

9. Como implementar um Teclado Virtual musical;
10. Como implementar interfaces com pentagrama musical utilizando fontes com símbolos musicais;
11. Como implementar uma interface para ler, editar e salvar arquivos MIDI em formato hexadecimal e decimal;
12. Como implementar um Editor de Partituras multicanal/multitimbral com salvamento em formato proprietário e SMF formato 1;

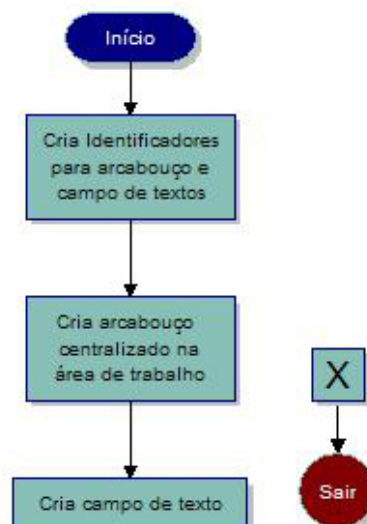
Interface NDI

Uma interface NDI (No Document Interface) é uma interface que possui uma janela de diálogo. Este tipo de interface é a mais simples que se pode implementar. Nela pode-se incluir botões, campos de edição de texto, campos de texto, popups, enfim, os mesmos componentes que a interface SDI permite. A interface NDI não permite a inclusão de menus, toolbars e bitmap de fundo. Por outro lado, possui a vantagem de não se precisar pensar no tamanho da janela, já que a mesma se adapta aos elementos incluídos nela onde apenas se incluiu um texto na interface, ficando o tamanho da mesma condicionada ao tamanho do texto e formatação das margens da janela.



A seguir é mostrado o fluxograma de montagem da janela NDI.

Interface NDI



O programa de implementação desta interface é mostrado na figura a seguir

```

1 module janelaNDI
2 import StdEnv, StdIO
3
4 Start :: *World -> *World
5 Start world
6 = startIO NDI 0 (initialize) [ProcessClose closeProcess] world
7
8 initialize proc
9 # (ids,proc)=openIds 1 proc
10 # (_,proc) =openDialog "" (janela ids) proc
11 =proc
12
13 janela ids = Dialog "JANELA NDI"
14               (TextControl ("Exemplo de janela NDI") [ControlPos (Right,zero)] )
15               [ WindowClose (noLS closeProcess), WindowId wdl]
16 where
17   wdl = ids!!0

```

Para iniciar a interface utiliza-se a técnica de programação com tipos únicos. A função que inicia uma interface é a *startIO*. Esta função possui o seguinte formato:

***startIO* tipo_de_janela estado_local (função) [atributos_do_processo]**

onde:

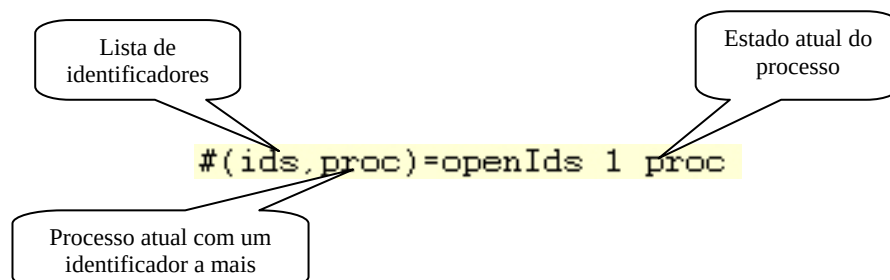
- **tipo_de_janela** = NDI, SDI ou MDI
- **estado_local** = adotou-se, no caso, o inteiro 0. Poderia ser qualquer outro tipo de dado. O mesmo é utilizado quando se deseja passar um dado para os processos da interface de forma global, já que o Clean não possui variáveis globais. Assim, o dado colocado nesta variável poderá ser visto por todos os processos através de seu estado local. Um processo possui um estado local, que é o que se precisa ter antes de iniciar o processo e o processo propriamente dito. Pode-se entender isto exemplificando com produção de refrigerantes. No processo inicial, deve-se partir do ponto que já se tem pelo menos a garrafa onde se colocará o refrigerante. Neste caso, a garrafa será o estado inicial local do processo. Uma fase do processo seria colocar tampa na garrafa do refrigerante, neste caso, o estado local deste processo não pode ser a garrafa vazia, a mesma deve, antes, sofrer o processo de se colocar líquido dentro. Ao se fazer isto, muda-se o estado local antes de passar para o processo de colocar tampa. Assim, podem-se fazer alterações constantes no estado local do processo para que outros processos possam realizar suas tarefas. O processo propriamente dito é controlado pela linguagem, e, assim, não se tem acesso ao mesmo. Nesta fábrica de refrigerante, o processo que não se tem controle poderia ser as máquinas, com

suas esteiras, motores, polias, engrenagens, etc. as quais não são objetos de controle do processo de engarrafamento pelo usuário ou trabalhador da linha de produção. Com algumas permissões e capacitação, poder-se-ia até atuar no processo das máquinas, mas, usualmente, trabalha-se apenas o estado local, deixando a cargo da linguagem, no caso do Clean, cuidar dos detalhes operacionais. Detalhes de como manipular o estado local e o processo podem ser vistos no texto *myCleanBook.pdf* e no manual e help da linguagem: *tutorial.pdf* no CD anexo.

- **função** = função que abre um processo para a criação da interface, no caso *initialize*.

```
8 initialize proc
9 # (ids,proc)=openIds 1 proc
10 # (_,proc) =openDialog "" (janela ids) proc
11 =proc
12
```

Esta função agora é parte do mundo aberto na etapa inicial da construção da interface. Ela inicia o processo de criação da janela. Na mesma é requisitado ao sistema operacional um número determinado de identificadores. Estes identificadores servem para permitir que se acessem os dispositivos da interface, como, por exemplo, os campos de edição de texto. Imagine terem-se dois ou mais campos de edição de texto e se deseja escrever algo em um deles. Nesta hora, deve-se indicar ao programa em qual campo escrever. Assim, ao se criar os dispositivos pode-se destinar um identificador para cada um, conforme foi mostrado nos atributos da janela, onde se nomeou um identificador **wd1** para a janela criada. A função que requisita do sistema operacional identificadores é a **openIds**. Ao utilizar esta função, a mesma cria uma lista de identificadores, conforme valor solicitado a ela. No exemplo apenas se solicitou um identificador. Neste caso, poder-se-ia ter utilizado a função **openId** a qual solicita apenas um identificador. Esta função utiliza operação de IO no sistema operacional, devendo modificar o estado do processo. Ela recebe o processo e devolve o mesmo modificado com um identificador a mais.



A próxima etapa do processo é criar a janela de diálogo. A função que cria uma janela de diálogo é a ***openDialog***. O formato da mesma é:

openDialog estado_local (função_janela)

No caso do exemplo, não se utilizou estado local, no lugar, colocou-se qualquer tipo de dado, no caso uma lista vazia “ ”. Esta função pega o processo, agora com um identificador e devolve um novo processo com uma solicitação de abertura de janela de diálogo.

`#(_,proc) =openDialog "" (janela ids) proc`

A função janela é chamada com a lista de identificadores como parâmetro.

```
janela ids = Dialog "INTERFACE NDI"
  {TextControl ("Exemplo de Interface NDI") [ControlPos (Right,zero)] }
where
  wdl = ids!!0
```

Esta função chama a função ***Dialog*** que abre a janela diálogo propriamente dita. O formato desta função é:

Dialog “texto_do_arcabouço” (componentes) [atributos_da_janela]

- o **componentes** = são os elementos, os dispositivos da interface, tais como: texto, botões, etc. Os mesmos são inseridos na forma de controles. No caso desta interface, apenas se incluiu o controle de texto.

TextControl “Exemplo de janela NDI” [ControlPos (Right,zero)]

O que este controle faz, é colocar um texto na janela. Como se tem apenas este dispositivo na interface vê-se o mesmo centralizado na janela, mas, caso se tivesse outros dispositivos e a janela fosse maior, este texto apareceria à direita da interface. Isto se daria devido à colocação do atributo ***ControlPos(Right,zero)*** neste controle.

- o **atributos_da_janela** = Nesta lista de atributos pode-se colocar os atributos da janela criada, tais como: margens direita, esquerda, superior e inferior, posicionamento da janela na área de trabalho, identificador (para o caso de se ter mais de uma janela), tamanho da janela, e, nas interfaces SDI e MDI, controle da posição do bitmap na janela, variável de controle de mouse e outros mais. No caso, colocou-se os atributos

para fechar a janela⁷⁰ (**WindowClose**) e o atributo para atribuir um identificador para a janela(**WindowId**).

```
[ WindowClose (noLS closeProcess), WindowId wd1]
```

O atributo **WindowClose** chama a função **closeProcess** da biblioteca **StdIO** para habilitar o fechamento da janela (o X do lado superior direito do arcabouço). A chamada desta função está precedida de **noLS** (*no local state*), devido à mesma não necessitar de estado local.

O atributo **WindowId** nomeia um identificador para a janela, no caso **wd1**. O **wd1** é um adoçante sintático correspondente ao primeiro identificador da lista **ids** de identificadores (o único na realidade). O mesmo é nomeado localmente, após a palavra reservada **where**

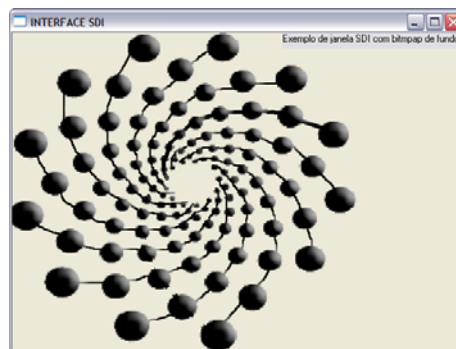
```
where
  wd1 = ids!!0
```

onde, recordando, os símbolos **!!** juntos constituem o operador de pegar um elemento da lista, assim, **ids!!0** pega o primeiro elemento (índice 0) da lista **ids**.

- **atributos_do_processo** = nesta lista pode-se colocar os atributos do processo. No caso deste exemplo, apenas se colocou o atributo de controle de fechamento do processo [**ProcessClose closeProcess**].

Interface SDI

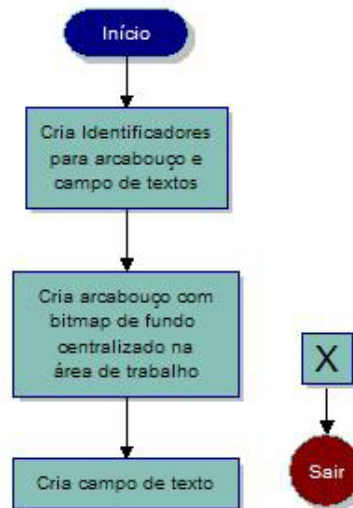
A interface SDI já é uma interface mais elaborada, permitindo-se incluir bitmap de fundo, menus, barra de ferramentas e outros recursos adicionais. A figura abaixo, mostra uma interface SDI com um bitmap de fundo e um texto no início e à direita da interface.



⁷⁰ Caso se tivesse mais de um processo rodando, com várias janelas, com tal atributo pode-se fechar apenas a janela desejada. No caso da função **ProcessClose**, do atributo do processo, fecha-se todos os processos ao se fechar um deles. Assim, caso não seja isto o desejado, elimina-se do processo este atributo.

O Fluxograma de implementação desta interface é mostrado a seguir:

Interface SDI



O programa completo desta interface, conforme fluxograma, é mostrado na figura a seguir:

```

1 module janelaSDICenter
2 import StdEnv, StdIO
3
4 Start :: *World -> *World
5 Start world
6 = startIO SDI 0 (initialize) [ProcessClose closeProcess] world
7
8 initialize proc
9 # (maybeBitmap, proc) = openBitmap "bitmapFundo.bmp" proc
10 | isNothing maybeBitmap = proc
11 | otherwise
12   # bitmap = fromJust maybeBitmap
13   # bitmapsize = getBitmapSize bitmap
14   # (erro, proc) = openWindow Void (janela bitmap bitmapsize) proc
15   | erro <> NoError = proc
16   | otherwise = proc
17
18 janela bitmap bitmapsize
19 = Window "INTERFACE SDI"
20
21   (TextControl ("Exemplo de janela SDI com bitmap de fundo!") [ControlPos (Right, zero)] )
22
23   [WindowViewSize bitmapsize,
24     WindowLook True (\_ _->drawAt zero bitmap),
25     WindowPos (LeftTop, OffsetVector {vx=(maxFixedWindowSize.w-(bitmapsize.w))/2,
26                                       vy=(maxFixedWindowSize.h-(bitmapsize.h))/2}
27   )
28 ]
  
```

Cria-se uma SDI

Abre-se um bitmap (*.bmp) e lê-se o tamanho do mesmo

Chamada de abertura de Window em vez de diálogo

Função que cria uma janela

Observe que pouco difere a janela SDI de uma NDI inicialmente.

1. Inicia-se o processo de criação da janela;

```
startIO SDI 0 (initialize) [ProcessClose closeProcess] world
```

No lugar de NDI, coloca-se SDI

2. Abre-se um bitmap com respectivo tamanho

```
initialize proc
# (maybeBitmap,proc) = openBitmap "bitmapFundo.bmp" proc
| isNothing maybeBitmap = proc
| otherwise
  # bitmap = fromJust maybeBitmap
  # bitmapsize = getBitmapSize bitmap
```

Onde:

- **openBitmap** – abre um bitmap dado um endereço. Quando se coloca o bitmap no mesmo diretório do arquivo executável, não se precisa fornecer o endereço. Caso o bitmap seja colocado em outro diretório, deve-se indicar o caminho completo colocando-se duas barras invertidas na separação de diretório em vez de uma. Ex. C:\\aplicativo\\fundo.bmp”.
- **getBitmapSize** = pega o tamanho do bitmap.

3. Chama a função de criação da janela

```
# (erro, proc) = openWindow Void (janela bitmap bitmapsize) proc
| erro <> NoError = proc
```

A função **openWindow** chama a função de criação da janela. Seu formato é:

openWindow estado_local (função_janela)

Novamente esta função é semelhante à de criar uma NDI, com a diferença que se está chamando a função de criação da janela com dois novos parâmetros: o bitmap a ser colocado de fundo e seu tamanho (para ajustar a janela no tamanho do mesmo): (**janela bitmap bitmapsize**). Como estado local, utilizou-se uma palavra reservada **Void** para indicar que não se vai definir o estado local.

4. Criação da janela

```
janela bitmap bitmapsize
= Window "INTERFACE SDI"
  (TextControl ("Exemplo de janela SDI com bitpmap de fundo!") [ControlPos (Right,zero)] )
  [WindowViewSize bitmapsize,
   WindowLook True (\_ _->drawAt zero bitmap),
   WindowPos (LeftTop,
    OffsetVector {
      vx=(maxFixedWindowSize.w-(bitmapsize.w) )/2,
      vy=(maxFixedWindowSize.h- (bitmapsize.h))/2
    }) ]
```

A função de criação de uma janela SDI é **Window** em vez de **Dialog** da interface NDI. Logo após, também se coloca o texto que vai no arcabouço da interface. O formato é o mesmo, ou seja, tem um conjunto de componentes e uma lista de atributos da janela. (agora com mais atributos).

Window “texto_do_arcabouço” (componentes) [atributos_da_janela]

componentes = O mesmo da interface NDI.

atributos_da_janela = Nesta lista de atributos acrescentou-se mais alguns, a saber:

o **WindowViewSize** – tamanho da janela, no caso, o tamanho do bitmap (WindowViewSize bitmapsize)

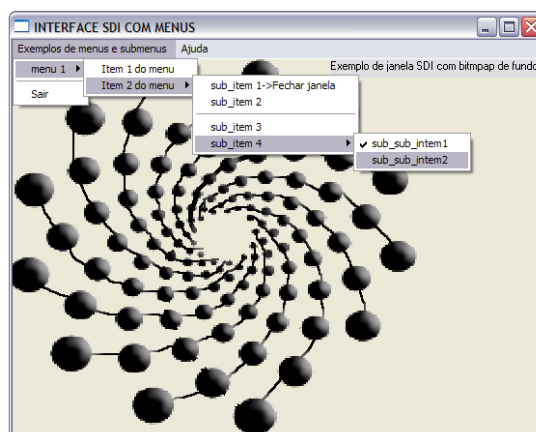
o **WindowLook** – plotagem do bitmap na janela e posição, no caso no início da janela, já que a mesma vai ter o tamanho do bitmap(posição zero) - WindowLook True (_ _->drawAt zero bitmap)

o **WindowPos** – Posição da janela em relação à área de trabalho. Escolheu-se colocá-la no topo à esquerda da área de trabalho (**LeftTop**), com um deslocamento (offset) para centralizá-la. Para o offset, utiliza-se a função que lê o tamanho da área de trabalho (em pixel) a **maxFixedWindowSize.w** e **maxFixedWindowSize.h**. Para deslocar a janela para esta posição, utiliza-se a função **OffsetVector**.

```
WindowPos (LeftTop,
OffsetVector {
vx=(maxFixedWindowSize.w-(bitmapsize.w) )/2,
vy=(maxFixedWindowSize.h- (bitmapsize.h))/2
}) ]
```

Interface SDI com Menus

Um item importante em interfaces é a criação de menus. Os mesmos permitem que a interface fique mais limpa, destinando aos itens e subitens do menu, ações que deveriam ser disparadas por botões, popups e outros dispositivos. A figura abaixo, mostra a mesma interface SDI, agora modificada pelo acréscimo de menus.



A abaixo, mostra o programa completo com a criação dos menus, onde se pode observar que é mantido mesmo bitmap de fundo.

```

1 module menu
2 import StdEnv, StdIO, Notice, Help
3
4 Start :: *World -> *World
5 start world
6 = startIO SDI 0 (initialize) [ProcessClose closeProcess] world
7
8 initialize proc
9 # (maybeBitmap.proc) = openBitmap "bitmapFundo.bmp" proc
10 | isNothing maybeBitmap = proc
11 # bitmap = fromJust maybeBitmap
12 # bitmapsize = getBitmapSize bitmap
13 # (erro, proc) = openWindow Void (janela bitmap bitmapsize) proc
14 | erro <> NoError = proc
15 # (erro, proc) = openMenu Void (Opcoes) proc
16 | erro <> NoError = closeProcess proc
17 # (erro, proc) = openMenu Void (menuHelp) proc
18 | erro <> NoError = closeProcess proc
19 = proc
20
21 where
22 Opcoes = Menu "Exemplos de menus e submenus "
23
24 ( SubMenu "menu 1"
25   ( MenuItem "Item 1 do menu" []
26     :+ SubMenu "Item 2 do menu"
27       ( MenuItem "sub_item 1->Fechar janela" [MenuFunction (noLS sair)]
28         :+ MenuItem "sub_item 2" []
29         :+ MenuSeparator []
30         :+ MenuItem "sub_item 3" []
31         :+ SubMenu "sub_item 4" ( RadioMenu
32           [ ("sub_sub_intem1", Nothing, Nothing, noLS nada),
33             ("sub_sub_intem2", Nothing, Nothing, noLS nada) ]
34           1 []
35         )
36       )
37     )
38   :+ MenuSeparator []
39   :+ MenuItem "Sair" [MenuFunction (noLS sair)]
40 ) []
41
42 menuHelp = Menu "Ajuda"
43 ( MenuItem "Help" [MenuFunction (noLS (showHelp "help.txt")), MenuShortKey 'H']
44   :+ MenuSeparator []
45   :+ MenuItem "&Sobre" [MenuFunction (noLS (showAbout
46     "Menus Clean" "help.txt"))]
47   :+ MenuSeparator []
48   :+ MenuItem "Sair" [MenuFunction (noLS sair)]
49 ) [MenuShortKey 'A']
50
51 sair mundo = openNotice (Notice ["Deseja realmente sair?"]
52   (NoticeButton "&Sim" (noLS closeProcess))
53   (NoticeButton "&N&ao" id)) mundo
54
55 nada proc = proc
56
57 janela bitmap bitmapsize
58 = Window "INTERFACE SDI COM MENUS"
59 (TextControl ("Exemplo de janela SDI com bitmap de fundo!") [ControlPos (Right, zero)] )
60 [ WindowViewSize bitmapsize,
61   WindowLook True (\_ _->drawAt zero bitmap),
62   WindowPos (LeftTop,
63     OffsetVector (
64       vx=(maxFixedWindowSize.v-(bitmapsize.w))/2,
65       vy=(maxFixedWindowSize.h-(bitmapsize.h))/2
66     ) )
67 ]

```

A principal diferença é a criação de dois menus ativados pela função **openMenu**:

- Menu **Opções**:

(erro, proc) = openMenu Void (Opcoes) proc

- Menu **menuHelp**:

(erro, proc) = openMenu Void (menuHelp) proc

Caso um dos menus tenha problemas ao abrir, o processo será fechado:

| erro <> NoError = closeProcess proc

O formato desta função é o mesmo da **openWindow** e **openDialog**. A mesma chama uma função para abrir um menu. As funções, neste exemplo, foram declaradas localmente após a palavra reservada **where**.

Menu Opções

A função que cria um menu é: **Menu**. O formato da mesma é:

Menu “texto do menu” (itens do menu) [atributos]

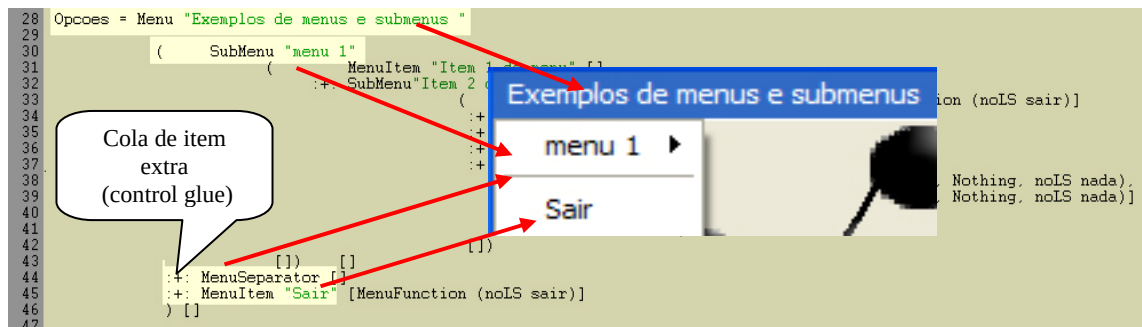
Os itens de um menu podem ser:

- Submenu (função **SubMenu**)
- Separador (função **MenuSeparator**)
- Um item (função **MenuItem**)

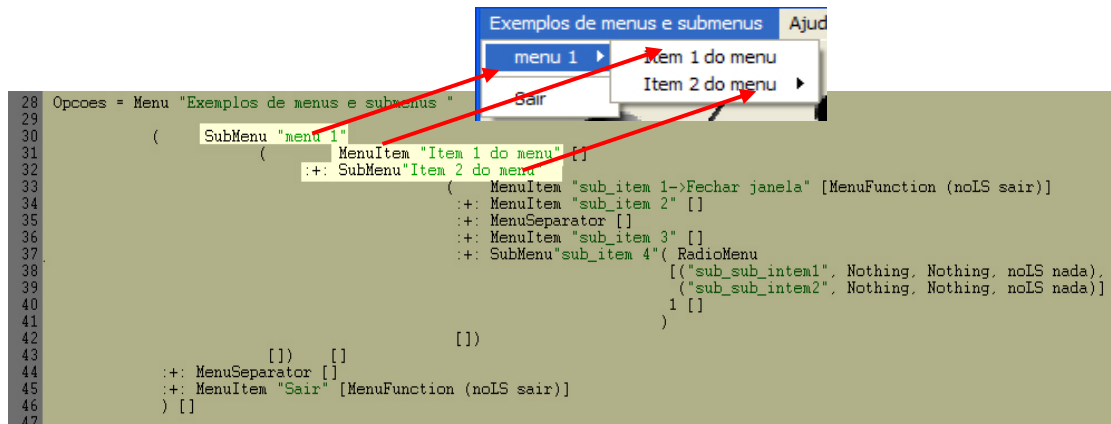
- Um menu tipo radio (função **RadioMenu**)
- Um menu tipo item radio (função **MenuRadioItem**)

No exemplo dado, utiliza-se no menu três tipos, o subMenu, o menu separador e o menu item.

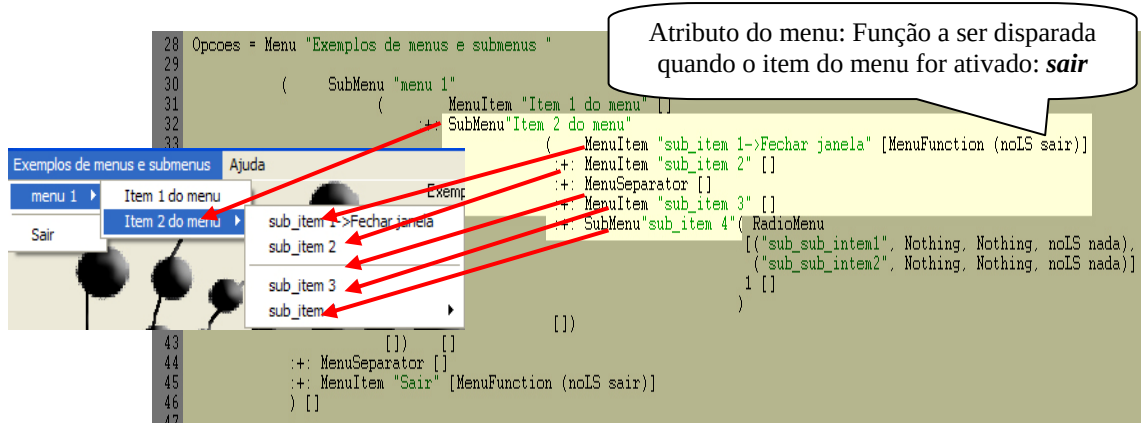
Observe na figura abaixo, o trecho de código correspondente à criação deste menu e de seus itens principais, com a figura do menu na interface depois de aberto. Observe que no item **menu 1** tem-se uma seta, isto se dá porque este menu é um **submenu**, o qual terá outros itens que será mostrado logo a seguir. Um novo item de menu é acrescentado aos existentes através do operador **control glue** (**:+:**).



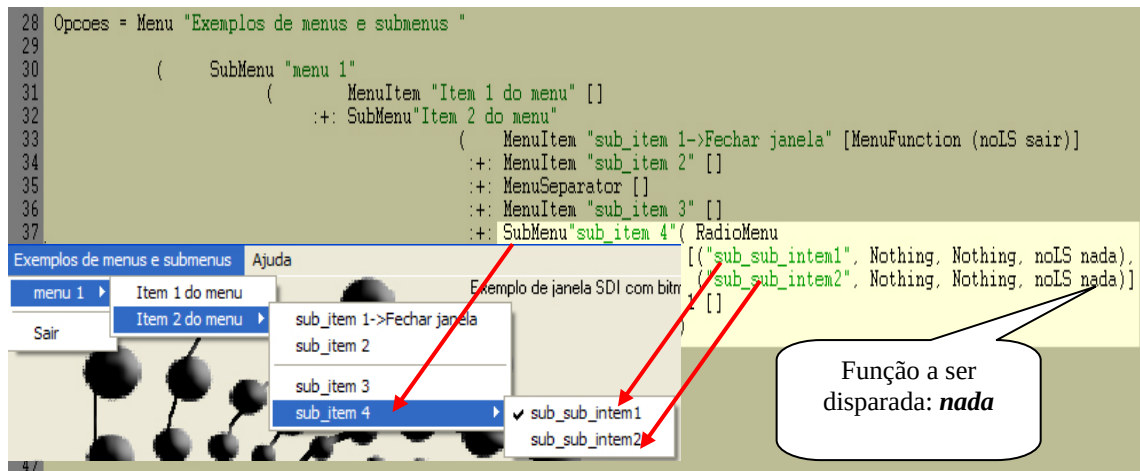
O trecho de programa que cria o submenu e seus itens é mostrado na figura abaixo:



Novamente tem-se um submenu como item do primeiro submenu. O trecho de programa que cria este submenu é mostrado na figura a seguir.



Novamente tem-se um novo subitem de menu disparado, o qual cria um radio menu.



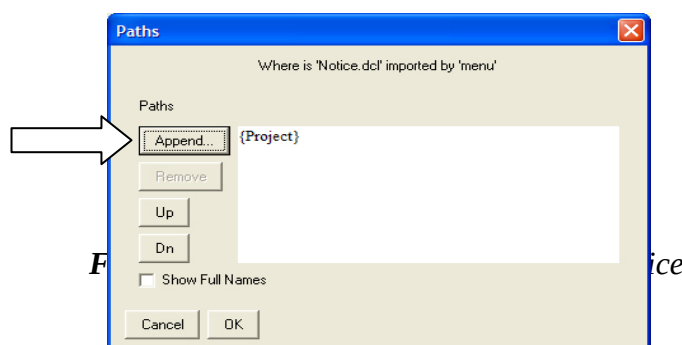
As funções disparadas nos menus foram: **nada** e **sair**. As mesmas foram implementadas como mostrado no trecho de código a seguir:

```

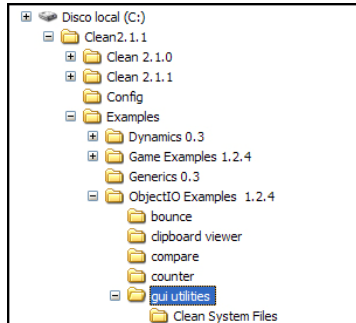
60 sair mundo = openNotice (Notice ["Deseja realmente sair?"]
61                             (NoticeButton "&Sim" (noLS closeProcess))
62                             [NoticeButton "&Não" id]) mundo
63
64
65 nada proc = proc

```

- A função **nada** abre o processo e, como regra, fecha o mesmo sem fazer nada. Assim, esta função faz nada, ou seja, se algum item do menu que possuir esta função for clicado, nada ocorrerá. Esta função também não utiliza o estado local do processo, assim, a chamada da mesma fica: (**noLS nada**).
- A função **sair** dispara uma janela de diálogo do tipo **notice**. Esta função também não utiliza o estado local do processo, assim, a chamada da mesma fica: (**noLS sair**). Esta janela é bastante comum no ambiente Windows, ou seja, é a janela onde você decide se realmente quer continuar a ação solicitada. A função **openNotice** não é uma função da biblioteca StdIO. Quando compilar o programa uma informação de onde se encontra a função será solicitada.



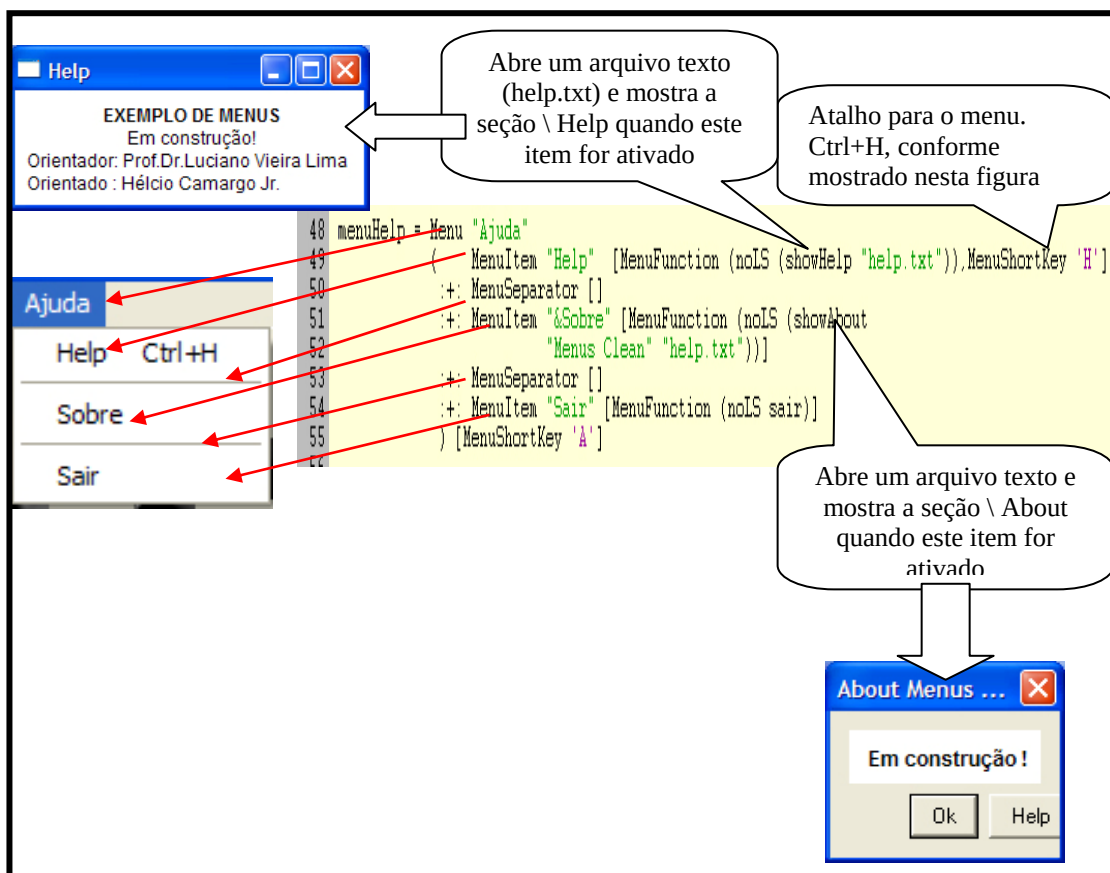
Para localizá-la, clique no botão **Append**. A janela do gerenciador de arquivos será aberta para você indicar a pasta que contém a biblioteca com a função *openNotice*. O caminho da mesma é mostrado na figura a seguir.



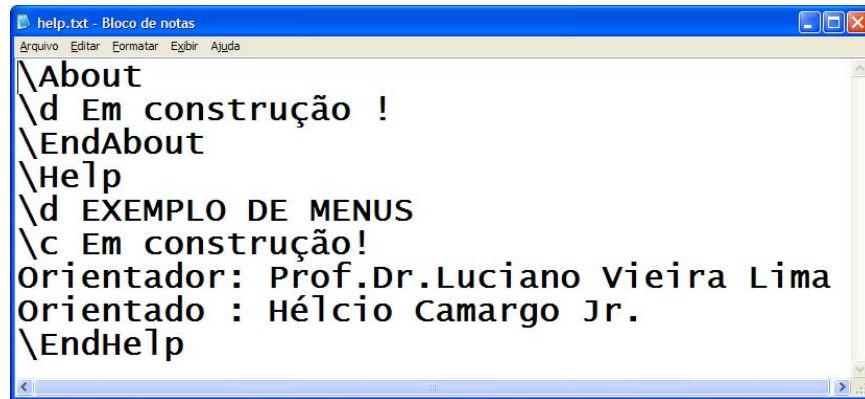
Feito isto, o caminho *C:\Clean2.1.1\Examples\ObjectIO Examples 1.2.4\gui utilities* (no caso de exemplo) é acrescentado no projeto. Para continuar a compilação basta clicar no botão OK.

Menu Help

A figura a seguir mostra o trecho de programa que cria o menu *Help* contendo todos os detalhes da implementação e recursos.



O formato de se criar um *Help*, como o do exemplo, é mostrado na figura a seguir:



Alguns comandos do *Help* e *About*

- A seção *About* inicia por: **\About** e termina com **\EndAbout**
- A seção *Help* inicia por: **\Help** e termina com **\EndHelp**
- **\d** – Centraliza o texto e o coloca em negrito
- **\c** – Centraliza o texto
- Se não colocar **\c** ou **\d** o texto fica à esquerda.

Interface SDI com Botões, Campo de Texto, Texto, Radio Button, Check Button e Pop-Up

Visto como implementar interfaces NDI, SDI, bem como acrescentar menus, um item importante a ser aprendido é como acrescentar controles do tipo botões, textos, etc., em interfaces visuais. Assim, neste item serão apresentados os mais utilizados, bem como os atributos mais comuns de cada um.

A figura a seguir mostra a interface contendo os controles. A figura identifica cada tipo de controle.





Antes de se mostrar a implementação da interface em Clean, é importante entender alguns conceitos de janelas e seus componentes.

Cada componente da janela, inclusive ela, deve possuir identificadores, caso se tenha necessidade de manipulá-los ou relacioná-los com outros componentes. Como exemplos:

- Pode-se querer clicar em um botão e escrever algo em um campo ou em um editor de texto;
- Pode-se desejar ao construir o *layout* da interface, colocar um componente em cima, em baixo ou de um lado de outro componente;
- Pode-se querer ler o conteúdo de um texto ou de um editor de texto ou de um item de um popup, ou mesmo apenas escrever nestes campos.

Assim, nestes casos é importante e obrigatório que tais componentes possuam um identificador.

A função que reserva, que solicita ao sistema operacional, um número determinado de identificadores, em Clean, é a ***openIds***, a qual, ao ser executada, cria uma lista de identificadores que poderão ser utilizados no arcabouço, na janela construída⁷¹. A figura a seguir, mostra o trecho de programa que cria a lista de identificadores.

⁷¹ Inclusive o arcabouço deverá possuir um identificador, já que, para consultar seus componentes, devemos dizer em que janela estão os componentes desejados (pode-se ter mais de uma janela nas interfaces SDI e MDI).

```

3
4 Start :: *World -> *World
5 Start world
6 #(ids,world) = openIds 10 world
7 = startIO SDI 0 (initialize ids) [ProcessClose closeProcess] world
8

```

No caso, foi solicitada a criação de uma lista contendo 10 identificadores. Os mesmos foram criados no início da montagem da interface, antes da chamada para se criar uma SDI, a qual já foi vista anteriormente. Na figura abaixo, *ids* é a lista de identificadores gerada. Assim, ao se criar a janela, passa-se como parâmetro a lista *ids* de identificadores para atribuir aos componentes um id. A listagem 5.12 mostra o trecho de programa que lê o bitmap da janela e que cria a função que dará origem à janela.

```

9 initialize ids proc
10 # (maybeBitmap,proc) = openBitmap "bitmapFundo2.bmp" proc
11 | isNothing maybeBitmap = proc
12 # bitmap = fromJust maybeBitmap
13 # bitmapsize = getBitmapSize bitmap
14 # (erro, proc) = openWindow Void (janela bitmap bitmapsize ids) proc
15 | erro <> NoError = proc
16 =proc
17
18 janela bitmap bitmapsize ids

```

Lista de identificadores

Função que cria a janela

Função que dispara criação da janela

De posse da lista de identificadores pode-se conhecer os conceitos e implementação de cada componente da janela exemplificada na figura anterior. Os componentes são registrados na criação da janela. Observe o formato da função **Window** que cria a janela, partindo da chamada da função **janela** mostrada na figura abaixo.

```

janela bitmap bitmapsize ids = Window "JANELA SDI" ( ) [ ]

```

Título da janela

Inserção dos componentes da janela: botões, texto, etc.

Atributos da janela: tamanho, margens, espaço entre componentes, etc.

Atributos da janela

Nos atributos da janela, normalmente se coloca o identificador da mesma, para que se possa referenciá-la em uma função de leitura de um de seus componentes, por exemplo,

o tamanho da mesma, margem superior e inferior, espaço entre componentes e outros atributos necessários a cada tipo de implementação.

```

5      [WindowViewSize bitmapsize,
6        WindowLook True (\_ _->drawAt zero bitmap),
7        WindowPos (LeftTop, OffsetVector {vx=(maxFixedWindowSize.w-(bitmapsize.w) )/2,
8                                              vy=(maxFixedWindowSize.h- (bitmapsize.h))/2}),
9        WindowHMargin 10 10,
10       WindowVMargin 10 10,
11       WindowItemSpace 5 15,
12       WindowId wd1
13     ]

```

Onde:

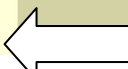
- **WindowViewSize** – define o tamanho da janela, no caso, o tamanho do bitmap de fundo: **WindowViewSize bitmapsize;**
- **WindowLook** – *plota* o bitmap de fundo na janela: **WindowLook True (_ _->drawAt zero bitmap);**
- **WindowPos** – define a posição da janela na área de trabalho - **WindowPos (LeftTop, OffsetVector {vx=(maxFixedWindowSize.w-(bitmapsize.w))/2, vy=(maxFixedWindowSize.h-(bitmapsize.h))/2})** , onde vx define a coordenada x centralizando a janela, ou seja, lê a largura em pixel da área de trabalho (**maxFixedWindowSize.w**) e subtrai da metade da largura do bitmap(**bitmapsize.w**) /2. O mesmo é feito para a altura da janela;
- **WindowHMargin** – define as margens na largura esquerda e direita da janela: **WindowHMargin 10 10 ;**
- **WindowVMargin** – define as margens superior e inferior da janela: **WindowVMargin 10 10;**
- **WindowItemSpace** – define o espaço horizontal e vertical entre componentes da janela: **WindowItemSpace 5 15;**
- **WindowId** – atribui um identificador para a janela: **WindowId wd1.**

Na realidade, o identificador é um dos elementos da lista com os 10 identificadores reservados no início do programa pela função **openIds 10**. O apelido (no caso, **wd1**) é nomeado no programa localmente, após os atributos da janela.

```

65 [WindowViewSize bitmapsiz,
66   WindowLook True (\_ _->drawAt zero bitmap),
67   WindowPos (LeftTop, OffsetVector {vx=(maxFixedWindowSize.w-(bitmapsiz.w) )/2,
68                                     vy=(maxFixedWindowSize.h- (bitmapsiz.h))/2}),
69   WindowHMargin 10 10,
70   WindowVMargin 10 10,
71   WindowItemSpace 5 15,
72   WindowId wd1
73 ]
74 where
75   CTEXT1 =ids!!0
76   TEXTO =ids!!1
77   CTEXT2 =ids!!2
78   wd1 =ids!!3

```



No caso, **wd1** é o quarto elemento (**ids!!3**) da lista dos 10 identificadores. A palavra reservada **where** inicia a declaração de variáveis e funções locais.

Componentes da Interface

Os componentes de uma interface são adicionados através de uma “cola de controle” (control glue), cujo operador possui o símbolo **:+:** . A seguir são mostrados os componentes da interface exemplo.

Botão

Um dos componentes mais comuns em uma interface é o botão. O mesmo é utilizado para disparar funções ao ser pressionado. Para se criar um botão, utiliza-se uma função de controle da biblioteca de I/O do Clean chamada **ButtonControl**. O Formato desta função é:

ButtonControl nome_do_botão [lista_de_atributos]

Na interface exemplo, tem-se dois botões criados: **Calculadora** e **APAGAR TEXTO**.

As figura abaixo mostram o código utilizado para criar os mesmos nesta interface.



```

28      :+: ButtonControl "Calculadora" [ControlFunction (noLS calc),
29                                         ControlWidth (PixelWidth 150),
30                                         ControlPos (LeftTop,zero)]
31

```



```

40      :+: ButtonControl "APAGAR TEXTO" [ControlFunction (noLS apagar),
41                                           ControlWidth (PixelWidth 150),
42                                           ControlPos (RightTop,zero)]

```

Na lista de atributos dos botões, temos 3 atributos em cada uma, definidos pelas funções:

1. **ControlFunction**

2. *ControlWidth*

3. *ControlPos*

1- *ControlFunction*

O atributo ***ControlFunction*** permite associar ao botão uma função do processo a ser disparada quando o botão for pressionado.

- No caso do botão “Calculadora” tem-se:

ControlFunction (noLS calc) onde *calc* é a função do processo que será ativada, a qual disparará um aplicativo de calculadora, conforme será visto posteriormente.

- No caso do botão “APAGAR TEXTO” tem-se:

ControlFunction (noLS apaga) onde *apaga* é a função do processo que será ativada para apagar os campos de texto da interface.

2-*ControlWidth*

Este atributo é responsável por determinar, em *pixels*, a largura do botão. Isto é útil para padronizar os tamanhos dos elementos da interface. Caso não se utilize este atributo, o botão terá o tamanho do texto associado ao mesmo.

-Para os dois botões, adotou-se a largura de 150 *pixels*. Assim, a implementação deste atributo, fica:

ControlWidth (PixelWidth 150)

3-*ControlPos*

Este atributo permite escolher onde colocar o botão ou outro componente da interface. Existem várias possibilidades de se posicionar um componente. As mais usuais são:

3.1- Posição relativa ao canto:

- *ControlPos (LeftTop,zero)*
- *ControlPos (RightTop,zero)*
- *ControlPos (LeftBottom,zero)*
- *ControlPos (RightBottom,zero)*

3.2 Relativa à próxima linha (nesta modalidade, este atributo faz saltar uma linha na interface):

- *ControlPos (Left,zero)*

- ControlPos (Center,zero)
- ControlPos (Right,zero)

3.3 Relativa a outro componente

- ControlPos (LeftOf Id,zero)
- ControlPos (RightTo Id,zero)
- ControlPos (Above Id,zero)
- ControlPos (Below Id,zero)

Onde: Id é o identificador do botão. Para colocar um identificador no botão, deve-se colocar o atributo: **ControlId Id**

3.4 Relativa a um componente colocado anterior a ele

- Control Pos (LeftOfPrev,zero)
- Control Pos (RightToPrev,zero)
- Control Pos (AbovePrev,zero)
- Control Pos (BelowPrev,zero)

3.5 Ajustando a posição relativa (*offset*) do componente

Pode-se deslocar o posicionamento de um componente colocando um *offset* no mesmo em vez de **zero**. Isto é feito através da função **OffsetVector**.

Pode-se deslocar o componente horizontalmente ou verticalmente, ou seja:

- Quando se utiliza o posicionamento *Left* ou *Right*, apenas pode-se deslocar o componente na vertical;
- Quando se utiliza o posicionamento *Center*, pode-se deslocar o componente tanto na vertical quanto na horizontal.

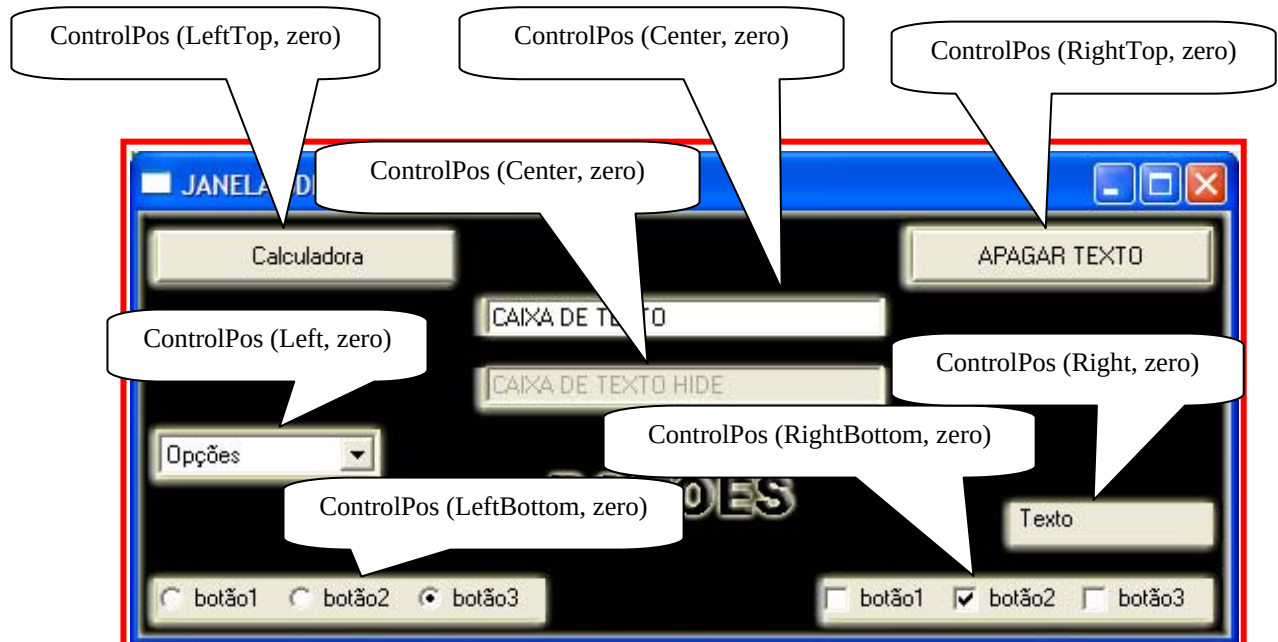
O formato no controle, para **Center**, fica:

ControlPos (Center, OffsetVector {vx = deslocamentoX, vy= deslocamentoY })

Onde o deslocamento **vx** é o deslocamento horizontal e **vy** é o deslocamento vertical (em *pixel*).

Nos dois botões da interface foram utilizados o posicionamento relativo aos cantos: **LeftTop** e **RightTop**.

A figura abaixo, mostra o atributo de posicionamento utilizado em cada componente da janela.



Campo de Edição de Texto

Outro componente muito utilizado em interfaces é o campo de edição de texto. Para se criar um campo de edição de texto, utiliza-se uma função de controle da biblioteca de I/O do Clean chamada **EditControl**. O Formato desta função é:

EditControl texto_inicial (*PixelWith* largura_pixel) número_de_linhas [lista_de_atributos]

Como a função de um campo é ser lido ou permitir a ação de escrita no mesmo, deve-se associar ao mesmo um identificador. Isto é feito através do atributo **ControlId**. No exemplo da figura abaixo, o nome do identificador é **CTEXT1**.



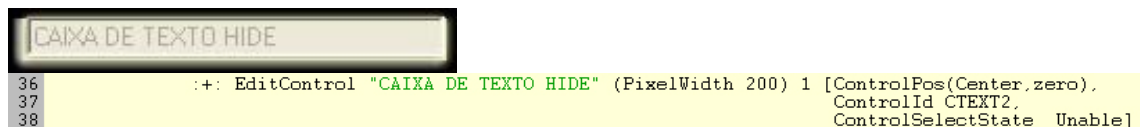
```
32      :+: EditControl "CAIXA DE TEXTO" (PixelWidth 200) 1 [ControlId CTEXT1,
33                                     ControlPos(Center,zero),
34                                     ControlTip "CAMPO EDITÁVEL!"]
```

Observe que no lado inferior direito do campo de edição, aparece uma mensagem CAMPO EDITÁVEL. Esta mensagem é denominada de **Tip**. Esta mensagem aparece quando se coloca o ponteiro do mouse em cima do componente, no caso, do editor de texto. O atributo que implementa o **Tip** é o **ControlTip**. O formato do mesmo é:

ControlTip dica_ou_mensagem

O posicionamento escolhido para este campo de texto foi o centralizado sem *offset*.

A figura abaixo, mostra o outro campo de texto criado. O mesmo diferencia do primeiro por ser apenas de leitura. Utilizou-se um atributo que desabilita o editor para escrita pelo usuário. Este atributo é **ControlSelectState Unable**.



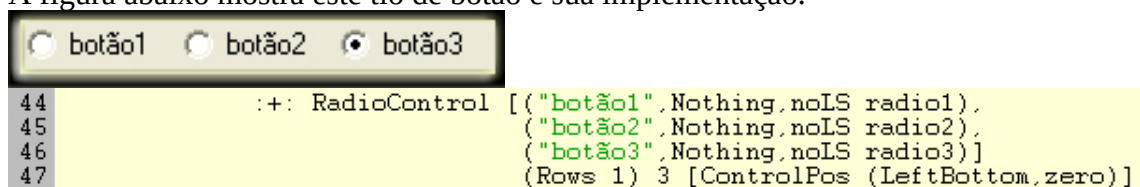
Botão tipo Radio

Este botão permite se associar uma função diferente para cada botão radio marcado.

Neste tipo de botão, apenas uma opção fica marcada por vez. O formato deste botão é:

Radio [(nomeBotao,*Nothing*, função do botão)]
 (Rows número_de_linhas número_botao_ativado)
 [lista_de_atributos]

A figura abaixo mostra este tipo de botão e sua implementação.

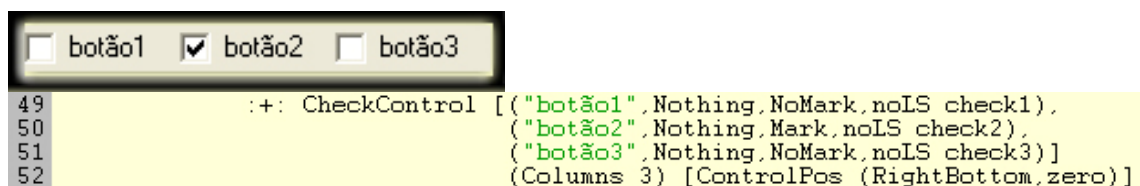


No caso de não se querer disparar nenhuma função com o botão, basta trocar a função pela palavra reservada **Id**. Exemplo: [(**"botão1"**,*Nothing*, **Id**),....]

A posição escolhida para este controle foi no canto inferior esquerdo (**LeftBottom**).

Botão tipo Check

Este botão é parecido como o tipo **Radio**, com a diferença que se pode ter mais de uma opção marcada ao mesmo tempo.



O formato deste atributo é:

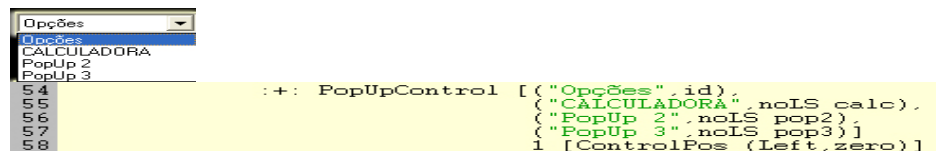
CheckControl [(nomeBotao,*Nothing*, marca_ou_não, função do botão)]
 (**Rows** número_de_linhas número_botao_ativado)
 (**Columnns** número_de_colunas)
 [lista_de_atributos]

O programador pode escolher quais caixas de marcação ficarão marcadas como padrão inicial. No caso, **NoMark** implica na caixinha ficar desmarcada e **Mark** implica na caixinha ficar desmarcada.

A posição escolhida para este controle foi no canto inferior direito (**RightBotom**) sem *offset*.

Menu Popup

Com este tipo de menu pode-se elencar vários itens para que o usuário escolha um. Ao fazer a escolha, uma função associada ao item é disparada.



O formato do menu popup é:

PopUpControl [(nomeBotão, função do botão)] linha_inicial [lista_de_atributos]

Caso não se deseje disparar uma função, pode-se utilizar **id**, como é o caso da primeira linha do menu popup: (“Opções”, **id**)

A posição escolhida para o popup foi relativa à próxima linha, à esquerda, ou seja: salta-se uma linha e coloca o menu popup na lado esquerdo da mesma sem *offset*.

Texto (campo de texto)

Este controle acrescenta um texto na interface, o qual pode ser modificado e lido, também, caso tenha um identificador. Na interface, dois campos de texto foram colocados, apesar de apenas um ser visível para o usuário. Um campo de texto invisível serve para se colocar espaço entre elementos da interface, para se saltar linha e para ser utilizado para guardar parâmetros de controle da interface, como se fosse uma variável global (já que podem ser lidos e modificados por qualquer elemento ou função da interface).

O formato deste controle é:

TextControl texto_desejado [lista_de_atributos].

Campo de texto visível ao usuário



```
60      :+: TextControl " Texto " [ControlPos(Right,zero),
61                               ControlWidth (PixelFormat 100),
62                               ControlId TEXTO]
```

Este campo possui um identificador, mostrado anteriormente, nomeado de **TEXTO**.

Campo de texto invisível ao usuário

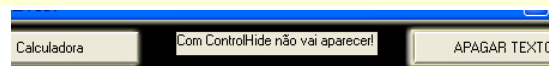
A figura a seguir mostra o código contendo o controle de texto com o atributo **ControlHide**. Este atributo é que faz com que o texto (ou outro controle qualquer) não fique visível ao usuário.

```
22      (      TextControl "Com ControlHide não vai aparecer!" [ControlPos(Center,zero),
23                                                       ControlHide]
```



Caso se eliminasse o atributo **ControlHide**, o texto apareceria na janela.

```
      (      TextControl "Com ControlHide não vai aparecer!" [ControlPos(Center,zero)]
```



Escrevendo em um campo de texto (editável ou não)

Para analisar como escrever nos campos de texto, observe a figura abaixo a qual mostra a função acionada pelo botão “APAGAR TEXTO”.

```
40      :+: ButtonControl "APAGAR TEXTO" [ControlFunction (noLS apagar),
41                                         ControlWidth (PixelFormat 150),
42                                         ControlPos (RightTop,zero)]
```

```
80      apagar proc
81      # proc= appPIO (setControlText CTEXT1 "")proc
82      # proc= appPIO (setControlText CTEXT2 "")proc
83      # proc= appPIO (setControlText TEXTO "")proc
84      =proc
```

Cada linha da função apagar se repete com a ação de escrever em um campo de texto. No caso, *apagar* escreve em três campos de texto:

- **CTEXT1** – campo do editor 1;
- **CTEXT2** – campo do editor 2;
- **TEXTO** – campo de texto.

A função de escrever possui o seguinte formato:

setControlText identificador_do_campo texto

O texto colocado em cada um dos campos é a string vazia, o que equivale a mandar apagar os campos. Para se aplicar esta função nos campos, utiliza-se a função ***appPIO***.

Lendo um campo de texto

Para se ler um campo através de uma função do processo, deve-se, primeiro, ler o arcabouço da janela para se conhecer os nomes dos identificadores e dos elementos da janela. Para tanto, a leitura é feita através da seguinte linha de programa:

(Just dial, proc) = accPIO(getWindow wd1) proc

Onde **wd1** é o identificador da janela.

Para se ler um dado do campo de texto 1, deve-se incluir na função a seguinte linha de programa:

(se, Just ctexto1) = getControlText CTEXT1 dial

Feito isto, o conteúdo do campo de texto 1 será armazenado na variável **ctexto1**, arbitrariamente nomeada no exemplo.

Exemplo: Implementando uma função que lê o conteúdo do campo de edição de texto 1 e escreve no campo de edição de texto 2. A figura abaixo mostra como ficaria a implementação desta função, nomeada de ***leCampo1Escreve2*** para realizar esta ação.

```
2 leCampo1Escreve2 proc
3 # (Just dial, proc) = accPIO(getWindow wd1) proc
4 # (se, Just ctexto1)= getControlText CTEXT1 dial
5 # proc= appPIO (setControlText CTEXT2 ctexto1)proc
6 =proc
```

Estes são os principais conceitos necessários para se implementar uma janela SDI com os componentes mais utilizados normalmente neste tipo de interface.

Registro Local em interface SDI

Um registro permite que se armazenem diferentes tipos de dados no mesmo, além de permitir ler e alterar valores do mesmo, acessando-se diretamente o campo que se quer editar. Como o Clean não possui variáveis globais, um registro possibilita a passagem de parâmetros de um processo para o outro globalmente, tutelados, por exemplo, por um

determinado processo, evitando os efeitos colaterais da perda da transparência referencial.

Para se criar um registro deve-se, antes, criar um tipo algébrico adequado que permita representar os dados do registro que se deseja manipular. O formato de um registro é:

No caso, criou-se o seguinte tipo de registro

:: Nome = {nomeCampo1 :: tipo_campo1, nomeCampo2 :: tipo_campo2, ..}

Onde:

- **Nome** – qualquer nome começando com uma letra maiúscula;
- **nomeCampo** = qualquer nome começando com uma letra minúscula;
- **tipo_campo** = um tipo primitivo da linguagem ou uma associação dos mesmos;
- Um registro possui um limite de 30 campos.

O registro criado possui o nome **Reg**, possuindo dois campos:

- **midiTexto** → tipo string
- **listaMIDItexto** → tipo lista de lista de lista de string

```
5 :: Reg = {midiTexto :: {#Char}, listaMIDItexto :: [[[{#Char}]]]}
```

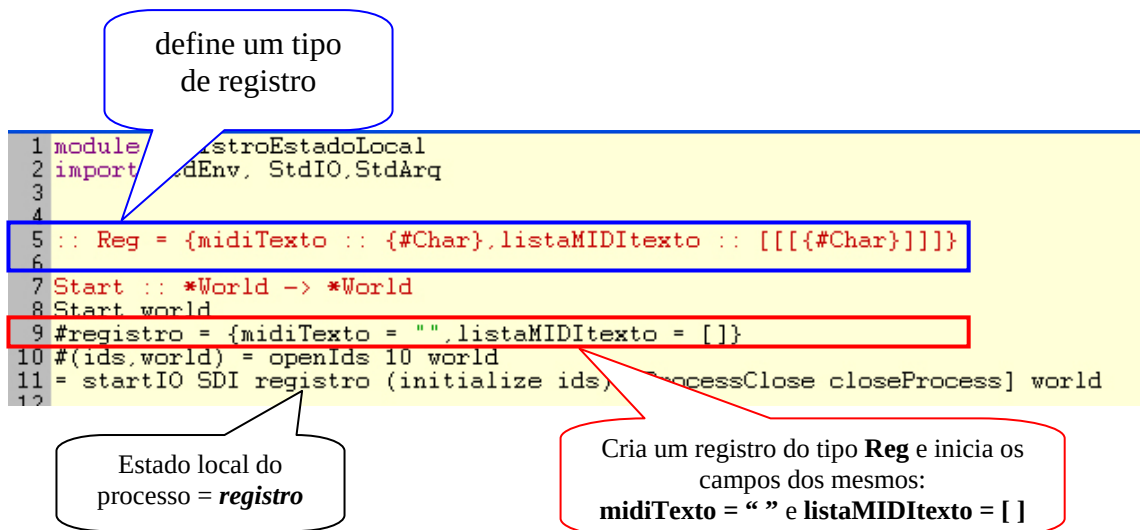
Como aplicação e exemplo, será mostrado a implementação de uma interface SDI que:

1. Faz a leitura de um texto musical, o qual foi mostrado no item 3.4.1 do capítulo 3.
2. Feita a leitura, o conteúdo do arquivo é armazenado em um registro e colocado no estado local do processo da interface, podendo ser posteriormente lido. Esta leitura é feita quando um determinado botão é apertado;
3. Ao mesmo tempo em que a leitura do registro é feita, através do estado local do processo, é realizada a transposição musical do texto mediante o número de semitons lido no menu popup da interface;
4. Feita a leitura, o conteúdo do arquivo é mostrado em um campo de edição de texto, e a música transposta é armazenada no registro do estado local do processo em forma de uma lista de compassos e de notas_figura;
5. Quando se pressiona outro botão, a lista da música transposta é convertida para formato texto, na sintaxe do arquivo lido e mostrada em outro campo de texto;
6. Nesta mesma função, o novo arquivo texto transposto é colocado no registro do estado local do processo no lugar do texto original;
7. O processo pode ser repetido. Como não se tratou o limite do range de notas neste exemplo, não se deve realizar transposições que excedam o limite da oitava 0 e da oitava 9.

O aplicativo:

Quando o programa é executado, as seguintes ações ocorrem e/ou podem ser ativadas:

1. Cria-se um tipo de registro (**Reg**);
2. Cria-se um registro no processo com seus parâmetros iniciais;
3. Cria-se uma lista de 10 identificadores;
4. A Interface SDI é iniciada;

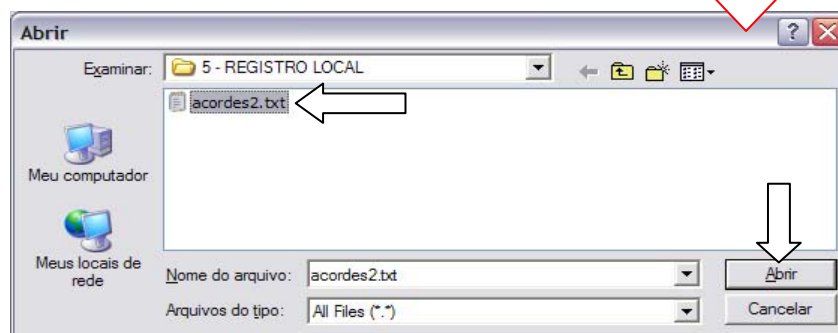


5. O arquivo Texto é aberto. A figura abaixo mostra o trecho de programa que abre um arquivo texto para edição e a tela inicial de execução deste aplicativo⁷².

```

13 initialize ids proc
14 # (maybeBitmap, proc) = openBitmap "fundo.bmp" proc
15 | isNothing maybeBitmap = proc
16 # bitmap = fromJust maybeBitmap
17 # bitmapsize = getBitmapSize bitmap
18 # (erro, proc) = openWindow Void (janela ids bitmap bitmapsize) proc
19 | erro <> NoError = proc
20 #(conteudo, endereco, proc) = AbrirArquivoTextoExlproc proc
21 # texto = StringTranslate is10 conteudo
22 = {ls = {proc.ls & midiTexto = texto}, io = proc.io }
23

```



⁷² Como arquivo texto musical, foi utilizado o arquivo "acordes2.txt", o qual também está no CD em anexo.

Neste trecho de programa em destaque, utiliza-se uma função da biblioteca StdArq para leitura de arquivo, já mostrada no 3.4.3.1 do Capítulo 3. A novidade é a linha seguinte (21) que:

- Aplica a função (***StringTranslate is10***) no conteúdo aberto. Esta função, também da biblioteca StdArq;
- A função ***StringTranslate*** aplica a função ***is10*** na string lida.
- A função que ***StringTranslate*** aplica na string é uma função que modifica um determinado caractere na string;
- No caso, deseja-se trocar o caractere 10 pelo caractere de saltar linha ‘\n’ (***toChar13,toChar 10***);
- Ao testar um caractere da string o mesmo for o caractere 10, ***StringTranslate*** troca este caractere pelo caractere ‘\n’ que o Clean entende. Se o caractere for o 13, é eliminado⁷³, e, se não for o 10, nada se faz.

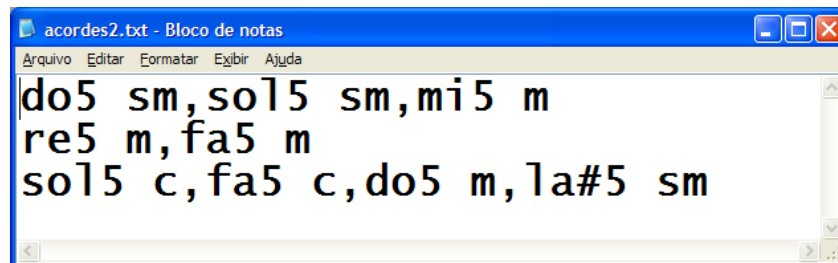
A função ***is10***, também da biblioteca StdArq, é a seguinte:

```
502 is10 :: Char -> {#Char}
503 is10 x
504 | x == toChar 10 = {toChar 13,toChar 10}
505 | x == toChar 13 = ""
506 | otherwise = toString x
```

A função da biblioteca StdArq, ***StringTranslate*** é mostrada a seguir:

```
227 StringTranslate :: (Char->{#Char}) {#Char} -> {#Char}
228 StringTranslate f str = makestr f str 0 ""
229 where
230   makestr f str n ac
231   | n == size str = ac
232   | otherwise = makestr f str (n+1) (ac +++ (f str.[n]))
```

O conteúdo do arquivo Texto utilizado, o acordes2.txt, é mostrado na *figura* a seguir.



6. Selecionado o arquivo, o mesmo é colocado em um registro e no estado local do processo, e, logo após, a interface é aberta com os dois botões, um popup e dois campos de texto.

⁷³ Elimina-se o caractere 13 para evitar duplicá-lo, caso no arquivo lido se tenha o \n no formato {toChar 143, toChar 10}



7. O texto lido é armazenado no estado local do processo na forma de um campo do registro criado. A linha de programa que permite e faz a atualização do registro no estado local do processo é mostrada na figura a seguir.

```

13 initialize ids proc
14 # (maybeBitmap,proc) = openBitmap "fundo.bmp" proc
15 | isNothing maybeBitmap = proc
16 # bitmap = fromJust maybeBitmap
17 # bitmapsize = getBitmapSize bitmap
18 # (erro, proc) = openWindow Void (janela ids bitmap bitmapsize ) proc
19 | erro <> NoError = proc
20 #(conteudo,endereco,proc) = AbrirArquivoTextoExplproc proc
21 # texto = StringTranslate is10 conteudo
22 = {ls = {proc.ls& midiTexto = texto}, io = proc.io }
23

```

Um processo, como já dito anteriormente, possui um estado local e o processo propriamente dito. Para se alterar um campo do registro dentro do estado local do processo o operador é **&**. Assim, deve-se pegar do processo seu estado local e atualizar o campo desejado, no caso, o campo ***midiTexto*** no qual vai ser guardado o conteúdo do arquivo lido.

A linha de programa que altera este campo é:

```

22 = {ls = {proc.ls& midiTexto = texto}, io = proc.io }

```

8. Ao se clicar no botão **Ler Música Texto**, o campo **midiTexto** do estado local do processo é lido pela função **ler**, colocando o texto lido no primeiro campo de edição, conforme figura, a seguir.



A função **ler** é mostrada no trecho de código da figura a seguir,

```

49 ler_proc =:{ls={midiTexto,listaMIDItexto}}
50 # (Just dial, proc) = accPIO(getWindow wd1) proc
51 # (se, Just semitonsTrans)= getControlText t3 dial
52 # proc = appPIO (setControlText t1 midiTexto)proc
53 # transposta = convTextoListMIDI midiTexto (toInt semitonsTrans)
54 = {ls = {proc.ls& listaMIDItexto = transposta}, io = proc.io }
55

```

Onde:

- Os campos do registro do estado local do processo, **midiTexto** e **listaMIDItexto**, são lidos.

```

45 ler_proc =:{ls={midiTexto,listaMIDItexto}}

```

- O popup contendo o número de semitons em que a música deve ser transposta é lido.

```

46 # (Just dial, proc) = accPIO(getWindow wd1) proc
47 # (se, Just semitonsTrans)= getControlText t3 dial

```

A função **getWindow** lê as características de uma determinada janela, no caso, a **wd1**, tal como a lista de identificadores e seus adoçantes sintático.

(Just dial, proc) = accPIO (getWindow wd1) proc

dial passa a guardar os dados da janela **wd1**, como, por exemplo, qual é o valor do identificador **t3** do popup que se deseja ler.

Conhecido o identificador dos dispositivos desejados, no caso, o identificador do popup, pode-se proceder à sua leitura. Esta leitura é feita pela função **getControlText**.

(se, Just semitonsTrans)= getControlText t3 dial

Na variável **semitonsTrans** é guardado o conteúdo do popup lido (o número de semitons)

- Lido o conteúdo, o mesmo é escrito no primeiro campo de edição de texto (identificador **t1**). A função que escreve em um campo de texto é **setControlText**

proc = appPIO (setControlText t1 midiTexto) proc

A figura a seguir mostra a interface com o texto escrito no campo de texto e a configuração do valor de semitons do popup.



- Feita a escrita do texto no campo t1, o mesmo é transposto em forma de lista, conforme função mostrada no item 3.4.1 do capítulo 3, pela função **convTextoListMIDI** da biblioteca StdArq. Como o número de semitons lido no popup está em formato texto, o mesmo é convertido para inteiro através da função **toInt** da biblioteca StdEnv.

```
49 # transposta = convTextoListMIDI midiTexto (toInt semitonsTrans)
```

- O estado local do processo é atualizado, agora com a lista musical transposta colocada no campo **listaMIDItexto**.

```
50 = {ls = {proc.ls& listaMIDItexto = transposta}, io = proc.io }
```

A função **transpoeTexto** é mostrada no trecho de código da figura a seguir,

```
56
57 transpoeTexto proc = {ls={midiTexto, listaMIDItexto}}
58 # textoConvertido= convListMIDItexto listaMIDItexto
59 # proc = appPIO (setControlText t2 textoConvertido)proc
60 = {ls = {proc.ls& midiTexto = textoConvertido}, io = proc.io }
61
```

Onde:

- Na linha 58 utiliza-se a função **convListMIDItexto**, inclusa na biblioteca StdArq, para converter a lista musical em texto.
- Novamente utiliza-se, na linha 59, a função **setControlText** para mostrar o texto da música transposta no campo de texto 2, conforme figura a seguir.



- Atualiza-se novamente o registro do estado local, agora com o texto musical transposto. Ao se clicar novamente no botão **Ler Música Texto**, agora será *plotado* o texto transposto, sendo o popup de semitons novamente lido, repetindo-se o processo.

Fica-se assim, detalhado o processo de criação, leitura e uso de registros em estado local de processos. As demais funções do programa já foram objetos de estudo nos itens anteriores deste anexo.

A figura a seguir mostra o programa completo de transposição utilizando registro e estado local do processo.

```

1 module registroEstadoLocal
2 import StdEnv, StdIO, StdArq
3 :: Reg = {midiTexto :: {#Char}, listaMIDItexto :: [[{#Char}]]}
4
5 Start :: *World -> *World
6 Start world
7 #registro = {midiTexto = "", listaMIDItexto = []}
8 #({ids, world}) = openIds 10 world
9 = startIO SDI registro (initialize ids) [ProcessClose closeProcess] world
10 initialize ids proc
11 # (maybeBitmap, proc) = openBitmap "fundo.bmp" proc
12 | isNothing maybeBitmap = proc
13 # bitmap = fromJust maybeBitmap
14 # bitmapsize = getBitmapSize bitmap
15 # (erro, proc) = openWindow Void (janela ids bitmap bitmapsize) proc
16 | erro <> NoError = proc
17 # (conteudo, endereco, proc) = AbrirArquivoTextoExplproc proc
18 # texto = StringTranslate is10 conteudo
19 = {ls = {proc.ls & midiTexto = texto}, io = proc.io }
20
21
22 janela ids bitmap bitmapsize = Window "EXEMPLO DE USO DE REGISTRO"
23   (ButtonControl "Ler Música Texto" [ControlFunction (noLS ler), ControlPos (Left, zero)])
24   :+ ButtonControl "Música Transposta" [ControlFunction (noLS transpoeTexto)]
25   :+ PopUpControl ([("Semitons", id)] ++ [(toString x, id) \\ x <- [1..12]] ) 1 [ControlId t3]
26   :+ EditControl "" (PixelWidth 390) 5 [ControlId t1, ControlPos (Center, zero)]
27   :+ EditControl "" (PixelWidth 390) 5 [ControlId t2, ControlPos (Center, zero)]
28   )
29   [WindowViewSize bitmapsize,
30   WindowLook True (\_ _->drawAt zero bitmap),
31   WindowPos (LeftTop, OffsetVector {vx=(maxFixedWindowSize.w-(bitmapsize.w))/2,
32   vy=(maxFixedWindowSize.h-(bitmapsize.h))/2}),
33   WindowHMargin 10 10, WindowVMargin 10 10, WindowItemSpace 5 15, WindowId wd1]
34 where
35   t1 = ids!!0
36   t2 = ids!!1
37   t3 = ids!!2
38   wd1 = ids!!3
39   ler proc = {ls={midiTexto, listaMIDItexto}}
40   # (Just dial, proc) = accPIO(getWindow wd1) proc
41   # (se, Just semitonsTrans) = getControlText t3 dial
42   # proc = appPIO (setControlText t1 midiTexto) proc
43   # transposta = convTextoListMIDI midiTexto (toInt semitonsTrans)
44   = {ls = {proc.ls & listaMIDItexto = transposta}, io = proc.io }
45
46   transpoeTexto proc = {ls={midiTexto, listaMIDItexto}}
47   # textoConvertido = convListMIDItexto listaMIDItexto
48   # proc = appPIO (setControlText t2 textoConvertido) proc
49   = {ls = {proc.ls & midiTexto = textoConvertido}, io = proc.io }

```

Exemplo de ToolBar

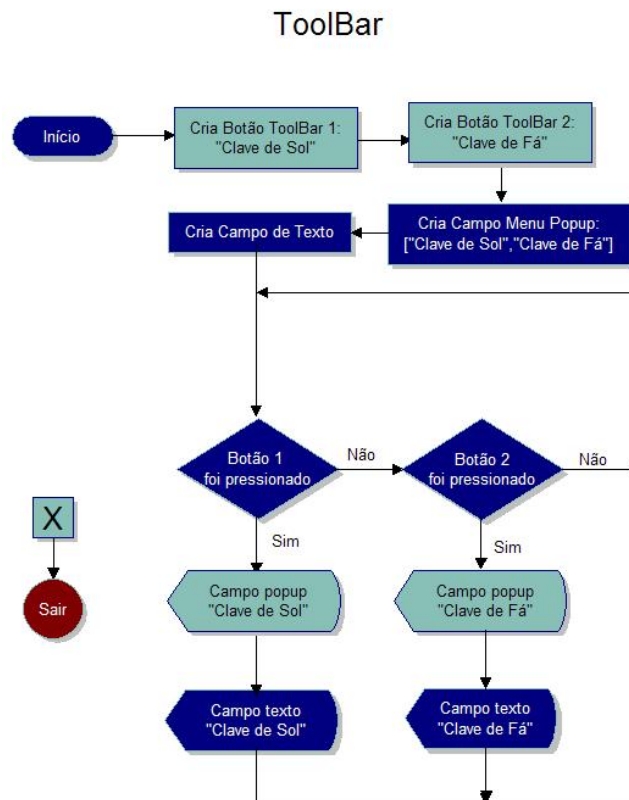
Neste exemplo é mostrado como criar uma barra de ferramentas “Toolbar”. Os botões da barra de ferramentas podem, por exemplo, ter desde uma função que abra outro

programa, até alterar um item que está sendo mostrado por um menu “popup” ou mostrar um texto em um campo de texto.

A figura abaixo mostra a interface principal do programa.



O fluxograma a seguir ilustra o processo de criação de uma *toolbar*



Para se criar uma barra de ferramentas, *toolbar*, deve-se acrescentar nos atributos do processo o atributo **ProcessToolBar**, conforme mostrado na figura a seguir.

```

1 module exToolBar
2 import StdEnv, StdIO
3
4 Start :: *World -> *World
5 Start world
6 # (ids.world)= openIds 10 world
7
8
9 # (ok1, file1, world)= fopen "sol.bmp" FReadData world
10 # (ok1, osbit1, file1)= osReadBitmap file1
11 # (ok1, world)= fclose file1 world
12 # bit1= toBitmap osbit1
13
14 # (ok1, file1, world)= fopen "fa.bmp" FReadData world
15 # (ok1, osbit1, file1)= osReadBitmap file1
16 # (ok1, world)= fclose file1 world
17 # bit2= toBitmap osbit1
18
19 # attributes = [ ProcessClose closeProcess,
20                 ProcessToolBar [ToolBarSeparator,
21                                 ToolBarSeparator,
22                                 ToolBarSeparator,
23                                 ToolBarSeparator,
24                                 ToolBarSeparator,
25                                 ToolBarSeparator,
26                                 ToolBarSeparator,
27                                 ToolBarSeparator,
28                                 ToolBarItem bit1 (Just "Clave de Sol") (claveSol ids),
29                                 ToolBarSeparator,
30                                 ToolBarSeparator,
31                                 ToolBarSeparator,
32                                 ToolBarSeparator,
33                                 ToolBarSeparator,
34                                 ToolBarItem bit2 (Just "Clave de Fa") (claveFa ids)
35                                 ]
36
37
38
39
40
41 = startIO SDI 0 (initialize ids) (attributes++[ProcessClose closeProcess]) world
42

```

Abre os 2 bitmaps que serão utilizados na criação dos botões.

Atributos de criação dos botões.

Cria o botão com a respectiva chamada da função

No mesmo deve-se colocar os itens da *toolbar* desejados através da função **ToolBarItem**. O formato da função **ToolBarItem** é como segue:

ToolBarItem bitmap_do_botao (tip_do_botao) (função do item)

A função **TollbarSeparator** é utilizada para colocar espaços entre os botões da barra de ferramenta. A novidade na função do botão é o controle do item que deverá ficar visível no menu popup, conforme exemplo da função **claveSol** e **claveFa**. No caso da função **claveSol** é o item 1 e o da função **claveFa** é o item 2 (linhas 100 e 105 respectivamente).

```

99 claveSol ids proc
100 # proc = appPIO (selectPopUpControlItem (ids!!1) 1) proc
101 # proc = appPIO (setControlText (ids!!2) "Clave de Sol") proc
102 = proc
103
104 claveFa ids proc
105 # proc = appPIO (selectPopUpControlItem (ids!!1) 2) proc
106 # proc = appPIO (setControlText (ids!!2) "Clave de Fá") proc

```

Nas linhas 101 e 106, quando os botões são pressionados, escrevem a mensagem “Clave de Sol” e “Clave de fá”, respectivamente, no campo de texto.

A figura abaixo mostra o programa completo.

```

1 module exToolBar
2 import StdEnv, StdIO
3
4 Start :: *World -> *World
5 Start world
6 # (ids, world) = openIds 10 world
7
8 # (ok1, file1, world) = fopen "sol.bmp" FReadData world
9 # (ok1, osbit1, file1) = osReadBitmap file1
10 # (ok1, world) = fclose file1 world
11 # bit1 = toBitmap osbit1
12
13 # (ok1, file1, world) = fopen "fa.bmp" FReadData world
14 # (ok1, osbit1, file1) = osReadBitmap file1
15 # (ok1, world) = fclose file1 world
16 # bit2 = toBitmap osbit1
17
18 # attributes = [ ProcessClose closeProcess,
19                 ProcessToolBar [ToolBarSeparator, ToolBarSeparator, ToolBarSeparator, ToolBarSeparator,
20                                 ToolBarSeparator, ToolBarSeparator, ToolBarSeparator, ToolBarSeparator,
21                                 ToolBarItem bit1 (Just "Clave de Sol") (claveSol ids),
22                                 ToolBarSeparator, ToolBarSeparator, ToolBarSeparator, ToolBarSeparator,
23                                 ToolBarSeparator, ToolBarSeparator,
24                                 ToolBarItem bit2 (Just "Clave de Fa") (claveFa ids)]]
25 = startIO SDI 0 (initialize ids) (attributes++[ProcessClose closeProcess]) world
26
27 initialize ids proc
28 # (maybeBitmap, proc) = openBitmap "fundo.bmp" proc
29 # bitmapFundo = fromJust maybeBitmap
30 # bitmapsize = getBitmapSize bitmapFundo
31 # (erro, proc) = openWindow Void (janela ids bitmapFundo bitmapsize) proc
32 | erro <> NoError = proc
33 # (erro, proc) = openMenu Void (menu) proc
34 | erro <> NoError = proc
35 | otherwise = proc
36 where
37     menu = Menu "Arquivo"
38           (MenuItem "Sair" [MenuFunction ((noLS closeProcess))])[]
39
40 nada proc = proc
41
42 janela ids bitmap bitmapsize
43 = Window "Exemplo de ToolBar"
44   ( EditControl "" (PixelFormat 180) 2 [ControlPos (Center, zero), ControlHide]
45     +: EditControl "" (PixelFormat 180) 2 [ControlPos (Center, zero), ControlHide]
46     +: PopUpControl [(Just "Clave de Sol", nada), (Just "Clave de Fa", nada)]
47     +: EditControl "" 1 [ControlPos (Center, zero), ControlId campo1]
48     +: EditControl "" (PixelFormat 180) 2 [ControlPos (Center, zero), ControlHide]
49     +: EditControl "" (PixelFormat 180) 2 [ControlPos (Center, zero), ControlId campo2])
50   [ WindowViewSize bitmapsize,
51     WindowPos (LeftTop, OffsetVector {vx=(maxFixedWindowSize.w-(bitmapsize.w))/2,
52                                       vy=(maxFixedWindowSize.h-(bitmapsize.h))/2}),
53     WindowLook True (\_ _->drawAt zero bitmap), WindowHMargin 3 3, WindowVMargin 0 0,
54     WindowItemSpace 7 3, WindowClose (noLS closeProcess), WindowId wd1]
55
56 where
57     wd1 = ids!!0
58     campo1 = ids!!1
59     campo2 = ids!!2
60
61 claveSol ids proc
62 # proc = appPIO (selectPopUpControlItem (ids!!1) 1) proc
63 # proc = appPIO (setControlText (ids!!2) "Clave de Sol") proc
64 = proc
65
66 claveFa ids proc
67 # proc = appPIO (selectPopUpControlItem (ids!!1) 2) proc
68 # proc = appPIO (setControlText (ids!!2) "Clave de Fa") proc
69 = proc

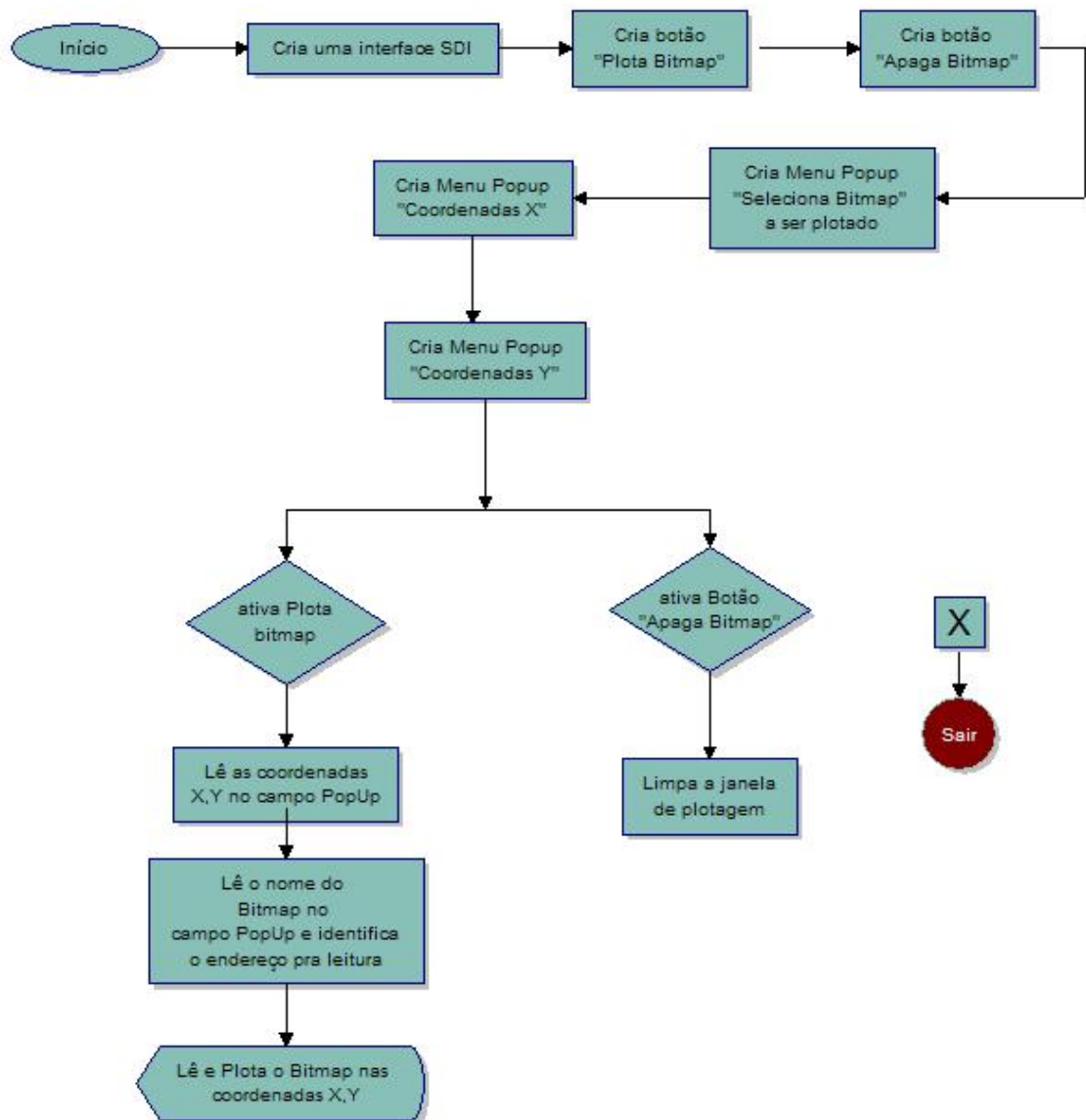
```

Exemplo de Plotar Bitmap

Este programa é um exemplo de como inserir uma figura no formato BMP (Bitmap) em qualquer lugar da janela aberta. Para tanto, disponibilizou-se quatro imagens possíveis de serem selecionadas em um menu popup e as coordenadas x e y (em *pixel*) de onde o mesmo será colocado. Para completar o exemplo, implementou-se uma função para apagar a janela.

O fluxograma, a seguir, mostra como é a lógica de implementação desta interface.

PLOTA BITMAP



Novidades deste tipo de implementação:

- **Criar uma janela SDI em branco, sem bitmap de fundo**