nome de um objeto contribui para a uniformidade dos formatos de edição.

A instrução **Texto** TxtDescr busca, exibe, edita e guarda o fragmento de texto conhecido pelo identificador de atributo texto TxtDescr (índice 0). A instrução **Texto** 10 busca, exibe, edita e guarda o fragmento de texto de índice 10. Cada objeto admite até 255 diferentes fragmentos de texto. Estes fragmentos são acessados por um índice. O valor deste índice pode ser o nome de um atributo, por exemplo TxtDescr, ou pode ser o resultado da avaliação de uma expressão inteira, por exemplo 10. Esta, por sua vez, pode lançar mão de constantes simbólicas.

O significado (semântica) de um fragmento de texto é definido pelo programa de formulário. A base de software trata todos os fragmentos de texto de forma igual. Para evitar confusão e incompatibilidade entre bases de software, cada instalação deve *padronizar* os fragmentos de texto definidos para cada uma das classes de elementos. No exemplo, o fragmento de texto de índice 0 representa uma descrição informal do processo sendo editado. Já o fragmento de texto de índice 10, representa uma descrição formal do processo sendo editado. Tanto o significado de cada fragmento de texto, como as notações a serem utilizadas ao preencher os correspondentes campos, devem ser descritas em um documento produzido pelo *Administrador de Ambientes*.

A instrução **Relação Agentes** busca, edita e guarda a relação dos agentes relacionados com o processo sendo editado. Neste exemplo esta relação identifica todos os agentes responsáveis pelo processo sendo editado.

4. INTERPRETAÇÃO DO PROGRAMA

A cada momento existe um *elemento corrente* que estará sendo processado. Este elemento possui um *tipo do elemento corrente*. O tipo do elemento corresponde à *classe do elemento corrente*. Ao operar com um atributo ou uma relação do elemento corrente, deve-se tomar o cuidado de que este seja sempre válido, considerando o tipo do elemento corrente. Cabe salientar, ainda, que objetos e instâncias de objetos são distintos. Assim o tipo do elemento corrente deve ser entendido como sendo formado tanto pelo tipo do elemento, como pelo fato de se tratar ou não de uma instância de objeto.

O elemento corrente inicial é o *elemento origem* ao qual é aplicado o formulário origem. No interior de um quantificador, o elemento corrente será o elemento gerado a cada ciclo de repetição.

Por exemplo, supondo que o elemento corrente seja uma instância de objeto, a instrução

```
ParaTodos LigEntra Faz
    ...
Fim ;
```

executará tantos ciclos quantas forem as ligações de entrada na instância de objeto corrente. Em cada ciclo, uma destas ligações de entrada será o elemento corrente. No corpo do quantificador, o tipo do elemento será instância de ligação. Note que uma ligação não possui especificação, portanto não existe um objeto associado a uma instância de ligação. Por esta razão pode-se simplificar a terminologia utilizando simplesmente ligação para denotar instâncias de ligação.

A linguagem admite variáveis tipo **objeto** (ou *instância de objeto*) e funções que retornam objetos (ou instâncias de objeto). Valores tipo objeto referenciam objetos ou instâncias de objeto. Pode-se navegar para estes objetos utilizando a construção a seguir:

```
ComObjeto VarObj Faz
   ...
Fim ;
```

No corpo desta construção o objeto corrente é o objeto referenciado pela variável **VarObj**.

É freqüentemente necessário referenciar o objeto corrente, por exemplo ao passar parâmetros. Isto pode ser realizado pela "constante" **Corrente**. Esta constante assume, continuamente, o valor da referência ao objeto corrente. Ao navegar para objetos pode ocorrer que o objeto corrente se torne nulo. Isto pode ocorrer, por exemplo, ao navegar de uma ligação para um rótulo de ligação, uma vez que os rótulos de ligações são opcionais. No código a seguir:

```
ExisteObj( Corrente )
```

o resultado será **verdadeiro** se o objeto corrente existir, e será **falso**, caso seja nulo.

5. TRATAMENTO DE ERROS

5.1 Erros de compilação

Erros de compilação provocam o término imediato da compilação. O cursor do editor é colocado sobre o ponto em erro e é exibida uma mensagem explanatória do erro encontrado.

Não é feita distinção entre os diferentes erros de sintaxe que podem ser encontrados. Por exemplo, erros decorrentes do uso indevido de palavras reservadas ou decorrentes do uso de pontuação incorreta, geram a mesma mensagem: Erro de sintaxe. Outros erros, por exemplo, uso de variável não declarada, inconsistência de tipos, etc. produzem mensagens específicas. No caso de erro de sintaxe envolvendo identificadores, verifique se as variáveis e/ou os identificadores padronizados estão sendo utilizadas de forma correta.

5.2 Erros de execução

Erros de execução interrompem o processamento do programa de formulário. O formulário editável, ou o texto gerado, serão gerados parcialmente. No entanto, as edições realizadas na porção gerada serão gravadas na base de software. As mensagens de erro identificam a linha do programa fonte na qual ocorreu o erro. Esta informação, e mais o texto da mensagem são, em geral, suficientes para determinar a causa do erro. Em caso de dúvida, verifique se o tipo do elemento corrente é compatível com a operação sendo realizada.

5.3 Depuração

Sugere-se que a depuração de programas de formulários seja realizada gerando formulários editáveis, mesmo que o programa de formulário em questão tenha outra finalidade. Isto pode ser feito, por exemplo, compilando-se os programas como se fossem exportadores. Ao ativar o programa, seleciona-se a opção *editar*. Após desenvolver o programa de formulário, recompila-se este programa, utilizando a classe apropriada de programas. Como erros de sintaxe e erros de execução interrompem imediatamente a execução, torna-se relativamente simples determinar a causa do erro e, posteriormente, corrigir o programa em erro. Recomenda-se, ainda, que os trechos novos dos programas sejam inseridos no início do código fonte. Desta forma o tempo de compilação é menor, enquanto ainda existirem erros sintáticos.

5.4 Projeto de programas de formulários

Em última análise, programas de formulários são software. Conseqüentemente, devem ser desenvolvidos com o mesmo cuidado que os programas aplicativos.

Recomenda-se fortemente que os programas de formulários sejam projetados e implementados utilizando o próprio Talisman. Para conseguir isto, instale Talisman em dois diretórios diferentes, por exemplo, `TALISMAN` e `TALISAUX`. Na linguagem *Estrutura modular* do Talisman contido no diretório `TALISAUX`, instale os programas de formulários `TALISESP.FRM` para especificar, `TALISVAL.FRM` para validar e `TALISLIN.FRM` para linearizar. Utilizando estes formulários pode-se projetar programas de formulários. Mantenha todos os programas de formulários interrelacionados em uma mesma base de software. Cada módulo corresponderá a um programa específico. Os módulos linearizados são compilados e instalados na linguagem e programa apropriados do Talisman contido no diretório `TALISMAN`. Mantendo-se duas janelas abertas, uma no diretório `TALISMAN` e a outra no diretório `TALISAUX`, facilita projetar, compilar e depurar programas de formulários.

Os programas de formulário `TALISxxx.FRM` estão descritos no capítulo *Linguagem Estrutura Modular*.

Além de projetar o programa de formulários, deve-se, ainda, padronizar os índices dos fragmentos de texto e aliases utilizados. Isto é feito definindo-se constantes simbólicas, agregando-as no formulário global. Estas constantes podem ser copiadas para os diversos programas de formulário, tornando-os indiferentes aos valores específicos dos índices.

6. ELEMENTOS BÁSICOS DA LINGUAGEM

6.1 Convenções de redação

Nesta seção será definido o padrão notacional utilizado no restante deste manual para descrever a gramática da linguagem de programação.

Símbolos terminais são apresentados em negrito, eles devem aparecer no programa fonte e devem ser digitados tal como aparecem. Cabe salientar que letras maiúsculas e minúsculas, acentuadas ou não, são interpretadas da mesma maneira. Exemplos:

InicFrm FimDecl ParaTodos Fim FimFrm

Símbolos não terminais (meta-símbolos) são apresentados em itálico. Símbolos não terminais representam conjuntos de possíveis porções de código. Exemplos:

Comandos_Executáveis Conjunto Para_Todos

Símbolos que correspondem a categorias léxicas são apresentados em fonte normal. As categorias léxicas serão descritas na seção a seguir. Exemplos:

IdentificadorVar IdentificadorFunc

Um não terminal seguido dos caracteres dois pontos e igual (":=") corresponde ao início da definição do conjunto de regras alternativas do não terminal.

Os caracteres de pontuação utilizados pela descrição (meta-pontuação), são apresentados em fonte normal. O caractere barra vertical ("|") simboliza o início de uma regra alternativa. O caractere ponto e vírgula (";") identifica o término de um conjunto de regras de definição de um não terminal.

Os caracteres de pontuação da linguagem de programação de formulários são apresentados em negrito e, em adição, estarão contidos entre aspas.

Exemplo 1:

```
Para_Todos  := ParaTodos Conjunto Faz
                  Comandos_Executáveis
               Fim ;
```

Neste exemplo o símbolo *Para_Todos* denota um não terminal, correspondente à operação de repetição sobre conjuntos definidos. Este símbolo está sendo definido neste fragmento. Os símbolos *Conjunto* e *Comandos_Executáveis* identificam não terminais a serem definidos em algum outro lugar. Os símbolos **ParaTodos**, **Faz**, e **Fim** são símbolos terminais e devem aparecer no código fonte do programa. O caractere ";" indica o término da definição do símbolo não terminal *Para_todos*.

Exemplo 2:

```
Domínio :=   BCO
           | BSW
           | BAX
           | USO ;
```

Neste exemplo o símbolo não terminal *Domínio* pode assumir exatamente um dos quatro valores **BCO**, **BSW**, **BAX** ou **USO**.

Exemplo 3:

```
Parâmetros  :=   "(" Parâmetros_Formais ")"
               | ;
```

Neste exemplo o símbolo não terminal *Parâmetros* pode assumir um valor formado pela seqüência **(***Parâmetros_Formais* **)** ou simplesmente não existir (alternativa vazia). O não terminal *Parâmetros_Formais* deverá ser definido em algum outro lugar indicando como redigir o código correspondente a este não terminal. Os parênteses estão em negrito, o que os caracteriza como símbolos terminais. Conseqüentemente, devem figurar no código fonte do programa.

6.2 Elementos léxicos da linguagem

O texto de um programa de formulários é formado por zero ou mais linhas. Cada linha poderá ter até 250 caracteres. O formato do programa é livre. No entanto, cada unidade da linguagem (constantes, identificadores, seqüências, operadores, etc.), exceto comentários, deve ser integralmente redigido em uma única linha.

Comentário — é uma descrição destinada ao leitor do programa. O texto de um Comentário não participa do código objeto gerado. Comentários podem ser incluídos em qualquer lugar, evidentemente não no interior dos elementos de linguagem. Em particular, no interior de constantes seqüência de caracteres, Comentários serão tratados como caracteres da seqüência. Comentários iniciam com a seqüência de caracteres **/*** e terminam na primeira seqüência de caracteres ***/**. Comentários podem ocupar uma ou mais linhas. No interior de Comentários valem quaisquer caracteres.

Exemplo:

```
/* Isto é um
         Comentário */
```

CteNúmero — é um valor numérico constante inteiro entre -32767 e 32767 inclusive.

Identificador — é uma seqüência de 1 ou mais caracteres, começando por letra e contendo letras, dígitos ou o caractere sublinhado ('_'). As letras podem ser acentuadas, maiúsculas e minúsculas. Ao comparar igualdade de identificadores, letras maiúsculas, minúsculas acentuadas ou não, são tratadas iguais. Os identificadores são subdivididos nas classes: Terminal, IdentificadorVar, IdentificadorFunc, IdentificadorAtrib, IdentificadorRel e IdentificadorDic. Estas classes serão definidas na seção a seguir.

Exemplo: os seguintes identificadores são todos iguais:

```
funcao  Função  FUNCAO  FuNçAo.
```

CteSeqüência — é uma sucessão de 0 ou mais caracteres contidos entre aspas duplas (**"**). No interior de seqüências constantes podem ser incluídos delimitadores de Comentários e/ou caracteres especiais. Caracteres especiais são precedidos pelo caractere barra inversa (****). Os caracteres especiais são:

\" — aspa dupla, inclui o caractere aspa dupla “"“ na seqüência.

\\ — barra inversa, inclui o caractere barra inversa “\” na seqüência.

\nnn — caractere definido através de seu valor *decimal*. Inclui o caractere correspondente ao valor decimal *nnn* na seqüência. Devem ser evitados os caracteres \000 , \001 , \002 , \003, \013 e \026, uma vez que possuem significado interno para alguns dos processadores Talisman.

Exemplos:

```
"ab\"c\\d"      corresponde à seqüência:   ab"c\d
"\097\065"      corresponde à seqüência:   aA
"/*****/"       corresponde à seqüência:   /*****/
```

Cada parcela de constante seqüência deve estar integralmente contida em uma única linha. Em adição, parcelas consecutivas são tratadas como uma única constante seqüência, mesmo que estejam separadas por uma ou mais linhas em branco. No entanto, comentários contidos entre parcelas de seqüências consecutivas, inibem a concatenação automática. A concatenação automática de constantes seqüência, permite gerar seqüências longas a partir da concatenação de seqüências menores. Cabe frisar que seqüências não devem ter mais do que 250 caracteres.

Por exemplo:

```
"abc"     "CDE"
      "ghi"              corresponde à seqüência:   abcCDEghi
```

6.3 Categorias de identificadores

Todos os identificadores definidos pelo ambiente Talisman correspondem a palavras reservadas. Todas estas palavras figuram no arquivo `LISTAREL.LST` fornecido junto com o sistema. Isto permite a consultar as palavras reservadas durante a edição de programas de formulários. A consulta pode ser realizada através da Edição

de Arquivos Seqüenciais, ativado, em qualquer lugar, pelo comando [!Q]. Palavras reservadas não podem ser utilizadas para identificar variáveis ou constantes.

Terminal — é um identificador correspondente a um símbolo terminal do compilador. Todas as palavras correspondentes a símbolos terminais são reservadas, não podendo ser utilizadas para denotar variáveis. O arquivo `LISTAREL.LST` discrimina todos os Terminais definidos na linguagem de programação.

IdentificadorFunc — é o nome de uma das funções disponíveis. As funções disponíveis são descritas na seção Funções Internas deste capítulo. O arquivo `LISTAREL.LST` discrimina todos os IdentificadorFunc's disponíveis, informando inclusive o tipo dos parâmetros e do valor retornado.

IdentificadorAtrib — é um nome padronizado (constante simbólica) utilizado para denotar o índice correspondente aos atributos *Aliás* e *Texto* de objetos. Estes nomes são padronizados em todas as instalações Talisman. O arquivo `LISTAREL.LST` discrimina os IdentificadorAtrib's disponíveis.

IdentificadorRel — é o nome padronizado de um atributo *relação* entre objetos ou entre instâncias de objeto. Estes nomes são padronizados em todas as instalações Talisman. O arquivo `LISTAREL.LST` discrimina os IdentificadorRel's e as diferentes relações disponíveis, indicando a classe de objeto origem, a classe de objeto destino e a correspondente relação inversa.

IdentificadorDic — é o nome de um *dicionário de classe de objeto*. Estes nomes são padronizados em todas as instalações Talisman. O arquivo `LISTAREL.LST` discrimina os IdentificadorDic's disponíveis.

IdentificadorVar — é o nome de uma variável. Qualquer identificador que não esteja definido em um dos conjuntos anteriormente definido, será tratado como se fosse uma variável. No entanto, todas as variáveis utilizadas em um programa de formulário devem ser declaradas. Conseqüentemente, o uso de um **IdentificadorVar** antes da correspondente declaração, produzirá uma mensagem de erro de compilação.

7. SINTAXE DA LINGUAGEM

7.1 Lista de formulários

```
PROGRAMA    :=   Formulário_Global  Formulários
               | Formulários ;
Formulários :=   Formulários Formulário
               | ;
```

Um programa de formulários é formado por:

- opcionalmente, um formulário global, e
- zero ou mais formulários normais.

O formulário global se destina a definir as constantes ou variáveis globais e a realizar as inicializações requeridas para todos os demais formulários. O formulário normal efetua as operações correspondentes à ação ativada pelo usuário.

7.2 Formulário global

```
Formulário_global  := InicGlobal
                         Declarações_globais
                         Comandos_executáveis
                      FimFrm ;
```

O formulário global, caso exista, sempre será ativado antes de iniciar a execução do formulário origem correspondente à ação de usuário escolhida. O formulário global tem por objetivo definir e iniciar constantes e variáveis globais conhecidas por todo o programa.

7.3 Formulário normal

```
Formulário       := Classe_form  Cabeçalho_form
                       Declarações_locais
                       Comandos_executaveis
                    FimFrm ;
Classe_form      :=   InicFrm
                    | InicExterno;
Cabeçalho_form   := Nome_formulário Parâmetros;
Nome_formulário  := CteSeqüência ;
Parâmetros       :=   "(" Parms_formais ")"
                    | ;
Parms_formais    :=   Parms_formais "," Parâmetro_formal
                    | Parâmetro_formal ;
Parâmetro_formal := Declaração_variável ;
```

Formulários são ativados ou pela ação do usuário escolhida, ou por comandos de ativação (chamada) de formulários. Formulários podem ser chamados recursivamente. O corpo de um formulário não pode conter definições de outros formulários. Ou seja, não se pode aninhar definições de formulários.

O nome de um formulário é uma seqüência de caracteres entre aspas duplas. Caracteres maiúsculos, minúsculos, acentuados ou não, são considerados iguais. Um ou mais brancos são considerados iguais a um branco. Para maior facilidade de entendimento do programa de formulários, deve-se escolher nomes de formulários que reflitam precisamente o objetivo deste formulário.

Os nomes de todos os formulários definidos em um programa são mantidos em um dicionário relativo a este programa. Todos os formulários definidos no programa podem ser associados como formulários *default* às ações de usuário. No entanto, somente os *formulários externos* (classe de formulário **InicExterno**) aparecem na lista de seleção de formulários ao executar ações com formulários selecionados, ver Editor de Dicionários.

Formulários admitem zero ou mais parâmetros formais. No caso de zero parâmetros, a lista de parâmetros formais estará ausente. Os formulários ativáveis diretamente por ações do usuário não devem possuir parâmetros formais. Em particular formulários externos não devem possuir parâmetros formais. Se for necessária a passagem

de informação entre formulários ativados por diferentes ações, utilize variáveis de domínios, descritas mais adiante.

Valores tipo **Lógico**, **Inteiro**, **Seqüência**, **Dicionário** e **Objeto** são passados por valor. Ou seja, é passada um cópia do valor da variável. Assim quaisquer atribuições realizadas a parâmetros destes tipos não serão percebidas pelo formulário que passou estes valores como parâmetros. Já valores tipo **Atributo**, **ListaTexto** e **ListaObjetos** são passados por referência. Neste caso as alterações das variáveis são percebidas pelo formulário que passou estes parâmetros.

Exemplo 1:

```
InicFrm "Exibir objeto ligado" ( Objeto Ligação ,
                                 Objeto Origem )
```

define o formulário formulário *"Exibir objeto ligado"* recebendo como parâmetros o objeto (instância de objeto) *Ligação* e o objeto (instância de objeto) *Origem*.

Exemplo 2:

```
InicExterno "Linearizar especificação"
```

define o formulário “Linearizar especificação”, incluindo-o na lista de formulários selecionáveis.

7.4 Declarações de dados

```
Declarações_globais :=   Lista_globais  FimDecl
                       | ;
Lista_globais       :=   Lista_globais Variável_global
                       | Variável_global ;
Variável_global     :=   Domínio Declaração_variável“;“
                       | Declaração “;“ ;
Domínio             :=   BCO
                       | BAX
                       | BSW
                       | USO ;
Declarações_Locais  :=   Lista_locais FimDecl
                       | ;
Lista_locais        :=   Lista_locais Declaração“;“
                       | Declaração “;“ ;
Declaração          :=   Declaração_Constante
                       | Declaração_Variável ;
```

Todas as variáveis e constantes de um programa devem ser declaradas. As variáveis podem ser declaradas no formulário global. Neste caso elas serão visíveis em todos os formulários do programa. As variáveis podem ser declaradas, ainda, localmente a formulários. Neste caso elas serão visíveis exclusivamente no interior do formulário em que foram declaradas.

Formulários admitem zero ou mais variáveis locais. Cada vez que um formulário é ativado, cria-se uma instância de ativação deste formulário. Chamadas recursivas a formulários criam novas instâncias de execução. As variáveis locais são alocadas ao iniciar a execução de uma instância de formulário. Serão desalocadas ao terminar a execução desta instância.

7.4.1 Domínio de variáveis

Um domínio define o espaço de dados (banco de dados) no qual a variável estará alocada. As variáveis *globais* podem estar contidas em diferentes domínios. Os domínios são:

`vazio` no programa corrente. As variáveis deste domínio serão desalocadas ao terminar a execução do programa.

`USO` na base de trabalho (arquivo `TALISMAN.$$1`). As variáveis deste domínio serão desalocadas ao encerrar a sessão de uso de Talisman.

`BCO` na base de conhecimento. As variáveis deste domínio serão desalocadas ao instalar uma nova versão da base de conhecimento.

BAX na base de auxílio. As variáveis deste domínio serão desalocadas ao instalar uma nova versão da base de auxílio. Caso a base de auxílio não esteja presente, será emitida uma mensagem de erro ao ativar qualquer ação de usuário que requeira o programa de formulário.

BSW na base de software corrente. Estas variáveis nunca são desalocadas.

Em adição aos domínios globais, temos ainda os domínios *locais* a formulários:

local na instância de ativação de um formulário normal. Como formulários podem ser ativados recursivamente, existirão tantos domínios locais quantas forem as instâncias de ativação do formulário.

parâmetro na lista de parâmetros correspondente a uma instância de ativação de um formulário ou de uma função.

Através do uso de variáveis de domínio, podem ser estabelecidas comunicações entre diferentes ações de usuário, mesmo que isto ocorra em diferentes sessões de uso de Talisman.

O uso de variáveis de domínio pode ser útil, por exemplo, para controlar a progressão do trabalho. No início da especificação, os formulários de especificação e validação devem ser simples, de modo que não interfiram no entendimento do sistema sendo especificado. A medida que a especificação e o desenvolvimento vai progredindo, os formulários devem se tornar mais completos e mais formais, de modo que a especificação venha a ser suficientemente rigorosa. Através de uma variável contida no domínio da base de software correspondente (por exemplo: *NívelProgresso*), pode-se controlar esta evolução. A cada vez que uma etapa de desenvolvimento tiver sido completada, atribui-se à variável um valor de nível correspondente à etapa completada. Nos formulários seleciona-se a composição dos campos em função do valor da variável. Desta forma assegura-se o cumprimento fiel de um método de trabalho usando um único programa de formulário.

7.4.2 Inicialização automática de variáveis

Ao iniciar a execução do formulário global, será examinado se todas as variáveis de domínio (variáveis que pertencem aos domínios BCO, BAX, USO e BSW) já estão alocadas. Caso já estejam, o nome da variável é associado ao valor já definido. Caso ainda não estejam alocadas, serão alocadas, inicializadas e o valor será associado ao nome da variável.

Ao iniciar valores de variáveis de domínio, serão definidos os seguintes valores:

Tipo	Valor inicial
Lógico	Falso
Inteiro	Zero
Seqüência	seqüência de tamanho zero
Objeto	referência a objeto nulo
Dicionário	identificação de dicionário nulo
Lista	lista de objetos vazia
ListaTexto	lista de texto vazia

Variáveis globais internas ao programa e variáveis locais declaradas no formulário sendo executado, serão inicializadas automaticamente somente se forem do tipo **Lista** ou **ListaTexto**. O valor inicial, nestes casos é lista vazia. As demais variáveis não serão iniciadas automaticamente. O uso de uma variável não inicializada provoca um erro de execução e o término da execução programa de formulários. Recomenda-se que as variáveis globais sejam sempre inicializadas pelo formulário global.

7.4.3 Escopo de variáveis

O escopo determina a visibilidade dinâmica das variáveis. O escopo de variáveis pode ser:

Global externo é o conjunto de todas as variáveis declaradas no formulário global possuindo um domínio explícito. Estas declarações são visíveis a todos os formulários contidos no programa. Estas variáveis são visíveis, ainda, em outros programas nos quais figurem no formulário global. Variáveis de domínios que não figuram no formulário global não são vi-

síveis no correspondente programa. A associação de variáveis de domínios a um programa em execução é realizada de forma simbólica, usando o nome da variável. Cabe salientar que este nome não pode ser redefinido. O valor das variáveis é preservado após o término da execução do programa de formulários. Variáveis contidas nos domínios **BSW**, **BCO** e **BAX** mantém o seu valor, inclusive, ao terminar a sessão de uso de Talisman.

Global interno — é o conjunto de todas as variáveis declaradas no formulário global com domínio vazio. Estas declarações são visíveis a todos os formulários contidos no programa. Os valores são perdidos ao terminar a execução do programa de formulário.

Local — é o conjunto de todas as variáveis declaradas no corpo de um dado formulário normal ou na lista de parâmetros deste formulário. Estas declarações são visíveis somente no formulário ao qual são locais. Não é permitido o renomeamento de variáveis globais, ou seja, variáveis locais não podem ter o mesmo nome que variáveis globais. O valor de variáveis locais é perdido quando a correspondente instância de execução do formulário terminar.

7.5 Declaração de constantes de compilação

```
Declaração_constante  := Constante  IdentificadorVar
                            "="Valor_constante ;
Valor_constante       :=   CteNúmero
                         | "-" CteNúmero
                         | CteSeqüência ;
```

Constantes permitem associar um nome mnemônico com um determinado valor. Isto facilita a leitura, bem com a manutenção de programas. Deve ser evitado o uso de números e seqüências "mágicas" no corpo do programa. Para tal recomenda-se a definição de todos estes números e seqüências sob a forma de constantes no formulário global.

Constantes existem somente durante a fase de compilação. O tipo de uma constante é definido a partir do seu valor. Constantes admitem somente os tipos **Inteiro** e **Seqüência**.

Em adição às constantes definidas no programa, diversas constantes são definidas pelo sistema. Estas constantes identificam os diversos atributos padronizados. As constantes do sistema dependem da configuração do ambiente. Cada linguagem de representação define as constantes que usa. O arquivo `LISTAREL.LST` fornece uma lista de todas as constantes definidas pelo sistema. Note que esta lista pode variar a medida que novas representações forem sendo desenvolvidas.

Exemplos:

```
Constante DescriçãoObjeto = 0 ;
Constante NomeProposto    = "Corpo de " ;
```

definem a constante numérica *DescriçãoObjeto* com valor 0, e a constante seqüência *NomeProposto* com valor "Corpo de ".

7.6 Declaração do tipo de variáveis

```
Declaração_variável := Tipo  IdentificadorVar ;
Tipo                :=   Lógico
                       | Inteiro
                       | Seqüência
                       | ListaTexto
                       | Dicionário
                       | Objeto
                       | Fila
                       | Pilha
                       | ConjuntoFila
                       | ConjuntoPilha ;
```

Nomes de variáveis e de constantes são seqüências de um ou mais caracteres letra ou dígito ou "_", iniciando com letra. Letras maiúsculas, minúsculas, acentuadas ou não são consideradas sendo iguais.

Valores admitidos pelos diferentes tipos:

Lógico Verdadeiro, Falso.

Inteiro valores entre -32767 e +32767.

Seqüência seqüências de 0 a 250 caracteres.

ListaTexto listas de 0 ou mais linhas, cada qual sendo uma seqüência de 0 a 250 caracteres.

Dicionário identifica uma classe de objeto conhecida. Cada linguagem de representação define um conjunto de classes de objeto. Um valor tipo dicionário pode assumir a identificação de qualquer uma destas classes.

Objeto referencia um objeto ou instância de objeto. Valores do tipo objeto armazenam a informação completa de acesso a um objeto ou a uma instância de objeto. Entende-se por objeto uma coletânea de dados (atributos) pertencente a um elemento de determinada classe. Entende-se por instância de objeto uma coletânea de dados pertencentes à ocorrência de uso de um objeto em um determinado diagrama. Para acessar um objeto basta conhecer-se a sua identificação. Para acessar uma instância de objeto, precisa-se saber a folha na qual está definida a instância e a identificação, nesta folha, da instância. O valor tipo Objeto contém a referência completa do objeto ou da instância de objeto. Dependendo da linguagem de representação, determinadas classes de objetos podem ser instanciadas, ou seja, podem figurar em um diagrama. Por exemplo, na linguagem Fluxo de Dados, objetos das classes *Processos*, *Depósitos de Dados* e *Entidade Externa* podem ser instanciados. Já objetos da classe *Estrutura de Dados* não podem ser instanciados. Algumas linguagens de representação admitem instâncias de objeto para as quais não existem dicionários de classe. Conseqüentemente, estes objetos podem ocorrer somente sob a forma de instâncias de objeto. Por exemplo, na linguagem *Fluxo de Dados*, objetos da classe *Ponto de Confluência* não são dicionarizados. Na linguagem *Entidades e Relacionamentos*, objetos da classe *Cardinalidade* não são dicionarizados.

Pilha listas de 0 ou mais objetos e/ou instâncias de objeto pertencentes a classes quaisquer. A inserção se dá no início da lista, sendo permitidas mais de uma referência a um mesmo objeto. A retirada é sempre no início da lista.

Fila listas de 0 ou mais objetos e/ou instâncias de objeto pertencentes a classes quaisquer. A inserção se dá no final da lista, sendo permitidas mais de uma referência a um mesmo objeto. A retirada é sempre no início da lista.

ConjuntoPilha listas de 0 ou mais objetos e/ou instâncias de objeto pertencentes a classes quaisquer. A inserção se dá no início da lista. Caso um objeto já figure na lista, ele não será inserido. A retirada é sempre no início da lista.

ConjuntoFila listas de 0 ou mais objetos e/ou instâncias de objeto pertencentes a classes quaisquer. A inserção se dá no início da lista. Caso um objeto já figure na lista, ele não será inserido. A retirada é sempre no início da lista.

ListaObjeto é qualquer valor do tipo Pilha, Fila, ConjuntoFila ou ConjuntoPilha. Na realidade este tipo não existe explicitamente. Ele será utilizado somente em lugares onde é irrelevante o tipo específico, como por exemplo ao definir parâmetros.

7.7 Lista de comandos

```
Comandos_executáveis  :=   Comandos_executáveis Executa ";"
                         | ;
Executa               :=   Edição_campo
                         | Comando ;
```

Os comandos executáveis formam uma lista de comandos, cada qual terminado por ";". Cada comando executável pode ser ou uma edição de campo ou um comando normal.

7.8 Edição de campo

```
Edição_campo  :=   Campo_não_editável
                 | Campo_editável
                 | Formatação_campo ;
```

Os comandos de edição de campo dirigem texto para a lista de edição. A lista de edição pode ser a lista de linhas do formulário editável, um fragmento de texto, a impressora ou um arquivo. O destino de geração da lista é definido pelos comandos de definição da geração e/ou pelo modo de ativar o programa de formulário. Ao ativar um programa de formulário pode-se:

- indicar que o texto gerado deve ser editado. Neste caso o texto é dirigido para uma lista de texto editável.
- indicar que o texto gerado deve ser atribuído ao fragmento de texto TxtLaudo do objeto origem. Isto ocorre ao executar programas de validação.
- indicar que o texto deve ser impresso. Neste caso o texto será dirigido para a impressora ou para o "spool" de impressão.
- indicar que o texto deve ser arquivado. Neste caso o texto será dirigido para o arquivo de texto ASCII identificado ao iniciar a geração (linearização).

Durante a execução a geração de texto pode ser redirecionada. Isto é efetuado pelos comandos de geração de texto, descritos mais adiante.

7.9 Campos não editáveis

```
Campo_não_editável  :=   Título Expressão
                       | TextoLaudo
                       | Protegido Campo_editável
                       | MacroTecla ;
```

Campos não editáveis são exibidos, porém o editor de formulários não poderá alterá-los. Todos os comandos de formatação valem para campos protegidos, inclusive tamanho de campo e margem direita. O valor de campo protegido será rolado se for maior do que o visor virtual na tela.

Título exibe valores constantes, conteúdo de variáveis quaisquer, ou valores calculados em expressões, desde que possam ser convertidas para seqüência de caracteres.

TextoLaudo é um fragmento de texto especial que se destina a registrar o laudo de validação do objeto. Este laudo é gerado ao validar o objeto usando o programa de validação. Nas formas:

- **TextoLaudo** é um campo protegido.
- **Texto** TxtLaudo é um campo editável que acessa o fragmento de texto laudo.
- **Texto** 255 é um campo editável que acessa o fragmento de texto laudo.

MacroTecla exibe a tecla associada à macro corrente. O mecanismo de edição de formulários pode ser utilizado também para editar o corpo de macros e lições. Neste caso, ao invés de se operar sobre um objeto corrente, opera-se sobre uma macro corrente. A edição de macros e lições é realizada pelo programa de formulários **Serviços**.

Protegido Qualquer campo editável poderá ser transformado em protegido, prefixando o seu uso com a declaração **Protegido**.

Observação para exibir um título cujo valor esteja contido em uma variável ou na base de software, será gerada uma cópia da seqüência e, depois, esta cópia será transferida para a lista de edição. Esta forma é ineficiente. Uma forma mais eficiente é utilizar a proteção de campos para tais valores.

7.10 Campos editáveis

```
Campo_editável  :=   Campo_BSW
                   | Variável IdentificadorVar
                   | Campo_macro ;
Campo_BSW       :=   Nome
                   | Aliás Expressão
                   | Texto Expressão
                   | Relação Expressão ;
Campo_Macro     :=   MacroNome
                   | MacroTexto ;
```

Um campo editável pode ser o valor de um *atributo de objeto* contido em uma base de software, pode ser o valor de uma *variável* ou, finalmente, pode ser o valor de um *atributo de uma macro* de teclado.

Os atributos de um objeto são agregados em *categorias de atributos*. Cada categoria possui um *nome da categoria* e um *discriminador de atributo*. O discriminador de atributo discrimina um entre os vários atributos de uma dada categoria. Estão definidas as seguintes categorias de atributos:

Nome corresponde ao nome do objeto. O atributo nome é único, conseqüentemente, não possui discriminador de atributo.

Aliás corresponde a uma seqüência de 0 a 250 caracteres. Podem existir até 255 diferentes aliases para cada objeto. O *aliás* específico a ser manipulado é discriminado ou por um IdentificadorAtrib definido na base de conhecimento, ou por um número inteiro entre 0 e 255, resultado da avaliação de uma expressão. Os IdentificadorAtrib's são definidos para cada linguagem de representação suportada. Essencialmente eles correspondem a uma constante inteira.

Texto *fragmento de texto*, corresponde a uma lista de 0 a 32760 linhas, cada uma contendo seqüências de 0 a 250 caracteres. Podem existir até 255 diferentes fragmentos de texto para cada objeto. O fragmento de texto específico a ser manipulado é discriminado ou por um IdentificadorAtrib definido na base de conhecimento, ou por um número inteiro entre 0 e 255, resultado da avaliação de uma expressão. Os IdentificadorAtrib's são definidos para cada linguagem de representação suportada. Essencialmente eles correspondem a uma constante inteira.

Relação corresponde a uma lista de objetos referenciados pelo objeto corrente (referência cruzada). Cada relação determina a classe do objeto destino e a relação inversa contida nos objetos destino. A relação específica a ser manipulada é discriminada por um IdentificadorRel definido na base de conhecimento, ou por um número inteiro entre 0 e 255, resultado da avaliação de uma expressão. Cabe frisar que, embora números inteiros sejam permitidos, relações devem ser sempre identificadas pelo seu nome simbólico IdentificadorRel. A única exceção a esta regra ocorre quando se deseja a passar como parâmetro o número da relação ao invés de uma referência ao atributo relação correspondente.

Além dos atributos de objetos, podem ser editados, ainda, os atributos de macros de teclado. Estes atributos são:

MacroNome é uma seqüência de até 250 caracteres que corresponde ao nome de uma macro.

MacroTexto é uma lista de texto que determina a seqüência de teclas que forma uma macro, incluindo as janelas explanatórias porventura definidas, ver capítulo *Macros e Lições*.

O formato de exibição e o modo de editar uma variável depende do seu tipo. Todas as variáveis, inclusive variáveis lógicas e numéricas, são exibidas alinhadas à esquerda. Ao editar um campo o “feedback” de estado, exibido na linha superior do vídeo (ver capítulo **Convenções**), identifica o tipo de campo sendo editado. Isto permite saber, a cada momento, que operações de edição são permitidas.

Para atributos não padronizados na base de conhecimento, recomenda-se o uso de constantes definidas no formulário global. Assegura-se assim a padronização e facilita-se eventuais necessidades de redefinição.

7.11 Formatação de campo

```
Formatação_campo  :=   MrgEsq Expressão
                     | MrgDir  Expressão
                     | AvMrgEsq Expressão
                     | NãoAvLin
                     | AvEsq Expressão
                     | TamCampo Expressão
                     | TextoEsq Expressão
                     | Endenta Expressão  ;
```

Os comandos de formatação descrevem como devem ser exibidos os campos editáveis ou não. Os comandos de formatação podem definir um contexto que vale:

- somente para o próximo campo a ser exibido, ou
- para todos os campos a serem exibidos a seguir.

O efeito de todos os comandos de formatação cessa:

- ou quando for executado um outro comando de formatação operando sobre o mesmo formato.
- ou quando o formulário no qual foram os comandos de formatação foram definidos retorne para quem o chamou. Cabe salientar que isto vale, inclusive, para chamadas recursivas.

O efeito dos comandos de formatação perdura nos formulários chamados pelo formulário que contém o comando de formatação.

Os comandos de formatação a seguir valem para todos os campos a serem exibidos após o comando:

MrgEsq e **MrgDir** definem as margens dos campos após o comando. A margem será tornada igual ao valor da *expressão*. A primeira coluna tem índice igual a 1.

AvMrgEsq redefine a *margem esquerda*, adicionando o valor da *expressão* ao valor corrente da margem. Caso o valor da expressão seja positivo, a margem esquerda é deslocada para a direita. Caso seja negativo, a margem será deslocada para a esquerda.

AvEsq determina que campos sucessivos contidos em uma mesma linha estejam separados por *expressão* caracteres em branco.

Por exemplo, supondo que o nome do objeto corrente seja "Nome do objeto", os comandos a seguir:

```
AvEsq 2 ;
Título "/*" ;
NãoAvLin ;
Nome ;
NãoAvLin ;
Título "*/" ;
```

cria uma única linha de texto com o formato:

```
/*..Nome do objeto..*/
```

onde o caractere “.” denota um espaço em branco.

Endenta define um valor a ser adicionado à margem esquerda em todas as linhas que não iniciem com um campo título. Caso o valor seja negativo, a margem será deslocada para a esquerda. Por exemplo, **Endenta 3** fará com que todos as linhas que iniciem com um campo que não seja um título, estejam 3 caracteres para a direita das linhas que iniciem com título. Cria-se, assim, um padrão de endentação de textos.

TextoEsq define uma seqüência de caracteres a ser colocada no início de cada nova linha, independentemente desta ser um título, linha de campo texto, ou outro campo qualquer. A seqüência será inserida sempre na margem esquerda 1. O controle de margem esquerda será realizado após esta seqüência de caracteres. O tamanho da seqüência é limitado a 20 caracteres, podendo conter qualquer valor. Para cancelar a inclusão de seqüências à esquerda, basta definir uma seqüência de zero caracteres.

Por exemplo, assumindo que o *Texto 1* do objeto corrente contenha as linhas a seguir:

```
Esta é a primeira de três linhas
Esta é a segunda de três linhas
Esta é a terceira de três linhas
```

a seqüência de comandos:

```
TextoEsq "C ----- " ;
Título "Comentários em FORTRAN" ;
AvMrgEsq 3 ;
Texto 1 ;
TextoEsq "" ;
Título "Sem texto à esquerda" ;
```

gerar as seguintes linhas:

```
C ----- Comentários em FORTRAN
C -----    Esta é a primeira de três linhas
C -----    Esta é a segunda de três linhas
C -----    Esta é a terceira de três linhas
   Sem texto à esquerda
```

Durante a edição, somente a primeira linha de um campo de texto ou de relação será precedida pela seqüência a adicionar. No entanto, ao imprimir, ao gerar arquivos e ao redirecionar o texto para uma variável ou atributo texto, todas as linhas serão precedidas pela seqüência definida, mesmo que estas sejam linhas de um campo texto ou de um campo relação.

Os comandos de formatação a seguir valem somente para o primeiro campo a ser gerado após o comando:

NãoAvLin determina que o próximo campo estará na mesma linha que o último caractere do campo anterior.

TamCampo define o tamanho do próximo campo. O campo a seguir ao próximo campo, caso esteja na mesma linha, será colocado após o limite definido pelo tamanho do campo, mesmo que o valor do campo ocupe um espaço menor do que o tamanho dado.

Caso a margem direita de um campo esteja além da margem direita física do vídeo, ao editar os caracteres que estejam fora dos limites da janela de edição, toda a janela será rolada. Caso a margem direita de um campo seja menor do que o tamanho de uma linha deste campo, o texto desta linha será rolado em uma janela virtual respeitando o tamanho do campo ou a margem direita definida. Ao gerar um fragmento de texto, ao imprimir, ou ao gravar campos os limites de tamanho e margem direita somente serão respeitados se o valor do campo não conflitar com estes limites, mesmo que isto provoque o desalinhamento do formato da listagem.

7.12 Comandos executáveis

```
Comando  :=   Define_geração
            | Chama_formulário
            | Chama_procedimento
            | Repetição
            | SaiRepet
            | Seleção
            | Atribuição ;
```

7.13 Geração de texto formatado

```
Define_geração  :=   GeraTexto Elimina Expressão
                   | GeraVar Elimina IdentificadorVar
                   | GeraLaudo Elimina ;
Elimina         :=   Novo
                   | ;
```

Dependendo da ação do usuário, os campos a editar serão dirigidos para a lista de edição, a impressora, um arquivo de linearização, ou para uma lista de texto. Assim, a ação de usuário:

Editar dirige o texto dos campos para a lista de edição.

Validar dirige o texto dos campos para o fragmento de *texto laudo* associado ao objeto origem.

Imprimir dirige o texto dos campos para uma impressora ou para um arquivo "spool".

Gravar dirige o texto dos campos para um arquivo seqüencial.

Durante a execução de um formulário, a geração de texto pode ser redirecionada para um fragmento de texto do objeto corrente ou para uma variável tipo **ListaTexto**. Ao gerar o texto são respeitados todos os comandos de formatação, viabilizando, assim a geração de uma lista formatada correspondente a uma extração da base de software tão complexa quanto se queira. O efeito do comando perdura até que termine a execução do formulário que contém o comando.

Os comandos a seguir redirecionam a geração de texto editável:

GeraLaudo direciona o texto para o fragmento de texto laudo (índice 255) do objeto corrente.

GeraTexto direciona o texto para o fragmento de texto do objeto corrente, cujo índice é o resultado da *Expressão*.

GeraVar direciona o texto para a variável tipo **ListaTexto** de nome IdentificadorVar.

Se a palavra **Novo** for incluída no comando, o texto existente será eliminado antes de iniciar a geração. Caso contrário, o texto gerado será adicionado ao final do texto existente.

Redirecionamentos sucessivos para uma mesma variável ou para um mesmo atributo de objeto, produzirão a concatenação dos textos gerados. Exceto, é claro se o redirecionamento contiver a palavra **Novo**.

Cada comando de redirecionamento ativa um novo descritor de redirecionamento. Assim, comandos de redirecionamento sucessivos contidos em um mesmo formulário criarão vários descritores. Isto pode causar efeitos imprevistos quando o destino da redireção é um mesmo atributo de objeto ou uma mesma variável. Para assegurar um comportamento determinístico, defina cada redirecionamento em um formulário próprio.

Os comandos de redirecionamento da geração permitem que se crie validadores e transformadores que navegam uma estrutura contida na base de software, gerando diversos laudos e/ou textos formatados distintos, atribuindo cada um deles a um atributo texto ou variável específica.

Por exemplo, o trecho de código a seguir:

```
InicFrm "Validar módulo"
     GeraLaudo Novo ;
          ...
FimFrm
     ...
InicFrm "Validar todos módulos"
     GeraVar Novo TextoErro ;
     ParaTodos Dicionário DicModulos Faz
          Frm "Validar módulo" ;
          Se Existe( [ Texto TxtLaudo ] )
          Então
               Título "Módulo contém laudo " ;
               NãoAvLin ;
               Nome ;
          Senão
          Fim ;
     Fim ;
FimFrm
```

redireciona o texto gerado para a variável *TextoErro*. O valor da variável é esvaziado. Depois, percorre todo dicionário de módulos. Para cada elemento deste dicionário, ativa o formulário *Validar módulo*. Todos os campos gerados durante a execução deste formulário serão direcionados para o texto laudo do respectivo módulo. Após validar um dado módulo, é verificado se este passou a ter o fragmento de texto TextoLaudo. Caso tenha, será acrescentada à variável *TextoErro* a mensagem *Módulo contém laudo* e o nome deste módulo.

7.14 Chamadas de formulário

```
Chama_formulário  :=   Frm Nome_Formulário Lista_parâmetros
                     | FrmElem Expressão Nome_Formulário
                          Lista_parâmetros
                     | Executa Expressão Lista_parâmetros ;
```

Durante a execução de um formulário podem ser chamados outros formulários. Os formulários chamados podem ser normais (**InicFrm**) ou externos (**InicExterno**).

Ao chamar um formulário existirá sempre um objeto corrente. Quando do início da execução, este é o objeto origem, relativo ao qual foi ativada a ação do usuário. Ao chamar um formulário de dentro de outro, o objeto corrente é definido em função da instrução de chamada de formulário utilizada.

Frm chama o formulário definido pelo nome, aplicando-o ao objeto corrente.

FrmElem chama o formulário definido pelo nome, aplicando-o ao objeto definido por *Expressão*.

Executa chama o formulário identificado pela seqüência de caracteres resultante da avaliação de *Expressão*, aplicando-o ao objeto corrente. O resultado da expressão deve ser o nome simbólico de um dos formulários definidos no programa corrente, normais ou externos. Caso não exista nome igual, será emitida uma mensagem de erro e a execução do programa de formulários será terminada.

Exemplos:

```
Frm "Exibir fluxos de entrada" ;
```

ativa o formulário *Exibir fluxos de entrada* aplicando-o ao objeto corrente.

```
FrmElem Obj "Exibir fluxos de entrada" ;
```

ativa o formulário *Exibir fluxos de entrada* aplicando ao objeto referenciado pela variável *Obj*.

```
Executa [ Alias 5 ] ;
```

ativa o formulário cujo nome está no atributo *Aliás 5* do objeto corrente.

7.15 Lista de parâmetros atuais

```
Lista_parâmetros   :=   Parâmetros_atuais
                     | ;
Parâmetros_atuais  :=   "(" Lista_atuais ")" ;
Lista_atuais       :=   Lista_atuais "," Expressão
                     | Expressão ;
```

Ao chamar um formulário pode-se passar zero ou mais parâmetros atuais. Cada parâmetro atual é o resultado da avaliação de uma expressão. As expressões de parâmetros atuais são avaliadas da esquerda para a direita.

Os parâmetros atuais devem corresponder em número e tipo aos parâmetros formais definidos no cabeçalho do formulário chamado. A associação entre parâmetros formais e atuais é posicional.

Em geral a passagem de parâmetros se dá por valor. Ou seja, é gerada uma cópia do valor do parâmetro atual, sendo esta cópia processada pelo formulário chamado. Conseqüentemente, todas as edições realizadas nestes parâmetros não serão percebidas pelo formulário que chamou. A passagem por valor não vale para parâmetros tipo Referência a atributo, **ListaObjeto** e **ListaTexto**. Variáveis deste tipo são passados por referência. Conseqüentemente, alterações realizadas nestas listas serão percebidas pelo formulário que chamou.

Caso o formulário chamado já tenha sido declarado e/ou usado anteriormente, os parâmetros atuais serão convertidos para os tipos definidos nos respectivos parâmetros formais. Serão emitidas mensagens de erro de compilação caso a conversão de tipos seja impossível, ou caso o número de parâmetros formais e atuais não seja igual.

Caso um formulário seja chamado antes de ser declarado, os tipos e a quantidade de seus parâmetros formais será extraída da lista de parâmetros atuais contida na chamada. Ao ser declarado mais adiante, o número e tipo dos parâmetros formais deve ser igual ao número e o tipo extraído do comando de uso deste formulário. Para assegurar que determinado tipo seja especificamente utilizado, preceda a expressão por um comando de conversão de tipos. Para evitar problemas decorrentes do uso de um formulário antes de sua declaração, assegure que a declaração sempre ocorra antes. Isto somente não será possível caso exista uma seqüência recursiva indireta de chamadas.

Exemplo 1:

```
InicFrm "Imprimir texto"( Seqüência TitTexto ,
                          Inteiro IdAtributo )
```

```
   Se Existe( [ Texto IdAtributo ] )
   Então
      Titulo TitTexto ;
      AvMrgEsq 3 ;
      Texto IdAtributo ;
   Senão
   Fim ;
FimFrm
```

ativado com a chamada:

```
Frm "Imprimir texto"( "Descrição" , 0 ) ;
```

gera nada, caso o texto *Texto 0* não esteja definido para o objeto corrente na base de software. Gera o texto precedido do título, caso o texto esteja definido na base de software.

Exemplo 2:

```
     ...
     Frm "Mostrar"( [ Nome ] ) ;
     ...
InicFrm "Mostrar" ( Seqüência Seq )
     ...
FimFrm
```

é chamado o formulário "Mostrar". Como este ainda não havia sido definido, será assumido que ele possui um parâmetro formal do tipo *referência a atributo nome*. Como a definição do formulário não está em concordância com este tipo assumido, será gerada uma mensagem de erro de compilação ao compilar a linha InicFrm "Mostrar ...

Exemplo 3:

```
     ...
     Frm "Mostrar"( Converte Seqüência [ Nome ] ) ;
     ...
InicFrm "Mostrar" ( Seqüência Seq )
     ...
FimFrm
```

ilustra o uso de conversão explícita para assegurar o tipo desejado dos parâmetros. Neste exemplo não existe erro de compilação, uma vez que a referência ao atributo Nome é explicitamente convertida para **Seqüência**.

7.16 Comandos de repetição e navegação

```
Repetição  :=   Enquanto
             | Para_Todos
             | Com_Objeto ;
Enquanto   := Enquanto Expressão_lógica Faz
                 Comandos_executáveis
              Fim ;
Para_Todos := ParaTodos Conjunto Faz
                 Comandos_executáveis
              Fim ;
Com_Objeto := ComObjeto Expressão Faz
                 Comandos_executáveis
              Fim ;
Conjunto   :=   IdentificadorVar
              | IdentificadorRel
              | Dicionário  IdentificadorDic ;
Sai_repet  := SaiRepetição ;
```

Os comandos de repetição e navegação executam o corpo zero ou mais vezes, dependendo de cada classe de comando. Em adição, alguns comandos podem provocar a navegação para outro objeto.

Enquanto repete o seu corpo enquanto a avaliação da expressão lógica resultar em **Verdadeiro**. O comando **Enquanto** não provoca navegação.

Exemplo 1:

```
Endenta 3 ;
I = 0 ;
Enquanto I < 30 Faz
     Se Existe( [ Texto I ] )
     Então
          Título "Texto " ;
          NãoAvLin ;
          Título I ;
          Texto I ;
     Senão
     Fim;
     I = I + 1 ;
Fim ;
```

exibe todos os fragmentos de texto de índices 0 a 29 inclusive desde que estejam definidos no objeto corrente. Caso o fragmento de texto exista, será impresso um título identificando o texto, seguido do texto.

Exemplo 2:

Assumindo que a variável *ListaPend* esteja definida e que contenha o objeto origem da exploração, o código:

```
Enquanto NÃO ListaVazia( ListaPend ) Faz
   ComObjeto RetiraObj( ListaPend ) Faz
      ...
      InsereObj( ListaPend , ObjetoVisit ) ;
      ...
   Fim ;
Fim ;
```

caminha sobre a estrutura definida pela seqüência de objetos visitados (variável *ObjetoVisit*) em largura, caso a lista seja uma fila, e em profundidade, caso a lista seja uma pilha.

ParaTodos repete o seu corpo para todos os objetos de um dado conjunto, navegando para estes objetos. Ao terminar a execução do comando, o objeto corrente será o mesmo que antes de iniciar a execução. O conjunto percorrido pode ser:

Dicionário IdentificadorDic neste caso serão percorridos todos os elementos do dicionário definido por IdentificadorDic. O nome do dicionário deve ser um dos nomes de classe de objeto definidos em alguma das linguagens de representação.

IdentificadorRel neste caso serão percorridos todos os elementos que figurem na relação IdentificadorRel do objeto corrente. O nome da relação deve ser um dos nomes de relação permitidos para a classe de objetos corrente. Caso, durante a execução do programa, for tentado utilizar uma relação que não esteja definida para o objeto corrente, a execução será interrompida com uma mensagem de erro.

IdentificadorVar onde a variável é do tipo **Inteiro**. É tratado da mesma forma como o caso IdentificadorRel. O valor da variável será tratado como identificador da relação a ser processada. Os identificadores numéricos das relações são listados nos anexos.

IdentificadorVar onde a variável é do tipo **ListaObjetos**. Caso seja uma **ListaObjetos** serão percorridos todos os elementos da respectiva lista de objetos. A lista permanece inalterada. Durante a execução da repetição a lista não deve ser modificada. Caso a lista seja modificada, os resultados serão imprevisíveis. Veja o exemplo 2 do comando **Enquanto** para um exemplo de processamento de lista variável durante a execução da repetição.

IdentificadorVar onde a variável é do tipo **Dicionário**. Percorre todos os elementos do dicionário.

Exemplo 1:

```
ParaTodos Dicionário DicModulos Faz
   Nome ;
Fim ;
```

gera a lista de todos os nomes de módulos conhecidos na base de software corrente.

Exemplo 2:

```
ParaTodos Decomp Faz
   Titulo "" ;
   Nome ;
   Texto 19 ;
Fim ;
```

gera uma seqüência de linhas formadas por uma linha em branco, uma linha contendo o nome e as linhas do fragmento de texto índice 19 de todos os objetos relacionados com o objeto corrente através da relação *Decomp*.

Exemplo 3:

```
ParaTodos InxRel Faz
   InsereObj( FilaObj , Corrente) ;
Fim ;
```

assumindo que a variável *InxRel* contenha um índice válido para uma relação do objeto corrente, serão introduzidos na lista *FilaObj*, todos os elementos relacionados com o objeto corrente através da relação *InxRel*.

Exemplo 4:

```
ParaTodos InstsLigadas Faz
   FrmElem ObjetoInst( Corrente ) "Exibir Especificação" ;
Fim ;
```

estando sobre uma instância de objeto X em um diagrama, percorre todas as instâncias de objeto Y ligadas a esta instância de objeto X. Por intermédio da função *ObjetoInst* é determinada a referência ao objeto instanciado pela instância Y e aplica formulário *Exibir especificação* a esta referência de objeto. Este exemplo ilustra o uso de uma relação gráfica. Relações gráficas são descritas com mais detalhe no capítulo **Representações gráficas**.

Exemplo 5:

```
ParaTodos FilaPend Faz
   Nome ;
Fim ;
```

assumindo que a variável *FilaPend* contenha uma fila de objetos, serão exibidos os nomes de todos estes objetos.

ComObjeto executa o corpo exatamente uma vez, navegando para o objeto resultado da avaliação de *Expressão*. A navegação utilizando **ComObjeto** é, em geral, mais fácil de entender do que a navegação utilizando a chamada de formulário **FrmElem**.

Exemplo:

Assumindo que o objeto corrente seja uma instância, o código:

```
ParaTodos LigSai Faz
   ComObjeto RotuloLig( Corrente ) Faz
      Se ExisteObj( Corrente )
      Então
         ComObjeto ObjetoInst( Corrente ) Faz
            Nome ;
         Fim ;
      Senão
         Título "Ligação sem rótulo" ;
      Fim ;
   Fim ;
Fim ;
```

exibe todos os rótulos de ligações de saída da instância de objeto corrente.

A execução de uma repetição **Enquanto** e **ParaTodos**, ou de uma navegação **ComObjeto**, pode ser interrompida a qualquer momento pelo comando **SaiRepet**. Este comando sai da repetição mais interna, continuando a execução após o correspondente delimitador **Fim**.

7.17 Comandos de seleção condicional

```
Seleção  := Se Expressão_lógica
            Então
               Comandos_executáveis
            Senão
               Comandos_executáveis
            Fim ;
```

Se a expressão lógica resultar em **Verdadeiro**, será executada a parte **Então** do comando. Caso resulte em **Falso** será executada a parte **Senão** do comando. Os elementos **Então**, **Senão** e **Fim** devem ser sempre fornecidos.

Exemplo 1:

```
Se I <0
Então
   Título "Indice de atributo negativo" ;
Senão
   Texto I ;
Fim ;
```

neste exemplo será exibido o título caso o valor de I seja menor do que 0. Será exibido o valor I em caracteres ASCII caso o valor seja maior ou igual a zero.

Exemplo 2:

```
Se Existe( [ Texto I ] )
Então
   Título "Texto " ;
   NãoAvLin ;
   Título I ;
   Texto I ;
Senão
Fim ;
```

este exemplo ilustra a obrigatoriedade da palavra **Senão**. Mesmo que não exista código para o caso **Falso**, a parte **Senão** deve ser fornecida com corpo vazio.

7.18 Comandos de atribuição

```
Atribuição  :=   IdentificadorVar "=" Expressão
               | Campo_BSW  "="Expressão ;
```

A atribuição converte o tipo resultado da expressão para o tipo do valor destino. Este tipo é determinado pelo tipo da variável ou pelo tipo do atributo. Ao atribuir a uma variável, será alterado o valor desta variável. Ao atribuir a um *Campo_BSW* será alterado o conteúdo da base de software. Ao atribuir um valor nulo a um *Campo_BSW*, o correspondente atributo será eliminado. A definição dos *Campo_BSW* é idêntica à definição usada para editar campos.

A atribuição é realizada imediatamente. Ou seja, todas as atribuições a objetos contidos na base de software são efetuados no local onde se encontra o comando de atribuição. Lembre-se que, ao editar formulários, as alterações são registradas somente ao terminar a edição do formulário.

Atributos *Nome* aceitam expressões do tipo **Seqüência** de caracteres. Ao atribuir a um nome, o nome do objeto corrente será substituído pela seqüência sendo atribuída. Não será criado um objeto novo. A seqüência deve ser não nula, e deve ser diferente dos nomes de todos outros objetos da classe do objeto corrente. Caso não o seja, será emitida uma mensagem de erro e a execução do programa de formulários será interrompida. Para criar um objeto novo, deve ser utilizada a função *CriarObjeto(Classe , Seqüência)*.

Atributos *Aliás* aceitam valores do tipo **Seqüência** de caracteres.

Atributos *Texto* aceitam valores do tipo **ListaTexto**.

Atributos *Relação* aceitam valores do tipo **ListaTexto**. Nestas listas cada linha não vazia corresponderá ao nome de um objeto. Caso o nome objeto na classe destino da relação já exista, será utilizada a referência ao ob-

jeto correspondente. Caso o nome não exista, será criado um novo objeto e será usada a referência a este novo objeto.

7.19 Expressões

7.19.1 Expressão lógica

```
Expressão         :=   Expressão  Operador_lógico Expressão_lógica
                     | Expressão_lógica ;
Operador_lógico   :=   OU
                     | E
                     | OUEX
                     | IMPLICA ;
```

Expressões operam com valores tipo *Lógico*. Todos os operadores lógicos são tratados com a mesma prioridade. A conversão automática obedece às seguintes regras de conversão:

Tipo	Valor	Resultado
Inteiro	0	**Falso**
Inteiro	não 0	**Verdadeiro**

Os operadores lógicos têm **E** e **OU** têm significado convencional. O operador lógico **OUEX** corresponde a *Ou exclusivo*. O operador lógico **IMPLICA** possui a seguinte tabela de verdade:

A **IMPLICA** B	B==**Falso**	B==**Verdadeiro**
A == **Falso**	**Verdadeiro**	**Verdadeiro**
A==**Verdadeiro**	**Falso**	**Verdadeiro**

7.19.2 Expressão de comparação

```
Expressão_lógica  :=   Expressão_lógica Comparador  Comparando
                     | Comparando ;
Comparador        :=   "<"
                     | "<="
                     | "=="
                     | "!="
                     | ">="
                     | ">" ;
```

O significado dos operadores de comparação é:

<	menor
<=	menor ou igual
==	igual
!=	não igual
>=	maior ou igual
>	maior

Expressões de comparação operam somente com valores tipo **Inteiro**, ou tipo **Seqüência**. Comparações com outros tipos devem ser realizadas através de funções. O resultado de uma expressão de comparação é um valor do tipo **Lógico**.

A comparação de seqüências não leva em conta o fato de letras serem maiúsculas, minúsculas acentuadas ou não. Em adição, a comparação das seqüências A < B resultará em **Verdadeiro** se existir um prefixo da seqüência A de caracteres iguais aos correspondentes caracteres da seqüência B e terminado por um caractere menor (segundo a tabela ASCII) do que o correspondente caractere em B. Também resultará em verdadeiro se A for um prefixo

próprio de B, ou seja, se todos os caracteres de A forem iguais aos correspondentes caracteres de B e o tamanho de A for menor do que o tamanho de B.

7.19.3 Expressão aritmética

```
Comparando        :=   Comparando  Operador_adição Termo
                     | Termo ;
Operador_adição   :=   "+"
                     | "-" ;
Termo             :=   Termo  Operador_multip  Fator
                     | Fator ;
Operador_multip   :=   "*"
                     | "/"
                     | RESTO ;
```

Expressões aritméticas operam com valores tipo **Inteiro** (-32767 a +32767), inclusive os operadores de multiplicação, divisão e resto. Caso os valores sejam de qualquer outro tipo, eles serão primeiro convertidos para **Inteiro**.

7.19.4 Operadores unários

```
Fator             :=   Operador_unário  Elemento
                     | Elemento ;
Operador_unário   :=   NÃO
                     | "-"
                     | Converte  Tipo ;
```

O operador *menos unário* se aplica a elementos inteiros. O operador *menos unário* antes de uma constante é processado em tempo de compilação gerando uma constante negativa.

O operador **NÃO** se aplica a elementos lógicos.

O operador **Converte *Tipo*** força a conversão do elemento a seguir para o tipo definido. Usualmente as conversões são realizadas de modo automático. No caso de formulários chamados antes de sua definição, o uso de conversões explícitas permite assegurar a correta definição dos tipos dos parâmetros formais.

A tabela a seguir identifica todas as conversões possíveis. Estas conversões são realizadas ou automaticamente ou pelo operador **Converte *Tipo***. As conversões que não figurem na tabela a seguir resultam em erro de compilação.

Todas as conversões automáticas são determinadas pelo elemento à esquerda. Ou seja, ao converter os valores manipulados por uma operação binária, o valor da direita será convertido para o tipo do valor à esquerda, sendo o valor da esquerda deixado inalterado.

Exemplos:

```
"001" == 1
```

resulta em Falso, uma vez que a seqüência "001" é diferente da seqüência "1".

```
[ Nome ] == "abcd"
```

resulta em erro de compilação, uma vez que a seqüência "abcd" não pode ser convertida para uma referência a um atributo Nome.

```
"abcd" == [ Nome ]
```

resulta em **Verdadeiro** se o objeto corrente possui o nome *"abcd"*. Neste caso a conversão de uma referência a atributo Nome é realizada, uma vez que o tipo da comparação é **Seqüência**.

```
Converte Seqüência [ Nome ] == "abcd"
```

resulta em **Verdadeiro** se o objeto corrente possui o nome *"abcd"*. Neste caso a conversão de uma referência a atributo Nome para seqüência é realizada em virtude do operador **Converte**.

Tipo origem	Tipo resultado	Como opera

Lógico	Inteiro	Se **Verdadeiro**, resulta em 1 Se **Falso**, resulta em 0
Lógico	Seqüência	Primeiro converte para **Inteiro**, depois para **Seqüência**
Inteiro	Lógico	Se **==** 0, resulta em **Falso**, Se **!=** 0, resulta em **Verdadeiro**
Inteiro	Seqüência	Gera uma seqüência contendo o número convertido para decimal, se necessário precedido por menos. O tamanho da seqüência será tão grande quanto necessário para conter o sinal opcional e todos os dígitos. Não são gerados caracteres 0 antes do primeiro não nulo. Se o valor inteiro for 0 será gerado "0"
Seqüência	Lógico	Primeiro converte para **Inteiro**, depois para **Lógico**
Seqüência	Inteiro	Caso a seqüência seja composta somente por dígitos, possivelmente precedidos por sinal, efetua a conversão Caso contrário, sinaliza erro de execução
Seqüência	Atributo Nome	Caso a classe de objetos admita dicionário e se a seqüência for diferente dos demais nomes de objetos deste dicionário, troca o nome. Caso contrário, sinaliza erro de execução
Seqüência	Atributo Aliás	Caso a seqüência tenha tamanho maior ou igual a 1 será atribuída ao Aliás. Caso contrário, o Aliás será excluído
ListaTexto	Atributo Texto	Caso a ListaTexto tenha tamanho maior ou igual a 1 será atribuída ao texto. Caso contrário, o texto ser excluído
ListaTexto	Atributo Relação	Cada linha não vazia da ListaTexto será considerada como um nome de objeto na classe destino da relação. Caso este nome já exista, será incluída na relação uma referência ao objeto correspondente. Caso o nome ainda não exista, será criado um objeto com este nome e a referência a este objeto será incluída na relação. Se existirem referências múltiplas a um mesmo nome, a relação também conterá referências múltiplas.
Atributo Nome	Seqüência	Busca o nome e atribui o valor à seqüência. Gera erro de execução caso o nome não exista. Isto pode ocorrer se o objeto corrente for **nulo**.
Atributo Aliás	Seqüência	Caso o Aliás exista, busca e atribui o valor à seqüência. Caso o Aliás não exista, gera a seqüência vazia.
Atributo Texto	ListaTexto	Caso o texto exista, busca e atribui o valor à lista de texto. Caso o texto não exista, esvazia a lista de texto.
Atributo Relação	ListaTexto	Caso a relação exista, cria uma lista de texto em que cada linha é o nome de um dos objetos referenciados na relação. Caso a relação não exista, esvazia a lista de texto.

7.19.5 Elementos de expressões

```
Elemento          :=   Constante
                     | IdentificadorAtrib
                     | IdentificadorDic
                     | IdentificadorRel
                     | IdentificadorVar
```

```
                        | Valor_base
                        | Chama_função
                        | "("Expressão ")" ;
Constante           :=   CteNúmero
                        | CteSeqüência
                        | Verdadeiro
                        | Falso
                        | Corrente ;
Valor_Base          := "[" Atributo_BSW "]" ;
Chama_procedimento  := IdentificadorFunc Parametros_Atuais ;
Chama_função        := IdentificadorFunc Parametros_Atuais ;
```

As constantes definidas são:

Verdadeiro valor lógico.

Falso valor lógico.

Corrente é a referência ao objeto corrente.

Em adição a estas constantes, estão definidas, ainda, as seguintes classes de constantes:

IdentificadorAtrib discrimina um atributo **Aliás** ou fragmento de **Texto**. Estes valores são convertidos para **Inteiro** caso necessário.

IdentificadorRel discrimina um atributo **Relação**. Estes valores são convertidos para **Inteiro** caso necessário.

IdentificadorDic identifica o dicionário de uma classe de objetos. Estes valores não são convertidos automaticamente.

Elementos IdentificadorVar designam variáveis. Todas as variáveis devem ser declaradas. Em uma expressão, as variáveis devem corresponder ao tipo da expressão. A definição de tipos é da esquerda para a direita, obedecendo a ordem de prioridade. Por exemplo, assumindo S sendo uma variável tipo **Seqüência** e I uma variável tipo **Inteiro**:

S == I converte I para Seqüência e depois compara

I == S converte S para Inteiro e depois compara

O texto "**[**"*Atributo_BSW* "**]**" referencia um atributo do objeto corrente contido na base de software. Para referenciar atributos de outros objetos, deve-se navegar para estes objetos. Isto pode ser feito, ou utilizando o comando **ComObjeto**, ou, então, ativando um formulário com o comando **FrmElem** identificando o objeto desejado.

O valor correspondente a um atributo é buscado, ou se for necessário converter, ou se for atribuído a outro valor. Parâmetros do tipo atributo são referências a estes atributos. Para que seja passado o valor é necessário, ou que a declaração do formulário correspondente anteceda o seu uso, ou que seja usada a conversão explícita.

A tabela a seguir identifica os tipos de cada categoria de atributo.

Categoria do atributo	Tipo do valor
Nome	**Seqüência**
Aliás	**Seqüência**
Texto	**ListaTexto**
Relação	**ListaTexto**

Cabe salientar que, para operar com atributos contidos na base de software, é necessário colocar colchetes em volta da referência ao atributo. Para editar atributos contidos na base de software, é necessário colocar um caractere ";" após à referência. Finalmente, para atribuir a um atributo contido na base de software, é necessário colocar um caractere "=" após a referência.

Exemplos:

```
ValNome = [ Nome ] ;
```

busca o valor do nome do objeto corrente e atribui-o à variável *ValNome*. Esta variável deve ser do tipo seqüência.

```
Nome = ValNome ;
```

atribui a seqüência contida em *ValNome* ao atributo Nome do objeto corrente.

```
Nome ;
```

edita o nome do objeto corrente.

8. FUNÇÕES INTERNAS

Todas as funções de Talisman são definidas pelo sistema. Não existe a possibilidade do usuário definir suas próprias funções. Para obter um efeito semelhante ao de chamada de funções, use formulários retornando valores em variáveis globais.

Procedimentos são funções que não retornam valor (funções tipo **Void**). Chamadas de procedimentos são, portanto, comandos executáveis. Não é permitido o uso de funções como se fossem procedimentos. Ou seja, funções podem ser chamadas somente em expressões. Tampouco é permitido que se use procedimentos como funções.

Nas seções a seguir serão definidas as funções existentes. Nestas definições são usados tipos especiais com o seguinte significado:

Void sem valor, corresponde ao "valor" retornado por um procedimento. Todas as funções que retornem **Void** são procedimentos.

ListaObjeto é uma lista de objetos. Corresponde aos tipos **Fila**, **Pilha**, **ConjuntoFila**, e **ConjuntoPilha**. Ao passar um parâmetro **ListaObjeto** é indiferente a política de inserção da lista, uma vez que isto é uma propriedade interna da própria lista.

Atributo é a referência a atributo de objeto. Valores deste tipo valem somente em listas de parâmetros de funções.

8.1 Funções sobre seqüências

```
ConcatSeq( Seqüência A , Seqüência B ) -> Seqüência
```

retorna a seqüência formada pela concatenação das seqüências A e B. O tamanho resultante não deve ultrapassar 250 caracteres.

```
ConvCaixa( Seqüência A ) -> Seqüência
```

retorna a seqüência formada pela conversão de todas as letras minúsculas, acentuadas ou não, para a correspondente letra maiúscula não acentuada.

```
ConvFormato( Inteiro Número , Seqüência Chars ,
             Inteiro Tamanho ) -> Seqüência
```

retorna uma seqüência de Tamanho caracteres formada pela conversão de *Número* para um valor formado pelos caracteres de *Chars*. O primeiro caractere de *Chars* é o caractere a ser utilizado para denotar os zeros à esquerda. Os demais caracteres são utilizados para denotar a conversão. A base para a qual é convertida é igual a TamSeq(*Chars*) - 1. Não ocorre conversão caso a base seja menor do que 2.

Exemplos

```
ConvFormato( 101 , "*01234567" , 15 )
```

retorna: "**********145"

```
ConvFormato( -101 , " 01234" , 7 )
```

retorna: " -401"

```
Indice( Seqüência Base , Seqüência Procurada ,
        Inteiro LimInferior , Inteiro LimSuperior ) -> Inteiro

IndiceParc( Seqüência Base , Seqüência Procurada ,
            Inteiro LimInferior , Inteiro LimSuperior ) -> Inteiro
```

Procura a seqüência *Procurada* na seqüência *Base*, iniciando a procura no caractere *LimInferior* e terminando no caractere *LimSuperior*. Caso a seqüência *Procurada* seja encontrada, retorna o índice do caractere inicial da primeira ocorrência. Caso não seja encontrada retorna -1. O primeiro caractere de uma seqüência tem índice 1. O parâmetro *LimSuperior* delimita o último caractere inclusive da seqüência *Base*, no qual poderia **iniciar** a seqüência procurada.

Na forma *Indice* a função compara caracteres diferenciando maiúsculas, minúsculas e letras acentuadas. Na forma *IndiceParc*, letras maiúsculas, minúsculas, acentuadas ou não, são consideradas iguais.

Exemplos:

```
Indice( "Abcdebcdebcd" , "bcd" , 1 , 4 )
```

retorna 2

```
Indice( "Abcdebcdebcd" , "bcd" , 4 , 30 )
```

retorna 6

```
Indice( "Abcdebcdebcd" , "Bcd" , 4 , 30 )
```

retorna -1

```
IndiceParc( "Abcdebcdebcd" , "Bcd" , 4 , 30 )
```

retorna 6

```
SubSeq( Seqüência Seq , Inteiro Origem ,
        Inteiro Tamanho ) -> Seqüência
```

retorna a sub-seqüência de *Seq* iniciando no caractere de índice *Origem* (primeiro índice é 1) e tendo *Tamanho* caracteres. Caso a definição da sub-seqüência ultrapasse o domínio de *Seq*, a seqüência retornada será truncada de modo que fique no domínio de *Seq*.

Exemplo:

```
SubSeq( "Abcdebcdebcd" , 2 , 4 )
```

retorna "bcde"

```
TamSeq( Seqüência Seq ) -> Inteiro
```

retorna o número de caracteres da seqüência *Seq*. Se *Seq* for vazia, retorna 0.

8.2 Funções sobre objetos

```
CriaNome( Dicionário IdDicionário, Seqüência Nom ) -> Objeto
```

cria um objeto com nome igual a *Nom* no dicionário *IdDicionário*. Caso o nome já exista no dicionário, será retornada a referência ao objeto correspondente. Caso o nome não exista, o objeto será criado e sua referência será retornada.

Exemplo:

```
CriaNome( DicProcesso , "Linearizar módulo" )
```

gera o processo "linearizar módulo" caso este ainda não exista no dicionário de processos.

```
Existe( Atributo Atrib ) -> Lógico
```

caso a referência ao atributo *Atrib* do objeto corrente exista na base de software, a função retornará Verdadeiro, caso não exista retornará Falso. Lembre-se que a referência ao atributo é definida na forma [ClasseAtributo Expressão].

Por exemplo, caso o *texto 1* do objeto corrente exista na base de software

```
Existe( [ Texto 1 ] )
```

retornará **Verdadeiro**.

```
ExisteObj( Objeto Obj ) -> Lógico
```

retorna **Verdadeiro** se o objeto ou instância de objeto *Obj* for não nulo.

`FolhaFilho( Objeto Obj) -> Objeto Folha`

retorna o objeto *folha* que especifica o objeto *Obj*. Caso o objeto *Obj* não possua folha filho, o objeto folha retornado será nulo.

`FolhaInst( Objeto InstObj ) -> Objeto`

retorna o objeto *folha* que contém a instância de objeto *InstObj*. Caso *InstObj* não seja uma instância de objeto, isto é, caso não seja um elemento de diagrama, o objeto retornado será nulo.

`FolhaPai( Objeto InstObj ) -> Objeto Folha`

retorna o objeto folha que contém a instância pai da folha que, por sua vez, contém a instância *InstObj*. Caso a folha que contém a instância de objeto *InstObj* não possua instância pai, o objeto retornado será nulo.

`InstânciaPai( Objeto InstObj1 ) -> Objeto InstObj2`

retorna a instância de objeto pai da folha que contém a instância InstObj1. Caso a folha que contém a instância de objeto InstObj1 não possua instância pai, a instância de objeto retornada será nula.

`LigaçãoEntra( Objeto Ligação , Objeto InstObj ) -> Lógico`

retorna **Verdadeiro** caso a (instância de) ligação *Ligação* seja orientada para dentro, da instância de objeto *InstObj*. Ligações com orientação dupla, também são orientadas para dentro. As duas instâncias devem pertencer à mesma folha.

`LigaçãoSai( Objeto Ligação , Objeto InstObj ) -> Lógico`

retorna **Verdadeiro** caso a (instância de) ligação *Ligação* seja orientada para fora, da instância de objeto *InstObj*. Ligações com orientação dupla, também são orientadas para fora. As duas instâncias devem pertencer à mesma folha.

`ModoLig( Objeto Ligação ) -> Inteiro`

retorna um número indicativo da direção associada à *Ligação*. O valor retornado está em correspondência com a seleção de direção realizada ao editar a ligação. Na prática esta função serve somente para determinar se a ligação não possui direção ou se possui direção dupla, uma vez que não é possível saber qual objeto foi origem ao criar a ligação com direção simples. Os valores são:

0	ligação sem direção
1	ligação para a origem
2	ligação para o destino
3	ligação dupla

`ObjetoInst( Objeto InstObj ) -> Objeto`

retorna a referência do objeto correspondente à instância de objeto *InstObj*. Lembre-se que, em diagramas figuram instâncias de objeto. As referências a instâncias de objeto são diferentes das referências a objetos. Esta função efetua a conversão. Caso a instância de objeto seja não dicionarizada, o objeto retornado é nulo.

`ObjetoNome( IdDicionário IdDic , Seqüência Nom ) -> Objeto`

retorna a referência ao objeto de nome *Nom* contido no dicionário de classes de objeto *IdDic*. Se o objeto não existe, retorna nulo.

`ObjetoPai( Objeto InstObj ) -> Objeto`

retorna a referência do objeto pai da folha que contém a instância InstObj.

`ObjetosIguais( Objeto Obj1 , Objeto Obj2 ) -> Lógico`

retorna **Verdadeiro** se *Obj1* e *Obj2* referenciam o mesmo objeto. Cabe salientar que instâncias de objetos são diferentes dos objetos que instanciam. Para verificar se duas instâncias são relativas o mesmo objeto, compare a igualdade dos resultados da função `ObjetoInst` descrita acima.

`OutraPonta( Objeto Ligação , Objeto InstOrigem ) -> Objeto`

retorna a instância de objeto ligada à instância *InstOrigem* pela ligação *Ligação*. Ambos *Ligação* e *InstOrigem* devem pertencer à mesma folha. Se *Ligação* não for uma das ligações que atingem a instância *InstOrigem*, retorna o objeto nulo.

`RotuloLig( Objeto Ligação ) -> Objeto`

retorna a referência à instância de rótulo associado à ligação *Ligação*. Caso a ligação não possua rótulo, retorna o objeto nulo.

`NomeArqObj( Objeto Obj ) -> Seqüência`

retorna o nome completo do arquivo base de software onde se encontra o objeto *Obj*.

`ObjetoBase( Objeto Obj ) -> Objeto`

retorna a referência ao objeto descritor da base de software na qual figura o objeto *Obj*. Caso a base de software não possua objeto descritor, retorna o objeto nulo. O objeto descritor da base de software é criado e editado através de linguagem de representação Identificação da Base.

8.3 Funções sobre listas de objetos

`EsvaziaLista( ListaObjáLista ) -> Void`

torna vazio o conteúdo da lista de objetos *Lista*. Note que ao iniciar a execução as listas são criadas vazias.

`ListaVazia( ListaObj Lista ) -> Lógico`

retorna **Verdadeiro** se *Lista* estiver vazia.

`InsereObj( ListaObj Lista , Objeto Obj) -> Void`

introduz a referência ao objeto, ou instância de objeto, *Obj* na lista de objetos *ListaObj*. A inserção se dará respeitando a política de inserção associada a *Lista*.

`RetiraObj( ListaObj Lista ) -> Objeto`

retorna o primeiro objeto da lista de objetos *Lista*, retirando-o desta lista. Caso a lista esteja vazia, retorna o objeto nulo.

`PertenceObj( ListaObj Lista , Objeto Obj ) -> Inteiro`

retorna o primeiro índice no qual a referência a objeto *Obj* figura na lista *Lista*. Retorna 0 se o objeto não figura na lista.

`InsereInxObj( ListaObj Lista , Inteiro Inx , Objeto Obj ) -> Void`

insere o objeto *Obj* na *Inx*-ésima posição na *Lista*. O objeto que estava nesta posição terá sido movido uma posição para cima. Se *Inx* for menor ou igual a zero insere antes do primeiro elemento da lista. Se for maior do que o tamanho da lista, insere após o último elemento da lista.

`ObjetoLista( ListaObj Lista , Inteiro Inx ) -> Objeto`

retorna a *Inx*-ésima referência a objeto contida em *Lista*. O primeiro objeto corresponde ao índice 1. Caso *Inx* esteja fora do domínio da lista retorna o objeto nulo.

`ExcluiObj( ListaObj Lista , Inteiro Inx ) -> Void`

Exclui o *Inx*-ésimo objeto da lista de objetos *Lista*. Se *Inx* estiver fora dos domínios da lista, faz nada.

`TamLista( ListaObj Lista ) -> Inteiro`

retorna o número de objetos contidos em Lista.

```
ObjRelInx( AtributoRelação IdRel , Objeto Obj ) -> Inteiro
```

retorna o índice em que o objeto *Obj* figura na relação *IdRel*. Caso o objeto não figure na relação, retorna -1. Lembre-se que a forma de identificar um atributo relação é [Relação IdRel].

```
RelInx( AtributoRelação IdRel , Inteiro Inx ) -> Objeto
```

retorna o objeto que se encontra no índice *Inx* da relação *IdRel*. Caso *Inx* esteja fora do domínio da relação *IdRel*, retorna o objeto nulo. O primeiro índice tem valor 0.

```
TamRel( AtributoRelação IdRel ) -> Inteiro
```

retorna o número de referências a objetos contido na relação *IdRel* do objeto corrente. Lembre-se que a forma de identificar um atributo relação é [Relação IdRel].

```
CopiaRel( ListaObj Lista , AtributoRelação IdRel ) -> Void
```

Copia os objetos contidos no atributo relação *IdRel* para *Lista*. A cópia respeita as políticas de inserção de *Lista*: *pilha*, *fila* e *conjunto*.

8.4 Funções sobre listas de texto

```
ConcatTexto( ListaTexto A , ListaTexto B ) -> Void
```

concatena a lista texto *B* ao final da lista texto *A*.

```
EsvaziaTexto( ListaTexto Text ) -> Void
```

torna vazio o conteúdo da lista de texto *Text*. Note que ao iniciar a execução as listas são criadas vazias.

```
InsereTexto( ListaTexto Text , Inteiro Inx ,
             Seqüência Linha ) -> Void
```

insere a seqüência *Linha* na *Inx*-ésima linha da lista de texto *Text*. Se *Inx* for menor ou igual a zero, insere antes da primeira linha de *Text*. Se *Inx* for maior do que o tamanho da lista, insere após a última linha de *Text*.

```
LinhaTexto( ListaTexto Text , Inteiro Inx ) -> Seqüência
```

retorna a seqüência de caracteres correspondente à *Inx*-ésima linha de *Text*. A primeiro a linha possui índice = 1. Se o índice estiver fora do domínio de *Text*, retorna a seqüência vazia.

```
ExcluiTexto( ListaTexto Text , Inteiro Inx ) -> Void
```

exclui a linha de texto *Inx* de *Text*. Se *Inx* estiver fora do domínio da lista, exclui nada. A primeira linha possui o índice 1.

```
TamTexto( ListaTexto Text ) -> Inteiro
```

retorna o número de linhas contidas em *Text*.

8.5 Funções sobre dicionários de objetos

```
ClasseInt( Dicionário IdDic ) -> Inteiro
```

retorna um número único associado ao dicionário *IdDic*. O valor *IdDic* pode ser resultado de uma expressão, ou pode ser um dos nomes de dicionários, IdentificadorDic, definidos na base de conhecimento. Para uma dada classe de objetos este número será sempre o mesmo, independente da linguagem de representação usada.

```
ClasseNome( Seqüência NomeClasse) -> Dicionário
```

retorna a identificação do dicionário correspondente ao nome simbólico *NomeClasse*. Caso não exista dicionário com o nome *NomeClasse* gera mensagem de erro de execução.

`ClasseObj( Objeto Obj ) -> Dicionário`

retorna a identificação do dicionário do objeto *Obj*.

`NomeClasse( Dicionário IdDic ) -> Seqüência`

retorna o nome simbólico do dicionário de classes de objeto *IdDic*.

`CopiaDic( ListaObj Lista , Dicionário IdDic ) -> void`

Copia os nomes contidos no dicionário *IdDic* para *Lista*. A cópia respeita as propriedades pilha, fila e conjunto de *Lista*.

9. PROGRAMAÇÃO USANDO DIAGRAMAS

Nesta seção serão descritos os elementos e as *relações gráficas* utilizados ao programar formulários envolvendo linguagens de representação gráficas. Estes elementos e relações são utilizados, por exemplo, para *acessar* relacionamentos definidos através de diagramas para os formulários de especificação; para *validar* diagramas e especificações; para *linearizar* diagramas; etc.

As linguagens de representação gráficas estabelecem relacionamentos através das ligações e do nivelamento de diagramas. Todas as linguagens de representação gráficas utilizam as mesmas relações gráficas. Os demais atributos, tais como textos, aliases e relações de dicionário dependem das linguagens de representação específicas e serão descritas nos capítulos que tratam das linguagens de representação suportadas.

Em adição às relações gráficas descritas nesta seção, estão disponíveis diversas funções através das quais se poderá navegar a partir de instâncias de objetos. Estas funções estão descritas na seção Funções Internas deste manual.

Cabe salientar que o domínio de relações gráficas são instâncias de objeto. Os objetos correspondentes a estas instâncias podem pertencer a diversas classes de objetos. Para saber a classe do objeto de uma determinada instância, devem ser utilizadas a função ClasseObj (*classe de objeto*). Para se ter acesso ao objeto correspondente a uma determinada instância, é necessário converter de instância de objeto para objeto, o que é realizado pela função ObjetoInst (*Objeto instanciado*). Cabe salientar que existem instâncias para as quais não estão definidos objetos. Por exemplo, *Comentários*, *Pontos de confluência* e *Cardinalidades* não são dicionarizadas e, conseqüentemente, não possuem objeto. Use a função ExisteObj (*Existe o objeto*) para determinar se a instância existe, e/ou se o objeto instanciado existe.

9.1 Elementos de linguagens gráficas

Folha superfície imaginária sobre a qual é desenhado um diagrama.

Instância de objeto referência, contida em uma folha, a um elemento da linguagem de representação correspondente ao diagrama.

Ligação relacionamento entre duas instâncias de objeto.

Rótulo instância de elemento associada a uma ligação. O rótulo qualifica a ligação.

9.2 Relações gráficas

Relações gráficas são relações envolvendo *folhas*, *instâncias* ou *rótulos*. Relações gráficas são criadas, alteradas e excluídas exclusivamente através do Editor de Diagramas. Relações gráficas não podem ser criadas e/ou alteradas através do Editor de Formulários nem através do Editor de Estruturas. Todavia, podem ser utilizadas em formulários, e, conseqüentemente, podem ser utilizadas pelo Editor de Formulários, como instrumentos de navegação.

InstsObjeto é a relação de todas as instâncias do elemento corrente (rótulo ou objeto), considerando todas as folhas da linguagem de representação corrente. Nesta relação não ocorrem instâncias de imagem de objeto. A origem da relação é elemento de linguagem de representação o destino é instância de elemento.

InstFolha é a relação de todas as instâncias de objeto existentes na folha corrente. A origem da relação é folha, o destino é instância de objeto.

LigFolha é a relação de todas as ligações existentes na folha corrente. A origem da relação é folha, o destino é instância de ligação.

LigsInstância é a relação de todas as ligações que atingem a instância de objeto corrente. A origem da relação é instância de objeto, o destino é instância de ligação.

LigsEntra é a relação de todas as ligações que atingem a instância de objeto corrente e que são direcionadas para este elemento (direção: *entrada* ou *dupla*). A origem da relação é instância de objeto, o destino é instância de ligação.

LigsSai é a relação de todas as ligações que atingem a instância de objeto corrente e que são direcionadas para fora (direção: *saída* ou *dupla*). A origem da relação é instância de objeto, o destino é instância de ligação.

InstsDaLigacao é a relação de todas as instâncias de objeto ligados pela ligação corrente (são sempre duas). A origem da relação é ligação, o destino é instância de objeto.

InstsLigadas é a relação de todas as instâncias de objeto que podem ser alcançadas por intermédio de uma ligação vinculada à instância de objeto corrente. De uma forma intuitiva, corresponde a todas as instâncias de objeto que se encontram na *outra ponta* das ligações que atingem a instância de objeto corrente.

10. EXEMPLOS

10.1 Formulário de edição

O formulário a seguir edita a especificação de um processo na linguagem de representação Fluxo de Dados.

```
InicFrm "Editar processo"
   Título "Nome do processo: " ;
   NãoAvLin ;
   Nome ;
   Título "Laudo de validação" ;
   TextoLaudo ;
   Título "Descrição do processo" ;
   Texto TxtDescr ;
   Título "Requisitos do processo" ;
   Texto TxtRequis ;
   Título "Hipóteses do processo" ;
   Texto TxtHipot ;
   Título "Descrição formal do processo" ;
   Texto 11 ;
   Título "Agentes responsáveis pelo processo" ;
   Relação Agentes ;
   Título "Relacionamentos a que corresponde" ;
   Relação Relacionamentos ;
   Título "Restrições de projeto" ;
   Texto TxtRestrProj;
   Título "Observações" ;
   Texto TxtObserv ;
FimFrm ;
```

10.2 Formulário de validação

Os formulários a seguir verificam se os dados que fluem para dentro de um depósito de dados podem ser armazenados neste depósito. Este exemplo ilustra, ainda, o uso de relações gráficas.

```
/************************************************
* Declarações globais
************************************************/

InicGlobal
      ConjuntoFila DadoEntra ;
   FimDecl
FimFrm

/************************************************
* Registrar todos os dados que fluem para o objeto
************************************************/

InicFrm "Registrar dados que fluem"
   EsvaziaLista( DadoEntra ) ;
   /* Objeto corrente pode ser instanciado */
   ParaTodos InstsObjeto Faz
      /* Objeto corrente é instância de objeto */
      ParaTodos LigsEntra Faz
         /* Objeto corrente é instância de ligação */
         ComObjeto RotuloLig( Corrente ) Faz
            /* Objeto corrente é instância de rótulo
               Será o objeto nulo, caso a ligação
               não possua rótulo */
            Se ExisteObj( Corrente )
            Então
               /* A ligação possui rótulo */
               ComObjeto ObjetoInst( Corrente ) Faz
                  /* Objeto corrente é
```

```
                        especificação de rótulo */
                     ParaTodos Estruturas Faz ;
                        InsereObj(DadoEntra, Corrente) ;
                     Fim ;
                  Fim ;
               Senão
                  Título "Ligação sem rótulo" ;
               Fim ;
            Fim ;
         Fim ;
      Fim ;
FimFrm

/*****************************************
* Validar se dados contidos no depósito são recebidos
*****************************************/

InicFrm "Validar depósito"
   Frm "Registrar dados que fluem" ;
   ParaTodos Dados Faz
      Se Não PertenceObj( DadoEntra , Corrente )
      Então
         Título "Dado a seguir não é transmitido para o depósito" ;
         Nome ;
      Senão
      Fim ;
   Fim ;
FimFrm
```

10.3 Fragmento de linearizador

O fragmento de programa de formulário a seguir lineariza uma estrutura de decomposição (relação **Decomp**) controlando e evitando a linearização múltipla de elementos. A linearização múltipla ocorre em estruturas recursivas (filho de elemento é antecessor deste elemento).

```
/************************************************
* Declarações globais
************************************************/

InicGlobal
      Fila Processados ;
   FimDecl
FimFrm

/************************************************
* Cabeçalho de linearização
************************************************/

InicFrm "Iniciar linearização"
   EsvaziarLista( Processados ) ;
   Título "Estrutura de decomposição de: " ;
   Nome ;
   Frm "Processar filhos" ;
FimFrm

/****************************************
* Este formulário lineariza a estrutura controlando recursão
****************************************/

InicFrm "Processar filhos"
   AvMrgEsq 3 ;
   ParaTodos Decomp Faz
      Se PertenceObj( Processados , Corrente )
      Então
         Título "Já foi processado " ;
         NãoAvLin ;
```

```
            Nome ;
         Senão
            InsereObj( Processados , Corrente ) ;
            Título "Linearizando ";
            NãoAvLin ;
            Nome ;
            Frm "Processar filhos" ;
         Fim ;
      Fim ;
   FimFrm
```

10.4 Listagem de laudos

O formulário a seguir exibe todos os laudos associados a objetos da classe do objeto corrente. A listagem recebe um título da classe de objetos caso exista pelo menos um objeto com laudo. Por construção do dicionário, a listagem estará em ordem alfabética.

```
   InicFrm "Exibir laudo da classe do objeto corrente"
         Dicionário  Dic ;
         Lógico      Primeiro ;
      FimDecl
      Endenta 3 ;
      Primeiro = Verdadeiro ;
      Dic = ClasseObj( Corrente ) ;
      ParaTodos Dic Faz
         Se Existe( [ Texto TxtLaudo ] )
         Entao
            Se Primeiro
            Então
               Primeiro = Falso ;
               Título "Laudo dos objetos da classe " ;
               NãoAvLin ;
               Título NomeClasse( Dic ) ;
            Senão
            Fim ;
            Nome ;
            AvMrgEsq  3 ;
            TextoLaudo ;
            AvMrgEsq -3 ;
         Senão
         Fim ;
      Fim ;
   FimFrm
```

11. APÊNDICE: PALAVRAS RESERVADAS

11.1 Palavras reservadas da linguagem

Aliás	AvEsq	AvMrgEsq
BAX	BCO	BSW
ComObjeto	ConjuntoFila	ConjuntoPilha
Constante	Converte	Corrente
Dicionário	E	Endenta
Enquanto	Então	Executa
Falso	Faz	Fila
Fim	FimDecl	FimFrm
Frm	FrmElem	GeraLaudo
GeraTexto	GeraVar	IMPLICA
InicExterno	InicFrm	InicGlobal
Inteiro	ListaTexto	Lógico
MacroNome	MacroTecla	MacroTexto
MrgDir	MrgEsq	NAO
Nome	Novo	NãoAvLin
Objeto	OU	OUEX
ParaTodos	Pilha	Protegido
Relação	RESTO	SaiRepetição
Se	Senão	Seqüência
TamCampo	Texto	TextoEsq
TextoLaudo	Título	USO
Valor	Variável	Verdadeiro

11.2 Nomes de aliases globais

AliásArquivo	AliásIdent	AliásSigla

11.3 Nomes de fragmentos de texto globais

TxtDescr	TxtHipót	TxtLaudo
TxtObjt	TxtObserv	TxtReqProt
TxtRequis	TxtRestrImpl	TxtRestrProj

11.4 Nomes de relações gráficas

InstFolha	InstsDaLigação	InstsLigadas
InstsObjeto	LigFolha	LigsEntra
LigsInstância	LigsSai	

11.5 Nomes de relações

Absorve
Absorvido
AbsorvidoEntidRelação
Agentes
AgregaEntidLigação
AgregaRelacionamentos
Areas
Arquivos
Assuntos
AtivaFluxos
AtivAnteriores
AtivaProcessos
Atividades
AtivLideradas
AtivPosteriores
Autores
Bancos
Blocos
BlocosDeclaram
BlocosGeram
BlocosValidam
Camadas
Capítulos
Cargos
Categorias
Cenas
Chama
Chamado
ChavesDet
ChavesDetDepósito
ChavesDetEntid
ChavesDetEntidLigação
ChavesDetRelac
ChavesEst
ChavesEstEntid
ChavesEstEntidLigação
ChavesSec
ChavesSecDepósito
ChavesSecEntid
ChavesSecEntidLigação
ChavesSecRelac
Classes
ColaboraCom
Colaboradora
Comp
Composições
Controladores
ControlaEventos
Controles
Dados
DadosInterface
DeclaradosEm
Decomp
DefineEntidade
DefineRelacionamento
DefinidoEm
Depende
DependFunc
Depósitos
Documentos
Editoras
EhCategoria
EhDependente
EhDependFunc
Entidades
EntidFortes
EntidFracas
EntidLigação
EntidLigaçãoAgregadas
Espaços
Estados
Eventos
EventosAtivam
EventosSinalizados
Exemplos
Figuras
Fluxos
FluxosGeram
GeraArquivos
GeraControles
GeraDados
GeraDepósitos
GeraEntidades
GeraEntidLigação
GeraRelacionamentos
GeraTipos
Hipóteses
Impactos
Indices
InterfaceMódulo
Interfaces
Janelas
Lidera
Módulos
MódulosGeram
MódulosValidam
Notas
Objetivos
Orgãos
Pacotes
PacotesGeram
PacotesInterface
PacotesValidam
Palavras
Papéis
Parágrafos
Pessoas
Processos
ProcessosAtivados
ProcessosGeram
ProcessosValidam
ProcLiderados
ProcsControle
Produtos
Projetos
ProjLiderados
Recursos

Redefine
RedefinidaPor
Relacionamentos
RelacsAgregados
Requer
Requerido
Requisitos
Restrições
RetornaClasse
RetornadoPor
RetornaTipo
SinalizaEventos
Sobrecarrega
SobrecarregadaPor
TemCategoria
Testes
Tipos
TiposInterface
Tomadas
Transições
UsaClasse
UsadoPor
UsaTipo
ValidaArquivos
ValidaDados
ValidaDepósitos
ValidaEntidades
ValidaEntidLigação
ValidaRelacionamentos
ValidaTipo
Visões

11.6 Nomes de dicionários

DicAgente
DicAreaInteresse
DicArquivo
DicAssunto
DicAtividade
DicAutor
DicBancoDados
DicBlocoPrograma
DicCamadas
DicCapítulo
DicCargoPessoa
DicCategoria
DicCenas
DicClasses
DicComentario
DicComposição
DicControlador
DicControle
DicDado
DicDepósito
DicDocumento
DicEditora
DicEntidade
DicEntidadeLigação
DicEspaçoDados
DicEstado
DicEvento
DicExemplos
DicFigura
DicFluxoDado
DicHipótese
DicImpactos
DicIndiceAcesso
DicInterfExterna
DicJanela
DicMódulo
DicNotaRodaPé
DicObjetivo
DicOrgãoEmpresa
DicPacotePrograma
DicPalavraChave
DicPapéis
DicParágrafo
DicPessoa
DicProcesso
DicProduto
DicProjeto
DicRecurso
DicRelacionamento
DicReqNãoFunc
DicRequisito
DicRestrição
DicTestes
DicTipoDados
DicTomadas
DicTransição
DicVista

11.7 Tabela de funções

ClasseInt retorna: Inteiro
recebe: Id dicionário

ClasseNome retorna: Id dicionário
recebe: Seqüência

ClasseObj retorna: Id dicionário
recebe: Objeto de BSW

ConcatSeq retorna: Seqüência
recebe: Seqüência Seqüência

ConcatTexto retorna: Void
recebe: Lista texto Lista texto

ConvCaixa retorna: Seqüência
recebe: Seqüência

ConvFormato retorna: Seqüência
recebe: Inteiro Seqüência Inteiro

CopiaDic retorna: Void
recebe: Lista Id dicionário

CopiaRel retorna: Void
recebe: Lista Atributo de BSW

CorrRelInx retorna: Void
recebe: Atributo de BSW Inteiro

CriaNome retorna: Objeto de BSW
recebe: Id dicionário Seqüência

EsvaziaLista retorna: Void
recebe: Lista

EsvaziaTexto retorna: Void
recebe: Lista texto

ExcluiObj retorna: Void
recebe: Lista Inteiro

ExcluiTexto retorna: Void
recebe: Lista texto Inteiro

Existe retorna: Lógico
recebe: Atributo de BSW

ExisteObj retorna: Lógico
recebe: Objeto de BSW

FolhaFilho retorna: Objeto de BSW
recebe: Objeto de BSW

FolhaInst retorna: Objeto de BSW
recebe: Objeto de BSW

FolhaPai retorna: Objeto de BSW
recebe: Objeto de BSW

Indice retorna: Inteiro
recebe: Seqüência Seqüência Inteiro Inteiro

IndiceParc retorna: Inteiro
recebe: Seqüência Seqüência Inteiro Inteiro

InsereInxObj retorna: Void
recebe: Lista Inteiro Objeto de BSW

InsereObj retorna: Void
recebe: Lista Objeto de BSW

InsereTexto retorna: Void
recebe: Lista texto Inteiro Seqüência

InstanciaPai retorna: Objeto de BSW
recebe: Objeto de BSW

LigaçãoEntra retorna: Lógico
recebe: Objeto de BSW Objeto de BSW

LigaçãoSai retorna: Lógico
recebe: Objeto de BSW Objeto de BSW

LinhaTexto retorna: Seqüência
recebe: Lista texto Inteiro

ListaVazia retorna: Lógico
recebe: Lista

ModoLig retorna: Inteiro
recebe: Objeto de BSW

NomeArqObj retorna: Seqüência
recebe: Objeto de BSW

NomeClasse retorna: Seqüência
recebe: Id dicionário

ObjetoBase retorna: Objeto de BSW
recebe: Objeto de BSW

ObjetoInst retorna: Objeto de BSW
recebe: Objeto de BSW

ObjetoLista retorna: Objeto de BSW
recebe: Lista Inteiro

ObjetoNome retorna: Objeto de BSW
recebe: Id dicionário Seqüência

ObjetoPai retorna: Objeto de BSW
recebe: Objeto de BSW

ObjetosIguais retorna: Lógico
recebe: Objeto de BSW Objeto de BSW

ObjRelInx retorna: Inteiro
recebe: Atributo de BSW Objeto de BSW

OutraPonta retorna: Objeto de BSW
recebe: Objeto de BSW Objeto de BSW

PertenceObj retorna: Inteiro
recebe: Lista Objeto de BSW

RelInx retorna: Objeto de BSW
recebe: Atributo de BSW Inteiro

RetiraObj retorna: Objeto de BSW
recebe: Lista

RotuloLig retorna: Objeto de BSW
recebe: Objeto de BSW

SubSeq retorna: Seqüência
recebe: Seqüência Inteiro Inteiro

TamLista retorna: Inteiro
recebe: Lista

TamRel retorna: Inteiro
recebe: Atributo de BSW

TamSeq retorna: Inteiro
recebe: Seqüência

TamTexto retorna: Inteiro
recebe: Lista texto