

-Se desejar abrir novamente o arquivo que estava editando, basta ativar o menu Arquivo e clicar na opção “Abrir arquivo .bhl”

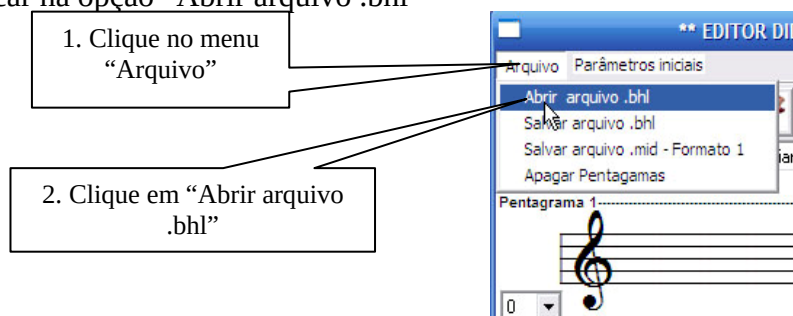


Figura 6.62 – Abrir arquivo .bhl

- O explorar é aberto para você localizar e abrir o arquivo desejado.

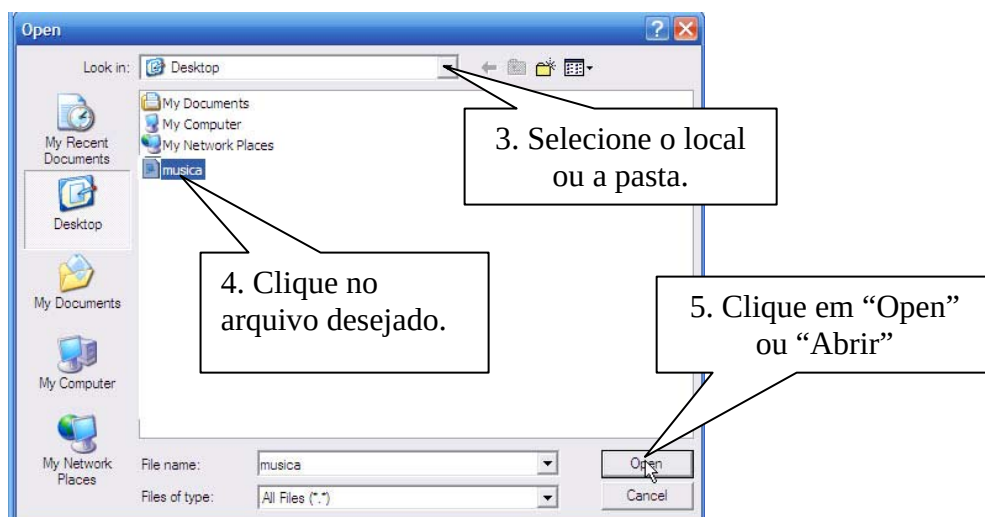


Figura 6.63 – Janela “Abrir”

- Se ao minimizar ou abrir uma janela a tela do seu aplicativo ficar branca totalmente ou em parte, para atualizá-la, voltar ao normal, basta clicar no botão de atualizar (refresh).

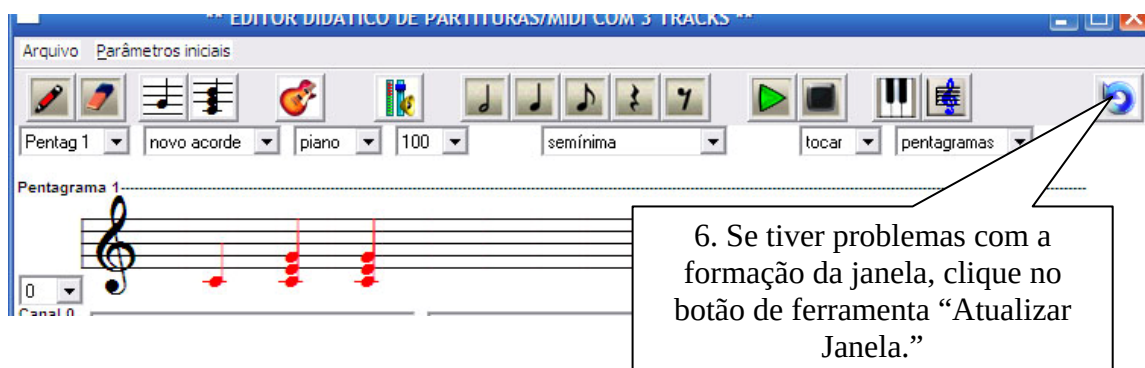


Figura 6.64 – Atualizar a janela

Salvar o arquivo em SMF formato 1

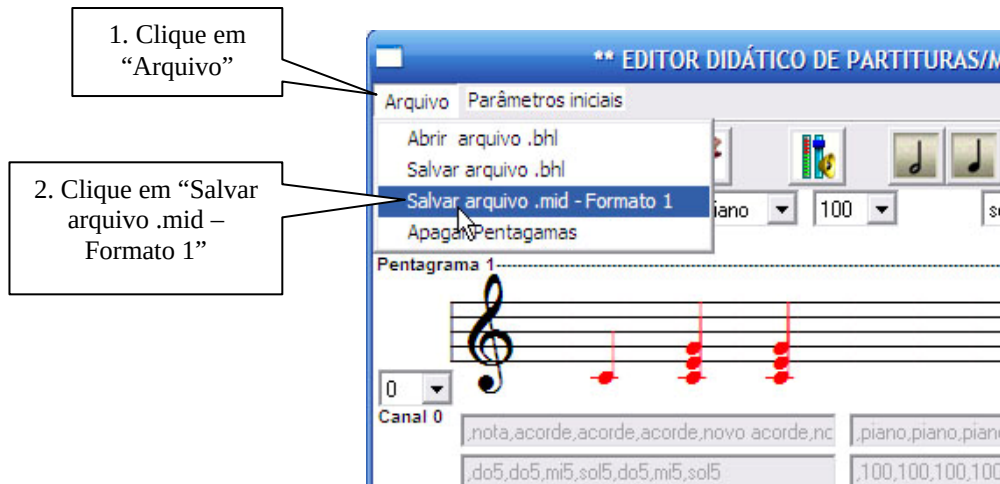


Figura 6.65 – Salvar o arquivo MIDI

- Ao fazer isto, o explorador novamente é aberto para que você converta o trabalho em formato MIDI 1 e salvá-lo.

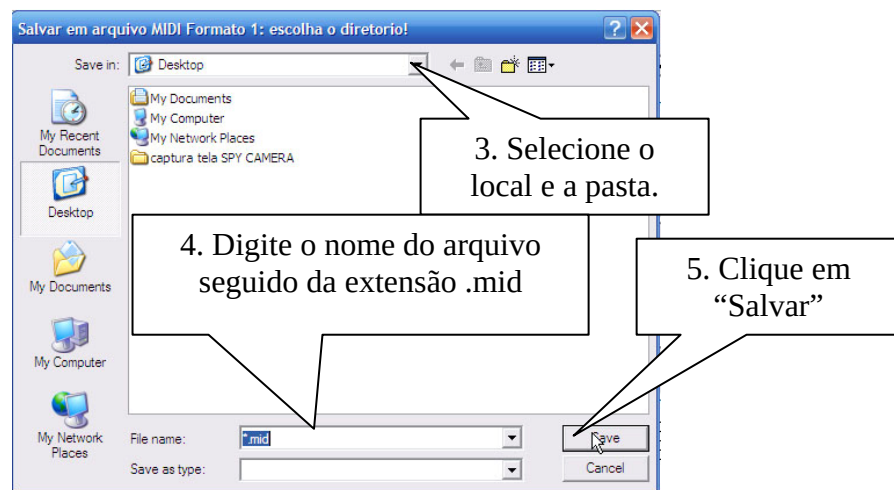


Figura 6.66 – Janela “Salvar”

- Para apagar notas do pentagrama ter-se-iam duas opções:

- 1- Eliminar a nota do pentagrama;
- 2- Colocar uma pausa no lugar da nota eliminada.

Decidiu-se pela primeira opção.

Para apagar uma nota, inicie escolhendo o modo apagar, e escolhendo o pentagrama que contém esta nota.

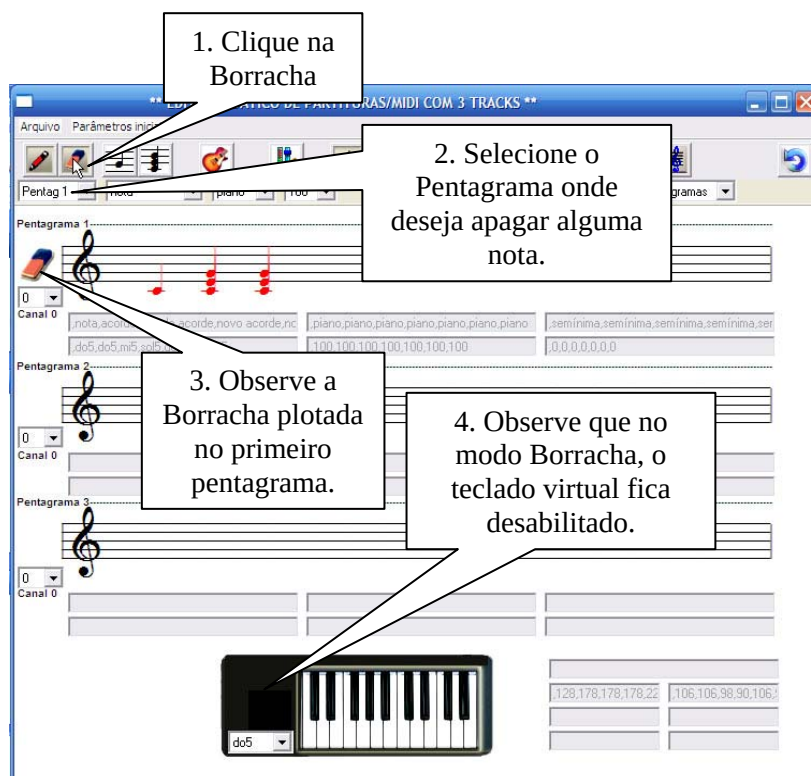


Figura 6.67 – Apagar notas

-Feito isto, posicione o ponteiro do mouse em cima da nota que deseja apagar. Quando estiver em cima de uma nota válida, no respectivo campo, no canto inferior direito da interface, mostrará a nota selecionada. Quando o ponteiro do mouse estiver fora de uma nota válida, aparecerá neste campo a mensagem “Nenhuma nota selecionada!”



Figura 6.68 – Nota selecionada

- Ao clicar na nota, com o ponteiro do mouse em cima dela, ela é eliminada do pentagrama e suas ocorrências são eliminadas de todas as listas (status nota/acorde, nota, instrumento, volume, canal, figura musical, coordenadas x e coordenadas y).

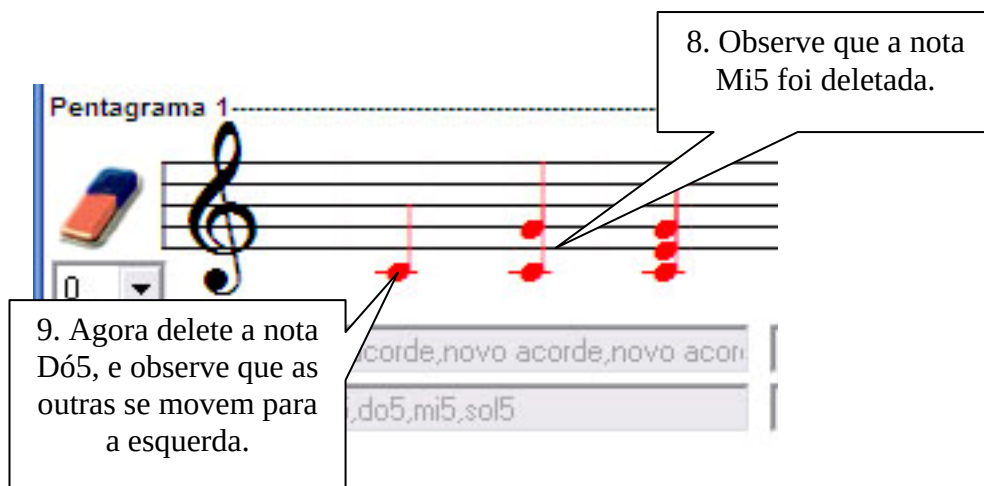


Figura 6.69 – Nota eliminada

- Se apagar uma nota ou um acorde inteiro, as notas subsequentes são *replotadas*.

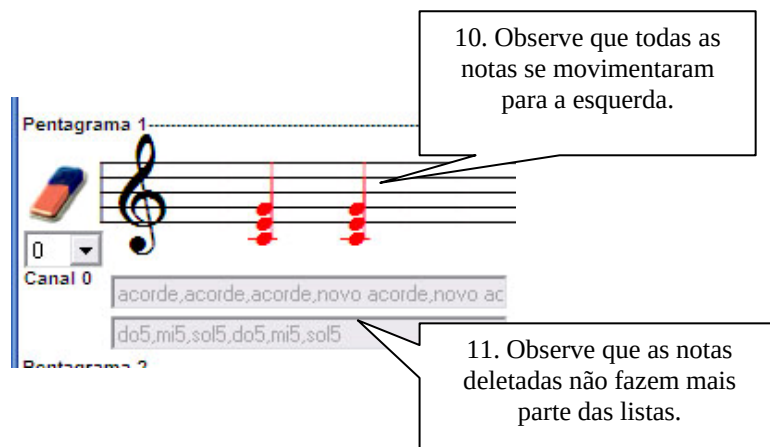


Figura 6.70 – Notas movimentam para esquerda

6.6 Conclusão.

Este capítulo apresentou estudos de casos das bibliotecas criadas para manipulação de arquivos MIDI SMF formato 0 e formato 1, definidos com várias estruturas e sintaxes diferentes, de forma a permitir uma maior flexibilidade ao programador e usuário no desenvolvimento de seus aplicativos MIDI. Mostrou também a utilização dos conceitos

na implementação de interfaces que exigem um controle da posição do mouse e de seus eventos destinados a realizar modificações significativas na interface, interagindo em tempo real com o usuário.

Capítulo 7

Conclusão e trabalhos futuros

7.1 Conclusão

Esta dissertação apresentou um estudo detalhado do protocolo MIDI, principalmente focando nos arquivos MIDI SMF, produzindo um material didático que permita a estudantes de música, computação e outras áreas entender, ler e produzir suas seqüências musicais, independente de suporte computacional. A mesma também apresentou um estudo de uma linguagem aderente à implementação de aplicativos MIDI multimídia interativo, produzindo um material, também didático, para o treinamento de outros profissionais para ingressarem no domínio da computação musical utilizando uma ferramenta e conceitos aderentes à implementação de soluções nesta área. Como estudo de casos, foram implementadas bibliotecas de funções para leitura e manipulação de arquivos MIDI, realizando transposições musicais, análise de range, e, entre outros, um editor com os recursos existentes na maioria dos editores musicais profissionais, possibilitando ao leitor entender os conceitos envolvidos e suas soluções. O material didático foi apresentado a um grupo de 7 estudantes do segundo período de engenharia elétrica, interessados na área, onde os mesmos validaram o texto didático, conseguindo apenas com o material reproduzir as ações feitas no texto, bem como implementar seus próprios programas.

Os objetivos propostos foram todos cumpridos, a saber:

1. Construção de bibliotecas de funções para se ler, editar e salvar arquivos MIDI SMF formato 0 e formato 1;
-> O capítulo 4 e o capítulo 6 apresentam três bibliotecas e várias funções que efetivam este objetivo.

2. Implementar uma biblioteca para se poder reproduzir arquivos MIDI;
-> As bibliotecas do capítulo 4 implementam tal solução.
3. Produzir um material didático para o ensino do protocolo MIDI e dos SMF formato 0 e formato 1;
-> O capítulo 2 se encarrega, junto com o capítulo 4, em apresentar os conceitos e exemplos que focam na teoria e implementação de soluções neste protocolo e SMF.
4. Definir e justificar a escolha de um paradigma e de uma linguagem de programação para implementação de sistemas multimídia interativos com interfaces visuais aderentes ao domínio musical com baixa relação custo/aprendizado;
-> O Capítulo 3 apresenta 16 (dezesesseis) motivos para a escolha do paradigma funcional e da linguagem Clean para adoção do paradigma funcional e da linguagem Clean para utilização em computação Musical.
5. Apresentar os conceitos e implementar exemplo aplicativo de construção de interfaces NDI e SDI⁶⁵
-> Conceitos e implementações mostradas no capítulo 5 e 6.
6. Apresentar os conceitos e implementar interfaces SDI com botões, campos de textos, *popups*, *radio buttons*, *check buttons*;
-> Conceitos e implementações mostradas no capítulo 5 e 6.
7. Apresentar os conceitos e implementar interfaces SDI com menus e barra de ferramentas;
-> Conceitos e implementações mostradas no capítulo 5 e 6.
8. Apresentar os conceitos e implementar um teclado virtual: plotagem de bitmaps em janelas e leitura de posicionamento e ações de mouse;
9. Apresentar os conceitos e implementar editores visuais de música multicanal e multitrack;
-> Conceitos e implementações mostradas no capítulo 5 e 6.
10. Apresentar os conceitos e implementar um kit didático para o ensino e construção de arquivos MIDI em baixo nível (em hexadecimal e decimal);
-> Conceitos e implementações mostradas no capítulo 2, 5 e 6.

⁶⁵ NDI – No Document Interface, SDI – Single Document Interface

11. Apresentar os conceitos e implementar soluções para inclusão de ícones nos arcabouços dos aplicativos gerados;
-> Conceitos e implementações mostradas no capítulo 5.
12. Mostrar como criar um projeto de instalação automática dos aplicativos desenvolvidos;
-> Conceitos e implementações mostradas no capítulo 5.
13. Promover estudos de casos e validações dos conceitos, bibliotecas e aplicativos gerados neste trabalho;
-> Conceitos e implementações mostradas no capítulo 6.

Assim, conclui-se este trabalho de dissertação de mestrado apresentando um material farto de conceitos, exemplos e implementações que servirão de base a outros estudantes para entender o Protocolo MIDI e os arquivos SMF, bem como, também, os conceitos e exemplos de implementações de funções e aplicativos visuais utilizando a linguagem funcional Clean.

7.2 Trabalhos futuros.

A partir das ferramentas e conceitos apresentados, vários projetos passam a se tornar viáveis, dentre eles:

- 1- Implementação de um editor completo de partituras, incluindo a leitura de SMF formato 0 e 1, número variável de tracks e compassos e um player com bancos virtuais de timbres;
- 2- Implementação de um editor de seqüências musicais para execução com dinâmica humanizada, onde a cada nota musical possa corresponder a uma dinâmica diferente, como: uso de palheta, unha ou dedo, *slap*, *bender*, harmônicos, trêmulo, enfim, onde a cada nota se possa associar uma dinâmica exclusiva de um instrumentista humano, visando tornar as reproduções mais fiéis ao que se faz por um humano acusticamente.
- 3- Implementação de um aplicativo visual para criação de composições a partir de um roteiro semântico qualquer (criação de trilhas sonoras), com aprendizado direto em arquivos SMF;

- 4- Gerar melodias, em editor personalizado, inéditas, baseadas em estilo de determinado compositor ou de uma determinada época, partindo de uma letra de música fornecida ao sistema, com aprendizado direto em arquivos SMF;
- 5- Interagir modelos de compositores diferentes, de mesma época e estilo (ou divergentes, se desejado for) para que os mesmos possam em conjunto elaborar arranjos multi-instrumentais;
- 6- Implementar um modelo baseado em entropia de informação e estilo que permita o pré-julgamento de uma música inédita no tocante às reais possibilidades da mesma vir a se tornar um sucesso de público;
- 7-Criar novas músicas aplicadas à musicoterapia na indústria agropecuária no incremento de produção de produtos;

Referências

- [1] MACHADO, André Campos. **Tradutor de Arquivos MIDI para Texto Utilizando Linguagem Funcional CLEAN**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2001.
- [2] LIMA, Sandra Fernandes de Oliveira. **Um sistema para transposição Automática de Seqüências MIDI baseada em Alcance Vocal**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2006.
- [3] LOPES, Guilherme Francisco Lopes. **Desenvolvimento de uma biblioteca e uma calculadora Midi, em linguagem funcional, para análise e manipulação de Arquivos padrão Midi (SMF) formato 0 e 1**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2004.
- [4] **MIDI Detailed Specification 1.0**. International MIDI Association, 1983, 1991, 1994.
- [5] LIMA, Luciano Vieira; RUFINO, Hugo Leonardo Pereira; GOULART, Reane Franco; CURY FILHO, Reny. **Linguagem Funcional CLEAN: uma solução moderna, eficiente e aderente à modelagem de funções e ao ensino da Matemática**. In: CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, São José do Rio preto, 2003, v.1, p.1-7.
- [6] RUFINO, Hugo Leonardo Pereira. **Criação e Utilização de DLLs e Objs em Linguagem Funcional CLEAN e seu Interfaceamento com C**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2004.
- [7] COSTA, Antônio Eduardo; **My Clean Book** (2006).
- [8] LIMA, Luciano Vieira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Cakewalk SONAR 2.0**. 1a. ed. São Paulo: Editora Érica Ltda, 2002.
- [9] LIMA, L.; C.A.L., et al. **Automatic Learning Composition Base don Polyphonic Wave Signals and Musical Histograms**. Tatra Mt Match. Publ. 23, 2001.
- [10] COPE, D. **Experiments Musical in Intelligence (EMI): Non-linear Linguistic-Based Composition Interface**, vol., 18, pp. 117-139, 1989.

- [11] COPE, D. **Computer Modelling of Musical Intelligence in EMI**. Computer Music Journal, vol. 16, no. 2, Summer, pp. 69-83, 1992.
- [12] COPE, D. **Music & Lisp**. OH Expert, March, pp.26-33, 1988.
- [13] LIMA, L. V. **Um Sistema de Composição Musical Dirigido por Estilo**. Tese de Doutorado, Universidade de São Paulo, São Paulo - Brasil, 1998.
- [14] LIMA, Luciano Vieira. Real time high performance platform for e-learning with high resolutions áudio, frames and vídeo file. In: **3ª. Conferência do PGL: Partnership in Global Learning “Consolidando Experiências em e-Learning”**, FGV, São Paulo – Brasil, 2005.
- [15] LIMA, Luciano Vieira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Cakewalk sonar 2.0 : Seqüenciamento e Técnicas de Estúdio Audiodigital**. 2ª. ed. São Paulo: Editora Érica Ltda, 2003.
- [16] LIMA, Luciano Vieira; LIMA; Sandra Fernandes de Oliveira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Computação Musical : Arranjo Automático, Conversão de CD em MIDI e MIDI em Áudio utilizando o Band-in-a-Box, Virtual Sound Canvas 3.23 e Akoff Music Composer 2.0**. 1ª. ed. São Paulo: Editora Érica Ltda, 2004.
- [17] LIMA, Luciano Vieira; LIMA; MACHADO, André Campos; PINTO, Marília Mazzaro. **Finale 2004: Editoração de Partituras, Composição e Arranjo**. 1ª. ed. São Paulo: Editora Érica Ltda, 2004.
- [18] PASCHOAL, Fausto de. **Música no Computador com Sibelius 3 para Windows**. 1ª. ed. São Paulo: Editora Érica Ltda, 2004.
- [19] LIMA, Luciano Vieira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Encore 4.5.4 : Editoração de Partituras**. 1a. ed. São Paulo: Editora Érica Ltda, 2003.
- [20] LIMA, Luciano Vieira; LIMA; Sandra Fernandes de Oliveira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Sound Forge 8.0: Gravação ao Vivo, Restauração de Sons de LPs e Masterização áudio Digital**. 1ª. ed. São Paulo: Editora Érica Ltda, 2005.
- [21] RATTON, Miguel. **Criação de Música e Sons no Computador**. Rio de Janeiro: Editora Campus, 1995.
- [22] GROUT, D.; PALISCA, C. **História da Música Ocidental**. Lisboa: Gradiva, 1994.

- [23] LIMA, Florêncio de Almeida Lima. **Elementos Fundamentais da Música**. 4^a.ed. Rio de Janeiro,1958.
- [24] BAS, **Trattato di Forma Musicale**. Milão: Ricordi, 1964.
- [25] ZAMACOIS, Joaquín. **Curso de Formas Musicales**. Cuarta edición. Barcelona: Editorial Labor S.A.,1960.
- [26] WINSTON & HORN. **LISP**. 3rd Edition. Addison-Wesley Publishing Company, 1980.
- [27] LIMA, L. V. et al. **Editing Standard MIDI File in Functional Language CLEAN**. Diderot 99, Vienna, 1999.
- [28] WELLESLEY,Alessandro Barros. **Programação Funcional com Clean 2.1 para Windows**. Uberlândia: Editora Pietro Ltda, 2004.
- [29] BIRD, Richard. **Introduction to Functional Programming using Haskell, 2nd edition**, Prentice Hall press,1998.
- [30] THOMPSON, Simon. **Haskell: The Craft of Functional Programming**, Second Edition, Addison-Wesley, 507 pages, paperback, 1999.
- [31] LIMA, Sandra Fernandes de Oliveira; RUFINO, Hugo Leonardo Pereira; LOPES, Guilherme Francisco; GOULART, Reane Franco; LIMA, Luciano Vieira. **Matemática aplicada à detecção de tessitura vocal e à transposição automática de músicas em arquivos MIDI padrão**. In: CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, São José do Rio preto, 2003, v.1, p.1-7.
- [32] MARTINS, Gláucia Macedo Mendes. **Projeto e Implementação de uma Interface Visual Interativa para Montagens de Conteúdo de Autoria Multimídia**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2005.

Anexo I

Monotimbralidade.

Em música, chama-se **timbre** à característica sonora que nos permite distinguir se sons de mesma frequência foram produzidos por fontes sonoras conhecidas e que nos permite diferenciá-las. Quando ouvimos, por exemplo uma nota tocada por um piano e a mesma nota (uma nota com a mesma altura) produzida por um violino, podemos imediatamente identificar os dois sons como tendo a mesma frequência , mas com características sonoras muito distintas. O que nos permite diferenciar os dois sons é o timbre instrumental. De forma simplificada podemos considerar que o timbre é como a impressão digital sonora de um instrumento ou a qualidade de vibração vocal.

Embora as características físicas responsáveis pela diferenciação sonora dos instrumentos sejam bem conhecidas, a forma como ouvimos os sons também influencia na percepção do timbre.

Nos sintetizadores e teclados mais antigos, o módulo de som tinha a característica de ajustar somente um mesmo timbre para todas as vozes do instrumento. Dessa forma o músico só podia tocar utilizando um timbre de cada vez, o que era uma limitação na época.

Anexo II

Especificação MIDI 1.0.

Descritivo em Português elaborado por Miguel Ratton

INTRODUÇÃO

MIDI (Musical Instrument Digital Interface) é um padrão de transmissão serial de dados, que permite a troca de informações entre instrumentos e equipamentos de aplicação musical.

Os equipamentos que implementam funções MIDI, podem não conter todas as funções previstas pelo padrão, mas aquelas implementadas devem seguir estritamente o que está padronizado.

CONVENÇÕES

A representação de números neste texto segue a seguinte convenção:

- Números representados na base hexadecimal são sempre seguidos de um H, como 34H.
- Números representados na base binária são sempre representados precedidos de um \$, como \$10001011.
- Os demais números estão na base decimal.

Os termos técnicos estão traduzidos, sempre que possível, mas os termos originais, em inglês, foram preservados e são citados, de forma a facilitar a leitura de textos estrangeiros, muito comuns nesta área.

O termo equipamento neste texto significa qualquer dispositivo eletrônico de aplicações musicais: instrumentos musicais, módulos de efeitos sonoros, baterias eletrônicas, seqüenciadores, equipamentos de áudio, etc.

CARACTERÍSTICAS DE HARDWARE

Os equipamentos que implementam o interfaceamento MIDI devem possuir um transmissor ou um receptor, ou ambos, que execute a transmissão/recepção de mensagens no padrão MIDI. Tanto transmissão quanto recepção devem operar à uma taxa de 31.250 bauds (+/- 1%), em modo assíncrono, com um bit de início (start bit), 8 bits de dados (D0 a D7) e um bit de fim (stop bit), perfazendo um total de 10 bits por byte serial, e ocupando um período de 320 micro-segundos.

O hardware do transmissor é composto de um UART (Universal Asynchronous Receiver/Transmitter) para transmissão dos dados seriais, e um acionador (buffer) de saída, capaz de drenar uma corrente de até, 5 mA. A saída deve possuir resistores de proteção na eventualidade de curto-circuito entre os terminais. O conector de saída deve ser aterrado ao equipamento.

O hardware do receptor é composto de um acoplador óptico e um UART para recepção dos dados seriais. O acoplador óptico deve ser capaz de chavear com uma corrente menor ou igual a 5 mA no LED, e os tempos de subida e de descida não devem ultrapassar 2 micro-segundos, sendo recomendados os acopladores ópticos PC-900 (Sharp) e 6N138 (HP). O conector de entrada não deve ser aterrado ao equipamento.

Os conectores do equipamento devem ser do tipo DIN de 5 pinos (fêmea) dispostos em 180 graus, e montados em painel. Os conectores devem ser identificados como MIDI IN (entrada), MIDI OUT (saída) e, opcionalmente, MIDI THRU (repetição da entrada). São recomendados conectores Switchcraft 57 GB5F.

A saída opcional MIDI THRU deve prover uma cópia direta dos dados que entram pelo conector MIDI IN, de forma a permitir a ligação de mais de dois equipamentos em cadeia. No caso de cadeias muito longas (mais de três equipamentos), o padrão

recomenda a utilização de acopladores ópticos mais rápidos, de forma a evitar que o acúmulo de atrasos de subida e descida no acoplador afetem a largura dos pulsos de sinal. O conector MIDI THRU deve ser aterrado ao equipamento.

O cabo de interligação de equipamentos deve possuir dois condutores e mais a blindagem, e não deve ter um comprimento o maior do que 15 metros(50 p,s). Os plugs devem ser do tipo DIN (macho) de 5 pinos em 180 graus, com a blindagem conectada ao pino 2 em ambas as extremidades. É recomendado o plug Switchcraft 05 GM5M. O diagrama abaixo apresenta uma sugestão para os circuitos de saída e de entrada, conforme estabelece o padrão.

FORMATAÇÃO DOS DADOS

A comunicação pelo padrão MIDI utiliza, basicamente, mensagens do tipo multi-bytes contendo um Byte de Status (Status Byte) seguido de um ou dois Bytes de Dados (Data Bytes). As exceções à essa regra são as mensagens de Tempo Real e as de Sistema Exclusivo. A seguir, é apresentada a classificação das mensagens MIDI.

As mensagens MIDI são classificadas em duas categorias principais, que são as Mensagens de Canal (Channel Messages) e as Mensagens de Sistema(System Messages).

- **MENSAGENS DE CANAL (CHANNEL MESSAGES).**

As Mensagens de Canal são identificadas pelos quatro bits mais significativos do Byte de Status. Os quatro bits menos significativos determinam o número do canal de MIDI (1 a 16) em que a mensagem está sendo transmitida. Isso significa que a mensagem é endereçada ao(s)equipamento(s) sintonizado(s) naquele canal.

As Mensagens de Canal são subdivididas em dois tipos: Mensagens de Voz (Voice Messages), que controlam as vozes (geradores de som) do equipamento, e são transmitidas pelo Canal de Voz (Voice Channel); e as Mensagens de Modo (Mode Messages), que definem como o equipamento deve responder à recepção das Mensagens de Voz. As Mensagens de Modo são transmitidas pelo Canal Básico (Basic Channel).

Tabela 1 - Mensagens de Voz (Channel Voice Messages)		
BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
\$1000nnnn	\$0kkkkkkk \$0vvvvvvvv	Nota Desativada (Note Off) \$0vvvvvvvv = veloc. que a tecla é solta
\$1001nnnn	\$0kkkkkkk \$0vvvvvvvv	Nota Ativada (Note On) \$0vvvvvvvv = veloc. que a tecla é abaixada \$0vvvvvvvv = 0 --> Nota Desativada
\$1010nnnn	\$0kkkkkkk \$0vvvvvvvv	Pressão na Tecla (Polyph. Aftertouch) \$0vvvvvvvv = valor da pressão
\$1011nnnn	\$0ccccccc \$0vvvvvvvv	Controle (Control Change) \$0ccccccc = número do controle (0 a 121) \$0vvvvvvvv = valor do controle \$0ccccccc = 122 a 127: vide Tabela 3
\$1100nnnn	\$0pppppppp	Mudança de Programa (Program Change) \$0pppppppp = número do programa (0 a 127)
\$1101nnnn	\$0vvvvvvvv	Pressão no Teclado (Channel Aftertouch) \$0vvvvvvvv = valor da pressão
\$1110nnnn	\$0gggggggg \$0hhhhhhh	Variação do Pitch Bend (Pitch Bend Change) Primeiro byte de dados é a parte menos significativa (LSB), e o segundo byte de dados é a parte mais significativa (MSB) (Vide Nota 10)

NOTAS:

1) \$nnnn = N-1, onde N é o número do canal, ou seja, \$0000 é Canal 1, \$0001 é Canal 2, ..., \$1111 é Canal 16.

2) \$kkkkkkk = número da tecla (nota), que vai de 0 a 127. A nota de número 60 corresponde ao Dó central do teclado.

3) \$vvvvvvv = velocidade com que a tecla foi pressionada. A faixa de valores possíveis de velocidade deve obedecer à uma escala logarítmica, do tipo abaixo:

0 1.....64.....127

off.....ppp.....pp.....p.....mp.....mf.....f.....ff.....fff

\$vvvvvvv=64, caso o teclado não tenha sensibilidade à velocidade. Nota Ativada com vel=0 significa Nota Desativada com vel=64.

4) Toda mensagem de Nota Ativada (Note On) deve ser seguida, em algum momento, de uma mensagem de Nota Desativada para a mesma nota, no mesmo canal.

5) \$ccccccc = Número do Controle, conforme a Tabela 3.

6) Enquanto as mensagens enviadas tiverem o mesmo Byte de Status, este pode ser omitido, ficando como Status Corrente (Current Status), até que um novo Byte de Status seja requerido.

7) A sensibilidade à variação do Pitch Bend é determinada no equipamento receptor. O padrão define que a um Pitch Bender indo para a posição central deve enviar uma mensagem com valor 2000H.

Tabela 2 - Mensagens de Modo (Channel Mode Messages)

BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
\$0111nnnn	\$01111010	Controle Local (Local Control) \$0vvvvvvvv=0: desliga Controle

	\$0vvvvvv	Local \$0vvvvvv=127: liga Controle Local
\$1011nnnn	\$01111011 \$00000000	Solta Todas as Teclas (All Notes Off)
\$1011nnnn	\$01111100 \$00000000	Desativa Modo Omni (Omni Off) (e solta todas as teclas)
\$1011nnnn	\$01111101 \$00000000	Ativa Modo Omni (Omni On) (e solta todas as teclas)
\$1011nnnn	\$01111110 \$0vvvvvv	Ativa Modo Monofonico (Mono Mode) (e solta todas as teclas) \$0vvvvvv é o número de canais \$0vvvvvv=0: o número de canais é igual ao número de vozes do receptor.
\$1011nnnn	\$01111111 \$00000000	Ativa Modo Polifonico (Poly Mode) (e solta todas as teclas)

NOTAS:

1) \$nnnn = N-1, onde N é o número do canal, ou seja, \$0000 é Canal1, \$0001 é Canal 2, ..., \$1111 é Canal 16..

2) Com exceção da mensagem de Controle Local, as demais mensagens de Modo funcionam como mensagens de Soltar Teclas, que fazem com que todas as notas que porventura estejam tocando no equipamento receptor sejam comandadas através do Canal Básico para cessar, como se tivessem sido recebidas as respectivas mensagens de Nota Desativada. Entretanto, o padrão não recomenda o seu uso em lugar de mandassem específica de Nota Desativada. No caso do receptor não ter qualquer nota tocando, estas mensagens de Soltar Todas as Teclas devem ser ignoradas.

3) A mensagem de Controle Local é usada opcionalmente para desvincular internamente o teclado dos circuitos geradores de som. Ao receber uma

mensagem de Desativar Controle Local, o equipamento passa a ser comandado apenas via MIDI, e não mais pelo seu teclado. A mensagem de Ativar Controle Local restaura a situação original..

4) O terceiro byte da mensagem de Modo Monofonico especifica o número de canais de MIDI nos quais as Mensagens de Voz serão enviadas.

Tabela 3 - Controles (control change)
atualizado até jan/93

NÚMERO	DESCRIÇÃO DO CONTROLE
0	Seleção de Banco de Programas - MSB (Bank Select)
1	Roda/Alavanca de Modulação - MSB (Modulation Wheel)
2	Controle por Sopros - MSB (Breath Controller)
3	Não definido
4	Pedal MSB (Foot Controller)
5	Tempo do Portamento - MSB (Portamento Time)
6	Entrada de Dados - MSB (Data Entry)
7	Volume Principal - MSB (Main Volume)
8	Equilíbrio - MSB (Balance)
9	Não definido
10	Pan - MSB (Pan)
11	Controle de Expressão - MSB (Expression Controller)
12	Effect Control 1
13	Effect Control 2
14 - 15	Não definidos
16 - 19	Controles de uso Geral 1 a 4 - MSB (General Purpose)
20 - 31	Não definidos
32 - 63	LSB dos Controles 1 a 31
64	Pedal de Sustain (Sustain/Damper Pedal)
65	Liga/Desliga Portamento (Portamento On/Off)
66	Pedal de Sostenuto (Sostenuto Pedal)
67	Pedal Abafador (Soft Pedal)
68	Pedal de Legato
69	Pedal de Sustain 2 (Hold 2)
70	Controle de som 1 (Sound Variation)
71	Controle de som 2 (Timbre/Harmonic Content)
72	Controle de som 3 (Release Time)
73	Controle de som 4 (Attack Time)
74	Controle de som 5 (Brightness)
75 - 79	Controles de som 6 - 10 (ainda não determinados)
80 - 83	Controles de Uso Geral 5 a 8 (General Purpose)
84	Controle de portamento
85 - 90	Não definidos
91	Intensidade do Reverb (Ext. Effect Depth)

92	Profundidade do Tremolo (Tremolo Depth)
93	Profundidade do Chorus (Chorus Depth)
94	Profundidade do Batimento (Celeste Detune Depth)
95	Profundidade do Phaser (Phaser Depth)
96	Incremento (Data Increment)
97	Decremento (Data Decrement)
98	Número de Parâmetro - LSB (Parameter Number)
99	Número de Parâmetro - MSB (Parameter Number)
100	Número de Parâmetro - LSB (Parameter Number)
101	Número de Parâmetro - MSB (Parameter Number)
102 - 119	Não definidos
120 - 127	Reservados para as Mensagens de Modo (vide Tabela 2)

NOTAS:

- 1) Os controles 0 a 63 são do tipo controle contínuo, e podem assumir valores de até 14 bits. Os controles 64 a 95 são chaves do tipo liga/desliga.
- 2) Os fabricantes podem alocar novos controles aos números não definidos, desde que forneçam as informações pertinentes, no manual de operação do equipamento.
- 3) Os controles contínuos são divididos em Bytes Mais significativos(MSB) e Bytes Menos Significativos (LSB). Caso o valor do controle não ultrapasse sete bits, apenas o MSB é transmitido. Caso haja necessidade de representar o valor com maior resolução do que sete bits, são enviados os Bytes MSB e LSB, nessa ordem. Se o valor do controle foi alterado apenas na parte LSB, pode ser transmitido apenas o Byte LSB, sem o envio do MSB.
- 4) Nos controles do tipo chave liga/desliga, o valor \$vvvvvvv = 0 significa desligado (off), enquanto \$vvvvvvv = 127 significa ligado (on). Valores de 1 a 126, inclusive, devem ser ignorados.

- **MENSAGENS DE SISTEMA (SYSTEM MESSAGES)**

As Mensagens de Sistema não utilizam canal de MIDI, e são subdivididas em três tipos:

Mensagens Comuns (System Common Messages), que são comuns a todos os equipamentos de uma cadeia;

Tabela 4 - Mensagens Comuns de Sistema (System Common Messages)		
BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
\$11110001	\$0nnndddd	Reservada para MIDI Time Code (MTC)
\$11110010	\$0ggggggg \$0hhhhhhh	Ponteiro de Seq. (Song Position Pointer) \$0ggggggg=parte menos significativa (LSB) \$0hhhhhhh=parte mais significativa (MSB)
\$11110011	\$0sssssss	Seletor de Seqüência (Song Select) \$0sssssss=número da seqüência
\$11110100	-	Não definido
\$11110101	-	Não definido
\$11110110	-	Requisita Afinação (Tune Request)
\$11110111	-	Fim de Mensagem Exclusiva

NOTAS:

1) O Ponteiro de Seqüência (Song Position Pointer) é um registro interno que mantém o número de tempos (beats) MIDI desde o início da música (1 tempo MIDI é igual a 6 clocks MIDI). Normalmente ele, ajustado para zero, no momento em que o botão de INICIAR (START), pressionado, dando início à execução da seqüência. O ponteiro então é incrementado a cada seis clocks MIDI, até que seja pressionada a tecla PARAR (STOP). Se a tecla CONTINUAR é pressionada, o ponteiro continua a ser incrementado de onde havia parado. O termo seqüência refere-se a qualquer grupo de notas armazenados com seus devidos tempos de ocorrência e de duração. Inclusive seqüências de bateria eletrônica.

2) Seleção de Sequência (Song Select) especifica qual a sequência (música) a ser executada quando da ocorrência de uma mensagem de INICIAR.

3) EOX é usado como marca de fim de transmissão de Mensagem Exclusiva.

Mensagens de Tempo Real (System Real-Time Messages), que também são comuns a todos os equipamentos de uma cadeia. Essas mensagens só possuem Byte de Status (nenhum Byte de Dados), e podem ser enviadas a qualquer momento, mesmo entre bytes de uma mensagem com outro Byte de Status. Nesse caso, a Mensagem de Tempo Real deve ser tratada (ou ignorada, se o equipamento não implementar funções de Tempo Real), sem prejuízo da mensagem dentro da qual ela foi inserida;

Tabela 5 - Mensagens de Sistema em Tempo Real (System Real Time)

BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
\$11111000	-	Clock de MIDI (Timing Clock)
\$11111001	-	Não definido
\$11111010	-	Iniciar (Start)
\$11111011	-	Continuar (Continue)
\$11111100	-	Parar (Stop)
\$11111101	-	Não definido
\$11111110	-	Sensor de Atividade (Active Sensing)
\$11111111	-	Reset (Reset)

NOTAS:

- 1) As Mensagens em Tempo Real tem por finalidade sincronizar todos os equipamentos de um sistema operando em tempo real.
- 2) O padrão permite que elas possam ser enviadas em qualquer momento, mesmo inseridas entre bytes de outras mensagens.
- 3) O Clock de MIDI o sistema sincronizar-se. Os clocks de MIDI enviados sempre à razão de 24 clocks por semínima.
- 4) A mensagem de Início (Start) é enviada imediatamente após a tecla de Início ter sido pressionada no equipamento (seqüenciador, bateria eletrônica, etc.).
- 5) A mensagem de Sensor Ativo (Active Sensing) é opcional, tanto para transmissores quanto para receptores. O padrão recomenda que essa mensagem seja enviada a cada 300 milissegundos, no máximo, caso não tenha havido o envio de qualquer outra mensagem nesse intervalo de tempo. O receptor ignorar tal mensagem até que receba a primeira, quando então, passar a esperá-la quando não houver recepção de qualquer mensagem MIDI dentro de um intervalo de 300 ms. Passados os 300 ms sem qualquer mensagem nem mensagem de Sensor Ativo, o receptor apagar todas as vozes (all notes off) e retornar à operação normal. Os equipamentos que não implementam Sensor Ativo devem ignorar estas mensagens.
- 6) A mensagem de Reset serve para Inicializar o sistema, e não deve ser enviada automaticamente, ao ser ligado o instrumento.

Mensagens Exclusivas (System Exclusive Messages), que podem conter qualquer quantidade de Bytes de Dados, além do Byte de Status. Como o número de Bytes de Dados é variável, o fim de uma Mensagem Exclusiva é detectado ao chegar um byte de Fim de Mensagem Exclusiva (EOX, End of Exclusive) ou um novo Byte de Status.

As Mensagens Exclusivas devem incluir em seus Bytes de Dados um Código de Identificação do Fabricante (ID code). Caso o equipamento receptor não reconheça o Código de Identificação, dever ignorar os Bytes de Dados até o fim da mensagem.

O padrão recomenda que os fabricantes divulguem a documentação das suas Mensagens Exclusivas, de forma a permitir que os usuários possam acessar plenamente os equipamentos via MIDI.

Tabela 6 - Mensagens Exclusivas de Sistemas (System Exclusive Messages)

BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
\$11110000	\$0zzzzzzzz : :	Transferência de Dados (Bulk Dump) \$0zzzzzzzz=identificação Pode ser inserido aqui qualquer número de bytes, para qualquer propósito, desde que tenham sempre o bit mais significativo igual a zero.
\$11110111	-	Fim de Mensagem Exclusiva (EOX)

NOTAS:

- 1) Nenhum Byte de Status ou de Dados deve ser inserido no meio de uma Mensagem Exclusiva, exceto Mensagens de Tempo Real.
- 2) A Mensagem Exclusiva termina com um EOX ou qualquer outro Byte de Status.

TIPOS DE BYTES

Existem dois tipos de bytes nas mensagens MIDI: os Bytes de Status (Status Bytes) e os Bytes de Dados (Data Bytes).

- BYTE DE STATUS (STATUS BYTE)

O Byte de Status é um número binário de 8 bits, sendo que o bit mais significativo (bit 7, mais à esquerda) tem sempre valor 1. O Byte de Status serve para identificar que categoria e tipo de mensagem está sendo transmitida. Como já foi mencionado anteriormente, excetuando-se os Bytes de Status de Mensagens de Sistema em

Tempo Real, que podem ser inseridos no meio de outras mensagens, um novo Byte de Status sempre indica uma nova mensagem, mesmo que a mensagem anterior não tenha sido completada.

Tabela 7 - Bytes de Status		
BYTE DE STATUS	QUANTIDADE DE BYTES DE DADOS	DESCRIÇÃO
\$1000nnnn \$1001nnnn \$1010nnnn \$1011nnnn \$1100nnnn \$1101nnnn \$1110nnnn	2 2 2 2 1 1 2	MENSAGENS DE VOZ (Channel Voice) Nota Desativada (Note Off) Nota Ativada (Note On) Pressão na Tecla (Polyph. Aftertouch) Controle (Control Change) Mudança de Programa (Program Change) Pressão no Teclado (Channel Aftertouch) Variação do Pitchbend (Pitchbend Change)
\$1011nnnn	2	MENSAGENS DE MODO (Channel Mode) Seleciona o Modo de Operação
\$11110000 \$11110sss \$11111ttt	***** 0 a 2 0	MENSAGENS DE SISTEMA (System Messages) Exclusiva (System Exclusive) Comum (System Common) Tempo-Real (System Real Time)

• NOTAS:

1. \$nnnn = N-1, onde N é o número do canal, ou seja, \$0000 é Canal1, \$0001 é Canal 2, ..., \$1111 é Canal 16.
2. ***** = \$0iiiiiii, <dados>, EOX.
3. \$0iiiiiii = Identificação do Fabricante/Equipamento.
4. \$sss = \$000 a \$111 (1 a 7).

5. \$ttt: idem.

- STATUS CORRENTE (RUNNING STATUS)

No caso específico das Mensagens de Voz e de Modo, quando um Byte de Status é recebido e processado, o equipamento receptor deve armazenar aquele Status até que um novo Byte de Status seja recebido. O padrão determina que, se um transmissor tiver que transmitir uma nova mensagem cujo Byte de Status é idêntico ao último transmitido, ele pode (opcionalmente) omiti-lo, transmitindo apenas os Bytes de Dados seguintes.

O Byte de Status omitido, nesse caso, é considerado como Status Corrente (Running Status). O receptor deve ser capaz de interpretar a transmissão com ou sem Status Corrente. A vantagem desse artifício é que no caso de grande quantidade de mensagens de mesmo Status (como por exemplo, uma série de teclas pressionadas), onde as mensagens ficam menores. O Status Corrente é interrompido sempre que chegar outro Byte de Status, exceto se for um Byte de Status de uma Mensagem de Sistema em Tempo Real, como já foi explicado anteriormente.

- STATUS NÃO IMPLEMENTADOS

O equipamento receptor deve ignorar os Bytes de Status (e respectivos Bytes de Dados, se houver) que identifiquem funções não implementadas por ele.

- STATUS INDEFINIDOS

O padrão recomenda que os códigos de Bytes de Status ainda não definidos não sejam transmitidos. Além disso, recomenda que se tome todo o cuidado necessário para evitar a transmissão de mensagens espúrias ao ligar ou desligar o equipamento

transmissor. Qualquer Byte de Status não definido deve ser ignorado pelo equipamento receptor, juntamente com os Bytes de Dados subsequentes.

- **BYTES DE DADOS**

Exceto nas Mensagens de Sistema em Tempo Real, os Bytes de Status são sempre seguidos por um ou dois Bytes de Dados. Esses bytes são números binários de 8 bits, nos quais o bit mais significativo (bit 7, mais à esquerda) tem sempre valor zero. Os valores que os Bytes de Dados podem assumir, dependendo do Byte de Status a que pertencem, estão definidos nas Tabelas, mais adiante. O transmissor deve sempre enviar o número correto de Bytes de Dados definido pelo padrão para cada Byte de Status. O receptor, por sua vez, só pode processar a mensagem ao recebê-la completamente, isto é, deve esperar a recepção de todos os Bytes de Dados esperados para o Status Corrente. O receptor deve ignorar quaisquer Bytes de Dados recebidos, que não tenham sido precedidos por um Byte de Status válido, exceto no caso do Status Corrente, citado anteriormente.

MODOS DE OPERAÇÃO

Os instrumentos musicais eletrônicos (sintetizadores, samplers, órgãos eletrônicos, etc.) possuem dispositivos internos (hardware ou software) responsáveis pela geração das notas (sons). Esses dispositivos, chamados de vozes (voices), são em número limitado (em geral 8, 16 ou 32), e determinam a polifonia do instrumento.

Quando uma tecla é pressionada no teclado do instrumento, o processador interno do mesmo verifica se há alguma voz disponível, isto é, se o instrumento já está tocando o número máximo de notas possível. Se não está ele aloca uma voz disponível à nota correspondente à Nota Ativada. À medida que as teclas são soltas, as vozes vão ficando livres (disponíveis) para serem ocupadas por novas notas. Caso todas as vozes estejam ocupadas, o processador decide, através de um algoritmo predeterminado, como tocar a nota nova. Em geral a voz que está tocando a nota mais antiga passa a ser ocupada pela nova nota, mas pode-se encontrar instrumentos que adotem procedimento diferente.

Dentro do padrão definido na MIDI Specification 1.0, há quatro modos possíveis de se operar as vozes de um instrumento, conforme mostram as a seguir.

Tabela 8 - Modos de Operação (recepção) modos de recepção para um instrumento ajustado para receber pelo Canal Básico N.	
MODO	OPERAÇÃO
Omni On / Poly	O instrumento processa todas as Mensagens de Voz, independente dos Canais de Voz em que elas estão sendo transmitidas, e responde a elas polifonicamente.
Omni On / Mono	O instrumento processa todas as Mensagens de Voz, independente dos Canais de Voz em que elas estão sendo transmitidas, e responde a elas monofonicamente.
Omni Off / Poly	O instrumento só processa as Mensagens de Voz que estão sendo enviadas no Canal de Voz "N", e responde a elas polifonicamente.
Omni Off / Mono	O instrumento só processa as Mensagens de Voz que estão sendo enviadas nos Canais de Voz "N" a "N+M-1", e responde a elas monofonicamente, alocando-as às vozes 1 a "M", respectivamente. O número de vozes "M" é definido no terceiro byte da Mensagem de Modo.

Tabela 9 - Modos de Operação (transmissão) modos de transmissão para um instrumento sintonizado no Canal Básico N.	
MODO	OPERAÇÃO
Omni On / Poly	O instrumento transmite todas as Mensagens de Voz através do Canal de Voz "N"
Omni On / Mono	O instrumento transmite todas as Mensagens de Voz, para uma voz, através do Canal de Voz "N"
Omni Off / Poly	O instrumento transmite todas as Mensagens de Voz, para todas as vozes, através do Canal de Voz "N".
Omni Off / Mono	O instrumento transmite as Mensagens de Voz das vozes 1 a "M" através dos Canais de Voz "N" a "N+M-1", respectivamente (uma voz por canal).

Um instrumento só pode operar em um único Modo de cada vez, e é recomendado que tanto transmissor quanto receptor(es) estejam operando no mesmo Modo. Caso um instrumento não possa operar no Modo requisitado, ele deve ignorar o comando de mudança de Modo de Operação, ou então utilizar um outro Modo alternativo (normalmente Modo 1).

As Mensagens de Modo só devem ser reconhecidas pelo receptor quando enviadas através do Canal Básico no qual ele está sintonizado, independentemente do seu Modo de Operação corrente. As Mensagens de Voz devem ser recebidas através dos Canais de Voz, conforme estabelecem as Tabelas 8 e 9, dependendo do Modo selecionado. O padrão permite, ainda, que um instrumento seja sintonizado em um ou mais Canais Básicos, por default ou por ação do usuário. Nesse caso, o instrumento se comporta como se fossem vários instrumentos independentes.

CONDIÇÕES INICIAIS

O padrão determina que o instrumento, ao ser ligado, adote as condições iniciais (default) de: Modo 1 de operação, recepção desabilitada de todas as Mensagens de Voz, exceto de Nota Ativada/Nota Desativada (Note On/Note Off) e supressão de quaisquer transmissões espúrias indefinidas.

OBSERVAÇÃO IMPORTANTE:

À época em que foi escrita MIDI Specification 1.0, o Modo 4 era esperado para transmissão/recepção de informações monofônicas através dos canais diferentes de MIDI, em um mesmo instrumento.

Posteriormente, os instrumentos multitimbrais passaram a operar de uma forma não definida em qualquer dos quatro modos, pois podem receber, simultaneamente, mensagens em vários canais diferentes com informações para execução polifônica em cada um desses canais, que são alocadas a uma ou mais vozes por canal. Esse quinto modo tem sido referenciado por muitos fabricantes como Modo Multi.

Tabela 10 - Códigos dos Fabricantes
atualizada até nov/87

Código	Fabricante	Código	Fabricante
00-00-14	Perfect Fretwork	21H	SIEL
01H	Sequential Circuits	22H	Synthaxe
02H	IDP	23H	Stepp _
03H	Voyetra/OctavePlateau	24H	Hohner
04H	Moog	25H	Twister
05H	Passport Designs	26H	Solton
06H	Lexicon	27H	Jellinghaus MS
07H	Kurzwei	28H	Southworth Music
08H	Fender	29H	PPG
0AH	AKG Acoustics	2AH	JEN
0BH	Voyce Music	2BH	SSL
0CH	Waveframe Corp.	2CH	Audio Veritrieb
0DH	ADA	2FH	Elka
0EH	Garfield Electronics	30H	Dynacord
0FH	Ensoniq	40H	Kawai
10H	Oberheim	41H	Roland
11H	Apple Computer	42H	Korg
12H	Grey Matter Response	43H	Yamaha
14H	Palm Tree Instr.	44H	Casio
15H	JL Cooper	46H	Kamyia Studio
16H	Lowrey	47H	Akai
17H	Adams-Smith	48H	Japan Victor
18H	E-Mu Systems	49H	Mesosha
19H	Harmony Systems	7DH	Uso Não Comercial
1AH	ART	7EH	Não Tempo-Real
1BH	Baldwin	7FH	Tempo-Real
1CH	Eventide		
1DH	Inventronics		
1FH	Clarity		

Anexo III

Tonalidades

Meta Evento Armadura de Clave -> **FF 59 02** byte1 byte2

Tabela em Hexadecimal

Tonalidade	<u>byte1</u>	<u>byte2</u>
Do Maior	00	00
Lá menor	00	01
Sol Maior	01	00
Mi menor	01	01
Ré Maior	02	00
Si menor	02	01
Lá Maior	03	00
Fá# menor	03	01
Mi Maior	04	00
Dó# menor	04	01
Si Maior	05	00
Sol# menor	05	01
Fá# Maior	06	00
Ré# menor	06	01
Dó# Maior	07	00
Lá# menor	07	01
Fá Maior	FF	00
Ré menor	FF	01
Sib Maior	FE	00
Sol menor	FE	01
MibMaior	FD	00
Dó menor	FD	01
Láb Maior	FC	00
Fá menor	FC	01
RébMaior	FB	00
Sib menor	FB	01
Solb Maior	FA	00
Mib menor	FA	01
Dób Maior	F9	00
Láb menor	F9	01

Anexo IV

Tipos de Mensagens MIDI

Byte de STATUS			Byte de DADOS 1		Byte de DADOS 2		
VOZ	Dec	Hex	Dec	Signif	Dec	Signif	
Note Off	128-143	80-8F	0-127	Nota	0-127	Veloc.	
Note On	144-159	90-9F	0-127	Nota	0-127	Veloc.	
Poly Aftert.	160-175	A0-AF	0-127	Nota	0-127	Pressão	
Control Chge	176-191	B0-BF	0-121	Controle	0-127	Valor	
Pitch Bender	224-239	E0-EF	0-127	Pos LSB	0-127	Pos MSB	
Prog. Chge	192-207	C0-CF	0-127	Prog.			
Ch. Aftert.	208-223	D0-DF	0-127	Pressão			
MODO							
Local On/Off	176-191	B0-BF	122		0=Off; 127=On		
Mono Mode	176-191	B0-BF	126		0-16; Canais		
All Notes Off	176-191	B0-BF	123		0		
Omni Off	176-191	B0-BF	124		0		
Omni On	176-191	B0-BF	125		0		
Poly Mode	176-191	B0-BF	127		0		
EXCLUSIVAS							
Sys-Ex	240	F0	0-127				
EOX	247	F7					
COMUNS							
Song Pointer	242	F2	0-127	Pos. LSB	0-127	Pos. MSB	
Song Select	243	F3	0-127	Seq.			
MTC	241	F1					
Tune Request	246	F6					
TEMPO REAL							
MIDI Clock	248	F8					
Start	250	FA					
Continue	251	FB					
Stop	252	FC					
Active Sense	254	FE					
Reset	255	FF					

Meta Eventos em Hexadecimal

META EVENTO	Byte 1	Byte 2	Lenght	Bytes de Dados
TEXT	FF	01	x	x Bytes de dados
COPYRIGHT	FF	02	x	x Bytes de dados
TRACK NAME	FF	03	x	x Bytes de dados
NOME INSTRUM.	FF	04	x	x Bytes de dados
LYRICS	FF	05	x	x Bytes de dados
MARCA	FF	06	x	x Bytes de dados
PONTO DE CORTE	FF	07	x	x Bytes de dados
FIM DE TRACK	FF	2F	00	
SET TEMPO	FF	51	03	tt tt tt (em microsegundos)
SAMPTE OFFSET	FF	54	05	5 Bytes de dados
TIME SIGNATURE	FF	58	04	4 Bytes de dados
KEY SIGNATURE	FF	59	02	2 Bytes de dados
DATA SEQUENCE	FF	7F	x	x Bytes de dados

Onde x pode ser representado de 1 a 4 Bytes, dependendo da quantidade de Byte de dados.

Meta Eventos em Decimal

META EVENTO	Byte 1	Byte 2	Lenght	Bytes de Dados
TEXT	255	1	x	x Bytes de dados
COPYRIGHT	255	2	x	x Bytes de dados
TRACK NAME	255	3	x	x Bytes de dados
NOME INSTRUM.	255	4	x	x Bytes de dados
LYRICS	255	5	x	x Bytes de dados
MARCA	255	6	x	x Bytes de dados
PONTO DE CORTE	255	7	x	x Bytes de dados
FIM DE TRACK	255	47	0	
SET TEMPO	255	81	3	tt tt tt
SAMPTE OFFSET	255	84	5	5 Bytes de dados
TIME SIGNATURE	255	88	4	4 Bytes de dados
KEY SIGNATURE	255	89	2	2 Bytes de dados
DATA SEQUENCE	255	127	x	x Bytes de dados

Anexo V

Valores Máximo e Mínimo de cada contagem com 1, 2, 3 e 4 Bytes.

Contagem com 1 Byte:

- Deve possuir o bit mais significativo igual a 0, ou seja, deve ser menor que 128 (0xxx xxxx).

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0 -> valor mínimo.

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 = 127 (7F_H) -> valor máximo.

Contagem com 2 Bytes:

- Deve possuir o primeiro Byte com o bit mais significativo igual a 1 e o segundo Byte com o bit mais significativo igual a 0. Neste caso, o primeiro Byte deve ser maior que 127 (7F = 0111 1111) e o segundo deve ser menor que 128 (80=1000 0000).

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 128 (80_H) -> valor mínimo.

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 = 255 (FF_H) -> valor máximo.

} 1º Byte

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0 -> valor mínimo.

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 = 127 (7F_H) -> valor máximo.

} 2º Byte

Contagem com 3 Bytes:

- Deve possuir o **primeiro e o segundo** Bytes com o bit mais significativo igual a **1**, e o **terceiro**, com o bit mais significativo igual a **0**.

1 0 0 0 0 0 0 0	= 128 (80 _H) -> <u>valor mínimo.</u>	} 1º Byte
1 1 1 1 1 1 1 1	= 255 (FF _H) -> <u>valor máximo.</u>	
1 0 0 0 0 0 0 0	= 128 (80 _H) -> <u>valor mínimo.</u>	} 2º Byte
1 1 1 1 1 1 1 1	= 255 (FF _H) -> <u>valor máximo.</u>	
0 0 0 0 0 0 0 0	= 0 -> <u>valor mínimo.</u>	} 3º Byte
0 1 1 1 1 1 1 1	= 127 (7F _H) -> <u>valor máximo.</u>	

Contagem com 4 Bytes:

- Deve possuir o **primeiro, o segundo e o terceiro** Bytes com o bit mais significativo igual a **1**, e o **quarto**, com o bit mais significativo igual a **0**.

1 0 0 0 0 0 0 0	= 128 (80 _H) -> <u>valor mínimo.</u>	} 1º Byte
1 1 1 1 1 1 1 1	= 255 (FF _H) -> <u>valor máximo.</u>	
1 0 0 0 0 0 0 0	= 128 (80 _H) -> <u>valor mínimo.</u>	} 2º Byte
1 1 1 1 1 1 1 1	= 255 (FF _H) -> <u>valor máximo.</u>	
1 0 0 0 0 0 0 0	= 128 (80 _H) -> <u>valor mínimo.</u>	} 3º Byte
1 1 1 1 1 1 1 1	= 255 (FF _H) -> <u>valor máximo.</u>	
0 0 0 0 0 0 0 0	= 0 -> <u>valor mínimo.</u>	} 4º Byte
0 1 1 1 1 1 1 1	= 127 (7F _H) -> <u>valor máximo.</u>	

Anexo VI

Figuras Musicais.

Usamos 7 figuras que representam 7 valores.

Figura e Pausa	Código	Nome em Português	Nome em Inglês	Valor Proporcional
	1	Semibreve	Hole Note	1
	2	Mínima	Half Note	$\frac{1}{2}$
	4	Semínima	Quarter Note	$\frac{1}{4}$
	8	Colcheia	Eigth Note	$\frac{1}{8}$
	16	Semicolcheia	16th Note	$\frac{1}{16}$
	32	Fusa	32th Note	$\frac{1}{32}$
	64	Semifusa	64th Note	$\frac{1}{64}$

A semibreve é a figura de maior valor, da qual todas as outras são subdivisões. Então:

- a **semibreve** vale **1** (inteiro)
- a **mínima** vale **1/2** (metade) da semibreve
- a **semínima** vale metade da mínima ou **1/4** da semibreve
- a **colcheia** vale metade da semínima ou **1/8** da semibreve
- a **semicolcheia** vale metade da colcheia ou **1/16** da semibreve
- a **fusa** vale metade da semicolcheia ou **1/32** da semibreve
- a **semifusa** vale metade da fusa ou **1/64** da semibreve

Cada nota musical pode soar um certo intervalo de tempo, dependendo da caixa de ressonância do instrumento. Em uma notação musical, os tempos de cada nota são

múltiplos uns dos outros. Assim, pode-se apresentar esta relação conforme a figura abaixo:

Diagrama das figuras musicais e suas durações relativas:

- Semibreve = 1
- Mínima = $1/2$
- Semínima = $1/4$
- Colcheia = $1/8$
- Semicolcheia = $1/16$
- Fusa = $1/32$
- Semifusa = $1/64$

O diagrama ilustra a hierarquia das figuras musicais em sete linhas de pauta. Cada linha representa uma figura, com exemplos de sua notação e duração. Linhas de pontilhado diagonais conectam as figuras, mostrando que uma semibreve equivale a duas mínimas, uma mínima a duas semínimas, e assim por diante, até a semifusa.

Anexo VII

Polimorfismo, Instâncias e Sobrecarregamento

Polimorfismo

O que é polimorfismo e para que serve?

O problema!

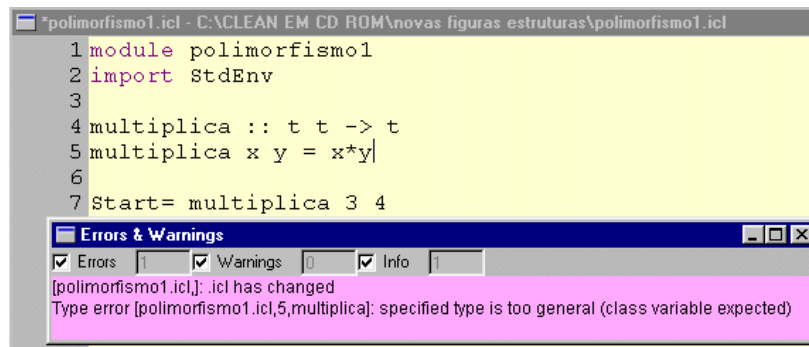
Imagine, inicialmente, que você deseja criar uma função de biblioteca a qual realiza a operação de multiplicação de dois argumentos os quais podem ser reais ou inteiros. Sabemos que para criar uma biblioteca devemos declarar o tipo da função e armazená-la em um módulo .dcl. O problema maior não é criar uma função que multiplique dois valores quaisquer. Já existe um operador que faz isto para você, o operador *. O problema é declarar um tipo para a função da biblioteca que aceite tanto inteiros quanto reais. Com a teoria até agora disponibilizada isto é impossível. Ou declaramos o domínio e contradomínio inteiro ou real.

A solução!

Clean disponibiliza uma solução para ao programador que é de deixar o mesmo criar um tipo genérico. Quando uma função possui uma ou mais variáveis como tipo, dizemos que ela é polimórfica.

Assim, vamos implementar uma função com tipo genérico e ver o que acontece.

Proceda da seguinte forma:



Inicialmente esperava-se que a declaração do tipo estivesse correta, ou seja:

-para qualquer tipo de dado *t* multiplicado por outro de mesmo tipo produzisse como resultado um dado de tipo *t*:

`multiplica :: t t -> t`

O problema ocorrido está definido na mensagem de erro, ou seja:

“O tipo especificado é muito geral (espera-se declarar uma classe para as variáveis”

Importante: O tipo *t* é denominado de tipo variável ou variável tipo. Estes tipos variáveis devem ser sempre escritos com letras minúsculas na declaração de tipo das funções.

No item seguinte veremos com detalhes o que são classes, por enquanto, para que se possa entender o exemplo, vamos defini-la de uma forma simplificada.

Quando o sistema pede para definir uma classe a ser aplicada aos argumentos da função, o mesmo está pedindo que você restrinja qual(is) função(ões) ou operação(ões) polimórfica se aplicam à função criada, de tal forma que o Clean possa rodar a função criada corretamente. Feito isto, o sistema passa a aceitar os mesmo tipos de dados que a classe escolhida aceitava. Neste caso, a operação (a classe) utilizada é a função multiplicar, utilizada pelo Clean através do símbolo ***.

Vamos modificar o exemplo para que o sistema rode:

```

polimorfismo1.icl - C:\CLEAN EM CD ROM\novas figuras
1 module polimorfismo1
2 import StdEnv
3
4 multiplica :: t t -> t | * t
5 multiplica x y = x*y

```

MS press any key to exit

Auto

12

O que foi feito é acrescentar a classe na declaração de tipo, ou seja:

Basta acrescentar o sinal de barra, a classe (no caso a operação `*`) e a variável trabalhada pela classe, como segue:

`multiplica :: t t -> t | * t`

O Programa fica:

```

polimorfismo1.icl - C:\CLEAN EM CD ROM\novas figuras
1 module polimorfismo1
2 import StdEnv
3
4 multiplica :: t t -> t | * t
5 multiplica x y = x*y

```

MS press any key to exit

Auto

12

Podemos acrescentar mais operações e definir a função **`media`**.

```

polimorfismo2.icl - C:\CLEAN EM CD ROM\novas figuras
1 module polimorfismo2|
2 import StdEnv
3
4 media::m m m-> m | +,/ m
5 media x y z= (x+y)/z
6
7 Start= media 4 6 2

```

MS press any key to exit

Auto

5

Neste caso, acrescentamos a operação de dividir, e, portanto, tivemos que acrescentar a operação (a classe) de divisão na declaração de tipo da função e a classe de soma.

Uma das funções que existe e que está relacionada com listas é a função *length*. Nos exemplos mostrados anteriormente a função *length* pode ser usada para calcular o tamanho de uma lista de inteiros assim como para calcular o tamanho de uma lista de reais. Isto nos conduz a seguinte pergunta: como é possível que uma função seja definida para uma lista que possa ser de diferentes tipos? Se olharmos a definição de *length* temos que:

$$\text{length} :: [a] \rightarrow \text{Int}$$

O tipo especificado para a lista a ser calculado o tamanho, $[a]$, indica que o tipo concreto dos elementos da lista não é estabelecido. Isto é possível pelo uso de uma *variável tipo* que é a letra *a* descrita entre os colchetes. Diferente de tipos concretos como *Int* ou *Real*, uma variável tipo é especificada por uma letra minúscula. Vejamos um outro exemplo: a função *hd* que devolve a cabeça de uma lista. Sua definição de tipo é a seguinte:

$$\text{hd} :: [a] \rightarrow a$$

Assim como ocorre com a função *length*, a função *hd* opera com listas de qualquer tipo. Observe que o resultado deve pertencer ao mesmo tipo dos elementos que compõem a lista dada como argumento. Assim sendo, se queremos a cabeça de uma lista de inteiros, o resultado deve ser um inteiro; se quisermos a cabeça de uma lista de reais, o resultado deve ser um real.

Um tipo que contém variáveis tipo é denominado de *tipo polimórfico*. As funções com um tipo polimórfico são chamadas de funções polimórficas. A linguagem Clean apresenta uma série de funções polimórficas, como é possível perceber pelas definições de seus tipos em suas bibliotecas.

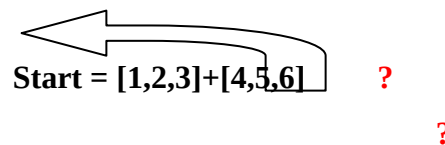
Instâncias e Sobrecarregamento:

O que vem a ser uma instância?

Criar uma instância ou realizar um sobrecarregamento possui o mesmo significado. O próprio nome já diz, ou seja, permitir que um operador que já existente, como por exemplo o operador de soma (+) ou subtração (-), possa ser aplicado a tipos de dados cujos operadores não foram instanciados. Um exemplo é instanciar o operador de soma para somar duas listas. Neste caso, devemos dizer à linguagem, no caso CLEAN, como o operador deverá se comportar e que tipos de dados as listas poderão possuir. O exemplo a seguir torna este conceito mais claro.

Ex.: Se você tentar realizar a seguinte operação no CLEAN:

Start = [1,2,3]+[4,5,6] ?
?



O que você espera obter como resultado desta operação?

- 1- Devolver uma lista com o mesmo número de elementos, de tal forma que cada um seja a soma dos elementos de mesma ordem?

$$[1,2,3] + [4,5,6] = [1+4,2+5,3+6] = [5,7,9]$$

- 2- Devolver uma lista com um elemento cujo valor seja a soma dos elementos das duas listas?

$$[1,2,3] + [4,5,6] = [21] \text{ ?}$$

A resposta é:

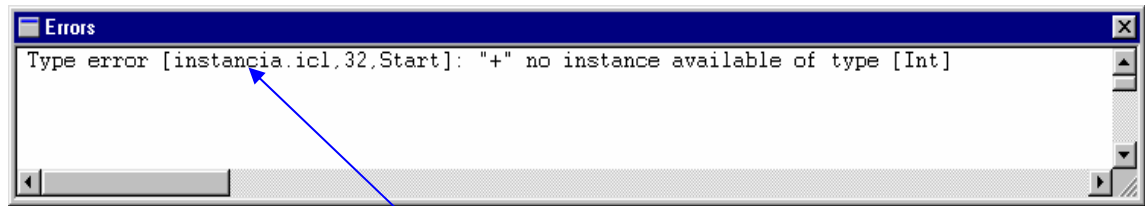
- Depende de como você instanciará o CLEAN para realizar a operação soma de listas.

Se você executar o seguinte comando antes de instanciar o operador + para listas, ocorrerá o seguinte erro:

Digite no ambiente do CLEAN:

Start = [1,2,3]+[4,5,6]

O CLEAN devolverá:



Ou seja, houve um erro de instância no comando Start, onde o operador “+” não está avaliado para o tipo lista de inteiros ([Int])

Para criar uma instância, uma sobrecarga de operadores, basta escrever a palavra reservada **instance** seguida da operação ou função que desejamos instanciar, seguido do tipo de dados que utilizaremos.

Atenção: Um ponto importante, antes de tentar criar uma instância, é saber que o resultado de uma operação de uma instância criada por você deverá ser do mesmo tipo de dados manipulados por ela.

Criando uma instância

Formato: **instance op tipo**

Onde:

. **op** é o operador ou função

. **tipo** é o tipo de dados que será manipulado pela operador instanciado.

Assim, para instanciarmos o operador de soma (+) para trabalhar com listas, deve-se proceder da seguinte forma:

Digite o CLEAN após o devido cabeçalho:

module instancia102000

import StdEnv

instance + [Int] //operador + aplicado a listas de inteiros

where

(+) a b = [sum(a ++ b)]

Start = [1,2] + [3,4] + [3,3]

O CLEAN devolve:

[16]

Explicação:

A soma de duas listas, da forma com que o operador foi instanciado, é uma lista com um elemento cujo valor é a soma dos elementos de duas listas de inteiros.

Conclusão: Criar uma instância nos permite utilizar funções e operadores já existentes em tipos de dados que não os suportam.

Vejamos mais um exemplo: **Somar dois números de tipos diferentes, como, por exemplo, Reais com Inteiros.**

Antes de criarmos esta instância, vejamos sua necessidade.

Vamos inicialmente somar dois números de mesmo tipo em CLEAN :

1- Digitando no ambiente do CLEAN

Start = 2+1

O mesmo devolve

3

2- Se digitarmos:

Start = 2.0 + 1.0

O mesmo devolve

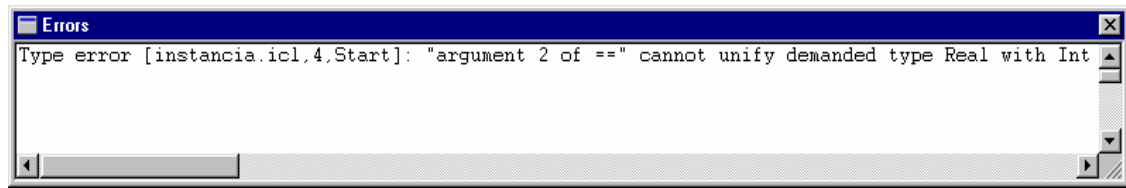
3.0

Agora vamos somar tipos diferentes

Se digitarmos

Start = 2.0 + 1

O mesmo devolve a seguinte mensagem de erro



Ou seja, o operador de soma não está instanciado para trabalhar com estes tipos mistos de dados.

Mas, e se quisermos realizar esta operação?

A resposta é: Não tem jeito!

Mas, então pra que perder tanto tempo neste exemplo?

A resposta é: Para que você tenha conhecimento que, se existir duas formas para o CLEAN tratar funções com tipos de dados iguais ou diferentes, a operação que será utilizada é a implementada na linguagem originalmente fornecida, ou seja, ela têm precedência sobre qualquer instância criada⁶⁶.

Mesmo assim, você pode criar um artifício que lhe permita somar tipos de dados diferentes, como por exemplo: Reais e Inteiros.

Vamos, portanto, fazer uma primeira tentativa de criar uma instância que informe ao computador como somar inteiros com reais.

Em primeiro lugar deve-se criar um tipo algébrico de dados que aceite inteiros e/ou reais.

Ex.:

:: Real_Eou_Inteiro = Int | Real

Assim, a uma primeira vista, estaríamos tentados em acreditar ter criado um tipo de dado que aceite inteiros e reais ao mesmo. Com este pensamento, uma lista do tipo [1, 2, 3.0, 4, 5.0] deveria ser entendida pelo CLEAN como uma lista com tipos Real_Eou_Inteiro.

Para confirmar se obteve sucesso, digite no ambiente do CLEAN.

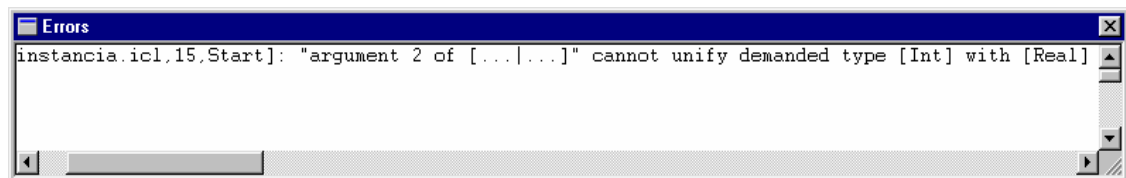
⁶⁶ A aplicação, portanto, de instâncias, é em situações não conflitantes com outras semelhantes às já existentes na linguagem.

Portanto, se executássemos:

```
module RealOuInteiro  
import StdEnv  
  
:: Real_Eou_Inteiro = Int | Real  
  
Start = hd [2,3.0]
```

Esperaríamos ter com resposta: **2**, que é a cabeça (hd) da lista **[2,3.0]**.

Em vez disso, o CLEAN devolve a seguinte mensagem de erro:



Ou seja, o CLEAN não consegue unificar o tipo Inteiro com o tipo Real.

Ora, não foi exatamente isto que pensamos ter acabado de criar?

O que ocorre é que ao criarmos o tipo **Real_Eou_Inteiro**, criamos uma situação que causaria uma dualidade (ambigüidade) de escolha ao compilador do CLEAN, ou seja, duas formas diferentes para executar uma mesma operação. O CLEAN encontra a função **hd [2,3.0]** e precisa decidir se aplica a função **hd** em uma lista de elementos cujo tipo é **Real_Eou_Inteiro** ou aplica **hd** em uma lista cujos elementos possuem tipos de dados diferentes. Nestes casos, os tipos pré-definidos pelo CLEAN têm precedência na escolha. Assim, o CLEAN interrompe o processamento e indica ao usuário o erro ocorrido.

- **Como solucionar, portanto, este caso?**
- **Como somar inteiros com reais?**

A solução é criar um tipo de dados que não cause ambigüidade, ou seja:

```
:: Real_Eou_Inteiro = INTEIRO Int | REAL Real
```

Colocando uma palavra (construtor de dados) antes do tipo que será utilizado, elimina-se a ambigüidade.

Agora, se executarmos:

```
module RealOuInteiro  
import StdEnv  
:: Real_Eou_Inteiro = INTEIRO Int | REAL Real
```

```
Start = hd [INTEIRO 2, REAL 3.0]
```

Teremos com resposta

```
(INTEIRO 2)
```

Vamos, agora, instanciar novos operadores para trabalhar com este tipo de dados.

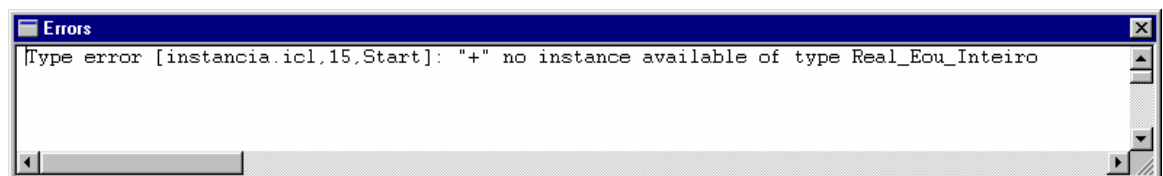
Se tentarmos agora executar

```
Start = (INTEIRO 2) + (REAL 3.5)
```

Gostaríamos que o CLEAN desse uma resposta do tipo

```
5.5
```

Mas, o que acontece é que o mesmo emitirá a seguinte mensagem de erro:



Ou seja: Não existe nenhuma instância que ensine o computador a aplicar o operador + no tipo Real_Eou_Inteiro.

Sendo assim, devemos agora “ensinar” ao sistema como este operador realizará a soma entre dois elementos cujo tipo de dados é o Real_Eou_Inteiro. Para que isto seja possível, devemos observar o seguinte:

- 1- Não podemos esquecer todas as combinações possíveis de soma que tal tipo permite.
- 2- Não podemos esquecer, também, qual o tipo de resposta que desejamos. Assim, vamos adotar que, ao somarmos dois elementos do tipo `Real_Eou_Inteiro`, e os mesmos forem do tipo **INTEIRO x**, daremos como resposta **INTEIRO z**. Se somarmos inteiros com reais, deveremos retornar **REAL x**. Se somarmos real com real, o sistema deverá retornar **REAL z**.

Desta forma, crie um arquivo **instanciar.icl** e digite o programa a seguir:

```
Module instanciar
```

```
Import StdEnv
```

```
::Real_Eou_Inteiro = INTEIRO Int | REAL Real
```

```
instance + Real_Eou_Inteiro
```

```
where
```

```
(+) (REAL x) (REAL y) = REAL (x+y) // real com real = real
```

```
(+) (INTEIRO x) (INTEIRO y) = INTEIRO (x+y) // inteiro com inteiro =
```

```
inteiro
```

```
(+) (INTEIRO x) (REAL y) = REAL ((toReal x)+y) // inteiro com real =
```

```
real
```

```
(+) (REAL y) (INTEIRO x) = REAL ((toReal x)+y) // real com inteiro =
```

```
real
```

Feito isso, se executarmos:

```
Start = (REAL 3.0) + (INTEIRO 2) + (REAL 3.5)
```

Teremos como resposta

```
(REAL 8.5)
```

Se quisermos eliminar a palavra **REAL** da resposta, basta criar uma função que faça isto, ou seja:

```
EliminaREAL (REAL x) = x
```

Se executarmos agora:

Start = EliminaREAL((REAL 3.0) + (INTEIRO 2) + (REAL 3.5))

Teremos como resposta:

8.5

Assim, esta instância criada nos permite somar inteiros com reais.

Vamos aproveitar este exemplo e criar novas instâncias para o tipo Real_Eou_Inteiro

Criando instâncias para comparar elementos do tipo Real_Eou_Inteiro:

```
instance == Real_Eou_Inteiro
  where
    (==) (REAL x) (REAL y) | x == y = True
    (==) (REAL x) (REAL y) | x <> y = False
    (==) (INTEIRO x) (INTEIRO y) | x == y = True
    (==) (INTEIRO x) (INTEIRO y) | x <> y = False
    (==) (INTEIRO x) (REAL y) | (toReal x) == y = True
    (==) (INTEIRO x) (REAL y) | (toReal x) <> y = False
    (==) (REAL x) (INTEIRO y) | (toReal y) == x = True
    (==) (REAL x) (INTEIRO y) | (toReal y) <> x = False
```

Podemos ver que criar instâncias é um recurso muito útil e fácil de ser realizado.

A seguir, mostramos alguns exemplos de instâncias criadas pelos programadores do CLEAN

SOMA DE COMPLEXOS:

```
instance + (!Real, !Real)
  where
```

```
(+) (x1,y1) (x2, y2) = (x1+x2, y1+y2)
```

SUBTRAÇÃO DE COMPLEXOS

```
instance - (!Real, !Real)
```

```
where
```

```
(-) (x1,y1) (x2, y2) = (x1-x2, y1-y2)
```

DIVISÃO DE COMPLEXOS

```
instance / (!Real, !Real)
```

```
where
```

```
(/) (x1, y1) ( x2, y2) = ((x1*x2+y1*y2)/d,(x2*y1-x1*y2)/d)
```

```
where
```

```
d= (x2*x2+y2*y2)
```

MULTIPLICAÇÃO DE COMPLEXOS

```
instance * (!Real, !Real)
```

```
where
```

```
(*) (x1, y1) (x2, y2) = (x1*x2-y1*y2, x2*y1+x1*y2)
```

Sobrecarregamento

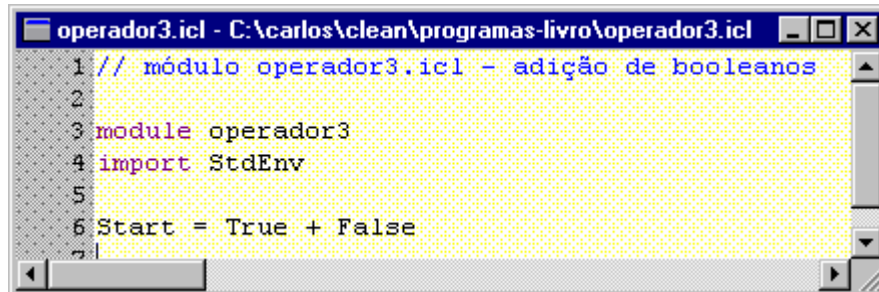
Você já deve ter observado que o operador + pode ser usado tanto para soma de valores do tipo Int quanto para valores do tipo Real. Portanto, o tipo do operador + pode ser:

- Int Int -> Int
- Real Real -> Real

Poderíamos inicialmente pensar que + pudesse ser uma função polimórfica definida da seguinte forma:

```
a a -> a
```

Se isto de fato fosse verdade poderíamos usar + para valores do tipo Bool. Entretanto, isto não é possível. Tente executar o programa que segue.



```
1 // módulo operador3.icl - adição de booleanos
2
3 module operador3
4 import StdEnv
5
6 Start = True + False
7
```

A execução do programa gerará a seguinte mensagem de erro:

"+" no instance available of type Bool

O erro decorre do fato de que a operação + não se encontra definida para operandos do tipo Bool.

Isto nos conduz a seguinte pergunta: qual é então o tipo do operador +?

Na realidade o que existe não é uma única definição para o operador +. Existem várias definições para o operador +. Uma delas é para números inteiros, outra para números reais e podem ainda existir muitas outras. Uma função ou operador que apresenta várias definições é chamada *sobrecarregada*.

Anexo VIII

Benchmark

Endereço <http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=all>

Gentoo : Intel® Pentium® 4 Computer Language Shootout

- all benchmarks - - all languages -

Create your own Overall Scores
 What fun! **April Fool's Day** all year long! Can you manipulate the multipliers and weights to make the fastest programming language in the Shootout?
 And remember, languages with more × non-scoring benchmarks (No Program, Error, Timeout) will be penalized.
 And remember, **"For every complex problem, there is a solution that is simple, neat, and fast."**

ratio	language	score ×	benchmark	weight
	best possible	100.0		
1.0	C gcc	76.6 1	Full CPU Time	1
1.0	C++ g++	75.8	Memory Use	0
1.1	D Digital Mars	70.3	GZip Bytes	0
1.3	Clean	57.0 3		
1.4	Pascal Free Pascal	56.4 2		
1.4	Eiffel SmartEiffel	53.1 3	binary-trees	1
1.6	OCaml	49.1 2	chameneos	1
1.6	Ada 95 GNAT	47.1 2	cheap-concurrency	1
1.7	Haskell GHC	46.1	fannkuch	1
1.7	SML MLton	45.3 2	fasta	1
1.7	Scala	44.9 1	k-nucleotide	1
1.7	Java JDK -server	43.9	mandelbrot	1
1.7	BASIC FreeBASIC	43.9 2	meteor-contest (new)	0
1.9	Lisp SBCL	41.3 5	n-body	1
2.0	Oberon-2 OO2C	39.2 7	nsieve	1
2.1	Nice	36.4 4	nsieve-bits	1
2.2	Forth bigForth	34.3 1	partial-sums	1
2.4	C# Mono	31.4 2	pidigits	1
2.9	Fortran G95	26.8 6	recursive	1
4.2	Erlang HiPE	18.3 2	regex-dna	1
4.9	Smalltalk VisualWorks	15.5 1	reverse-complement	1
5.3	Mozart/Oz	14.4 2	spectral-norm	1
6.4	Scheme MzScheme	11.9 4	startup	0
6.7	Python	11.5	sum-file	1
7.3	Lua	10.5		
7.5	PHP	10.2 3		
8.6	Pike	8.9 3		
9.4	Tcl	8.2 3		
9.5	Perl	8.1 2		
13	Icon	5.9 7		
16	Ruby	4.7 1		
34	JavaScript SpiderMonkey	2.3 7		

about the Overall Scores**The Basic Score**

For each benchmark, the *best measurement* **B** is the lowest non-zero measurement.

For each language implementation, the measurement **L** for the language implementation is converted to a basic score:

$$\text{basic score} = B/L \times 100$$

The highest possible unweighted score is 100. Missing programs and programs which did not complete the benchmark score 0.

The Mean Score

The basic score can be adjusted by a measurement multiplier (make CPU time or memory use or lines-of-code more or less important) and by a benchmark weight (make particular benchmarks more or less important).

For each language implementation, the weighted scores for every benchmark are summed. The CPU time score, memory use score, and lines-of-code score are summed to give a total score.

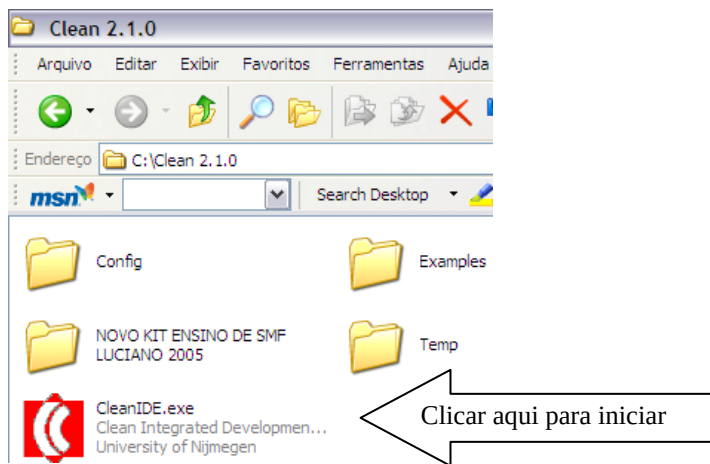
The total score is divided by the number of non-zero weighted benchmarks to give the mean score.

Anexo IX

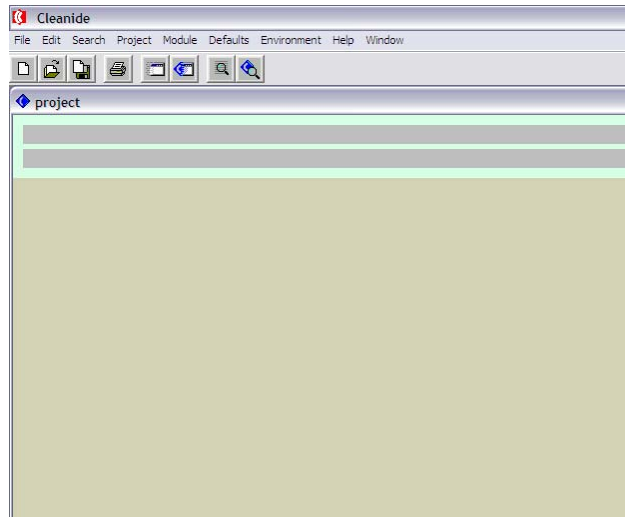
Utilizando Clean pela primeira vez

Para utilizar o Clean pela primeira vez, siga os passos a seguir:

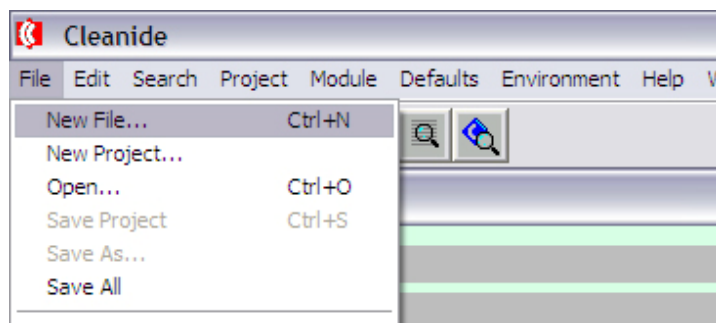
- 1- Copiar o diretório do CLEAN para a raiz de um de seus HDs. No caso deste exemplo, será utilizado o disco C.
- 2- Entre no diretório do CLEAN, dê dois cliques de mouse no ícone CleanIDE.exe



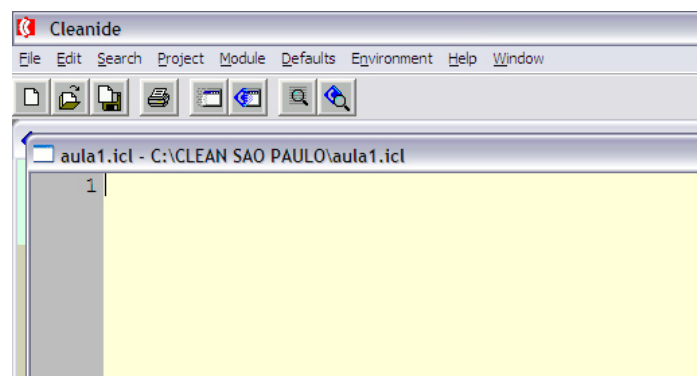
- 3- Ao fazer isto, o ambiente do CLEAN é aberto, conforme figura a seguir:



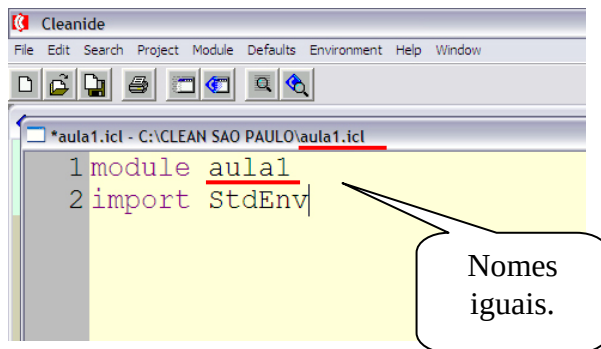
- 4- Inicie um arquivo de trabalho novo (.icl). Para tanto, ative o menu File e clique na opção New File, conforme figura a seguir:



- 5- Escolha um diretório para criar o arquivo. Digite o nome do arquivo com o sufixo “.icl”. Todas as letras “icl” devem ficar em minúsculas. Evite espaço entre letras.
- 6- A interface com o ambiente do CLEAN é aberta, conforme mostra a figura a seguir



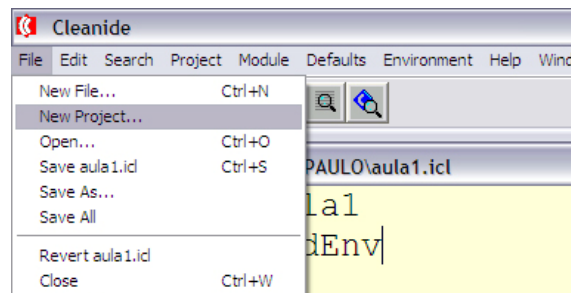
- 7- Feito isto, deve-se digitar o cabeçalho padrão de um módulo de programa em CLEAN, conforme a figura a seguir:



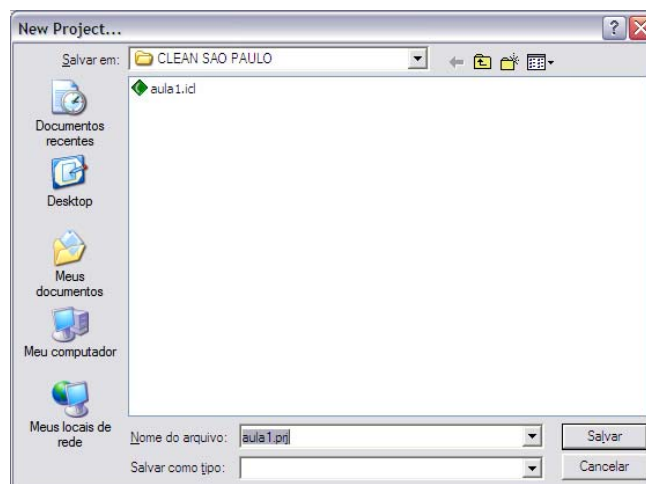
Observe que o nome do módulo tem que ser o mesmo nome do arquivo. No caso, o arquivo é “**aula1.icl**” e o nome do módulo é “**aula1**” (sem o sufixo .icl).

- 8- O próximo passo é criar um novo projeto. Para tanto, deve-se clicar no menu File, e clicar na opção New Project.

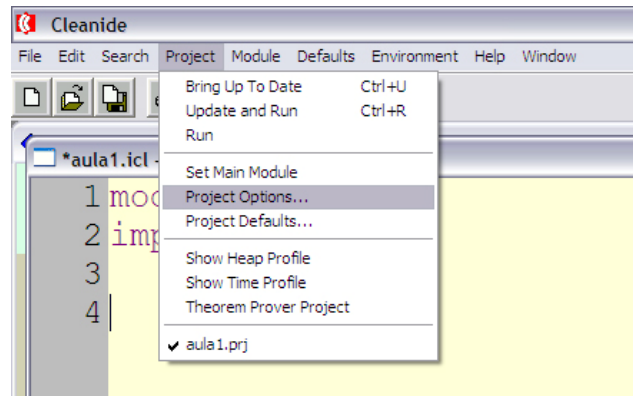
Conforme mostra a figura a seguir.



- 9- Ao fazer isto, a janela do gerenciador de arquivos é aberta. Para criar o projeto basta clicar em “Salvar”. O CLEAN se encarrega de nomear o arquivo de projeto como **aula1.prj**.

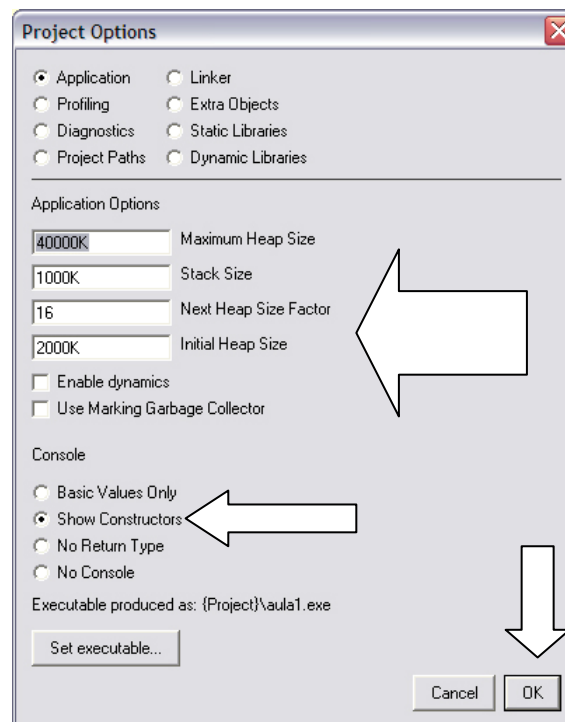


- 10- Voltando à interface do ambiente, configure o projeto, clicando no menu **Project** e clicando na opção *Project Options*.

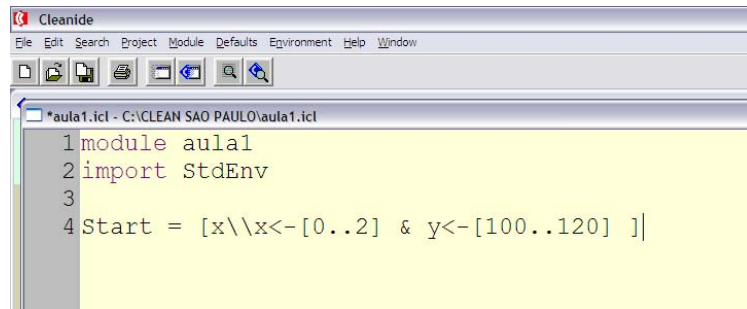


- 11- Ao fazer isto, uma nova janela é aberta. Configure os campos de texto da mesma, conforme a figura a seguir. Ou seja:

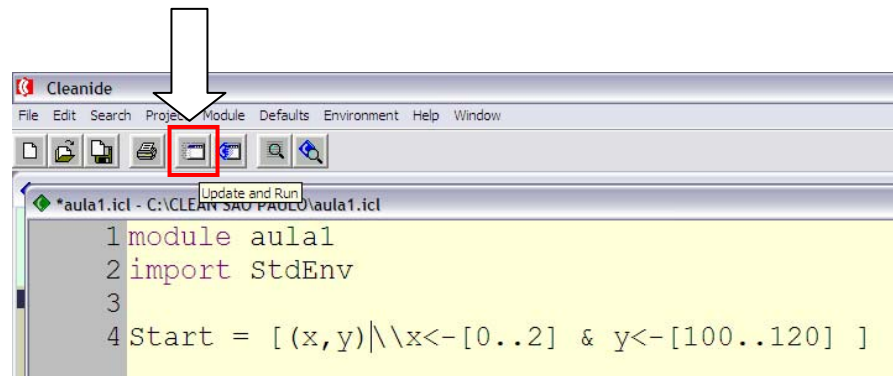
- Trabalhar com a console para saída de dados: **ShowConstructors** ativado. Caso se fosse implementar interfaces visuais, deve-se marcar a opção **No Console**.
- *Heap* e *stack* com valores suficientes para manipular arquivos grandes (optou-se por 40MBytes)



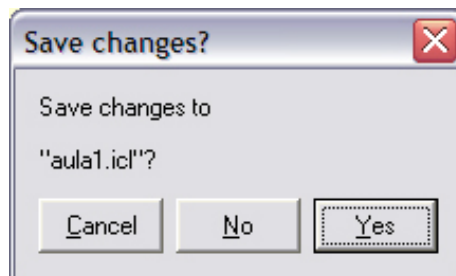
- 12- Voltando à interface do ambiente do Clean, digite o exemplo, conforme a figura a seguir.



- 13- Para rodar um programa em Clean, deve-se ter uma chamada com a função **Start**. Ao digitar o que se quer, você deverá salvar (atualizar) e rodar o programa (*Update and Run*). Para se fazer isto, existem várias formas. Uma delas é clicar no ícone mostrado na figura a seguir. Outra é clicar no menu *Project* e clicar na opção *Update and Run*.

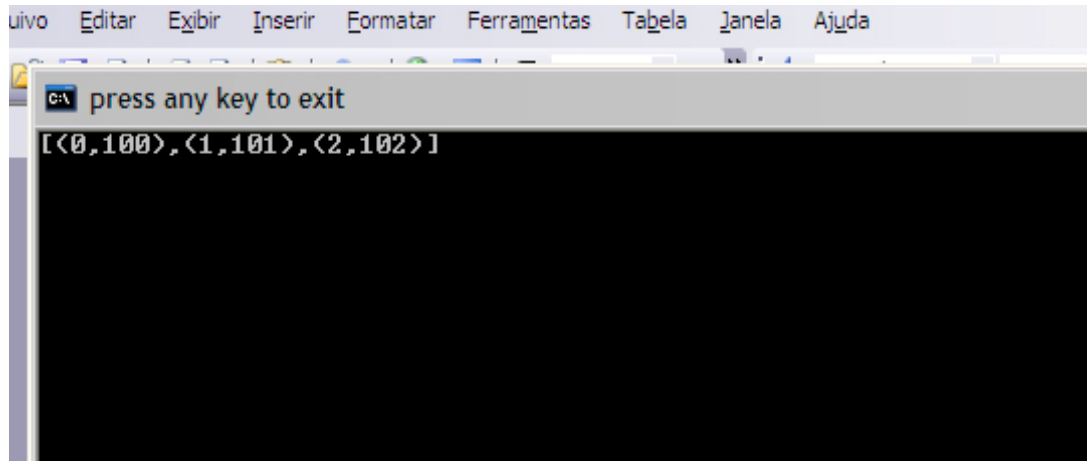


- 14- Ao fazer isto, uma janela de diálogo é aberta. Clique em Yes, ou seja, você deseja atualizar (*update*) e rodar (*Run*) o programa.

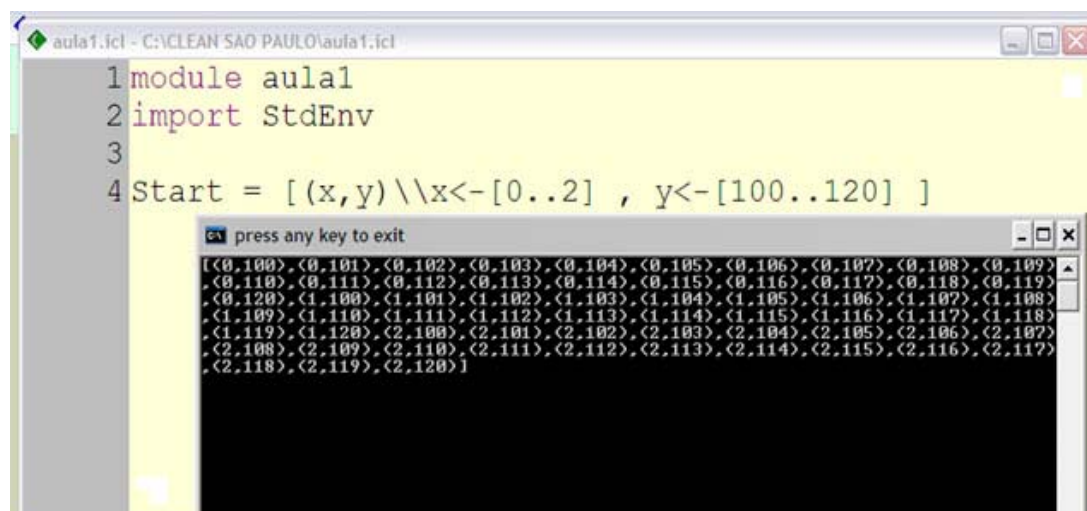


- 15- O programa é executado. Se tudo der certo, uma janela preta (console) será aberta apresentando o resultado do que se pediu, ou seja, uma lista contendo uma tupla de dois elementos (**x,y**), cujo primeiro elemento pertence à lista dos inteiros de 0 a 2 (**x<-[0..2]**) e o segundo elemento pertence à lista dos elementos

de 100 a 120 ($y < -[100..120]$). A tupla será formada com o primeiro elemento da primeira lista com o primeiro elemento da segunda lista, e assim por diante, conforme mostrado na figura abaixo. Quando uma das listas não tiver mais elementos, o processo é terminado.



16- Para gerar uma lista com todas as combinações, basta trocar o gerador “&” pelo gerador “,” (vírgula), conforme mostrado na figura a seguir.



Programando com a técnica de Tipos Únicos

Um dos principais conceitos necessários à implementação de aplicativos é saber como abrir e salvar arquivos. Para tanto, deve-se utilizar funções da biblioteca *StdIO* do Clean.

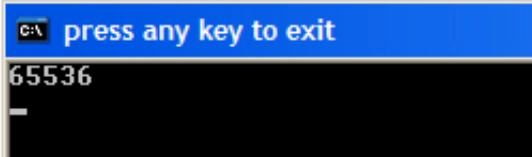
Manipular arquivos e interfaces visuais é tarefa que não possui transparência referencial e que necessita realizar avaliações destrutivas em dados, memória e arquivos. Para tanto, o Clean utiliza a técnica de **Tipos Únicos** para evitar efeitos colaterais

indesejados derivados destas duas características comuns em linguagens procedurais. Nesta técnica, Tipos Únicos, a linguagem, antes de começar a trabalhar com arquivos e processos assume o *status* do computador com seu conteúdo e passa a monitorar todas as modificações de IO que ocorrem no sistema, impedindo que mais de um processo ou função manipule o mesmo dado. Somente após o dado, arquivo ou qualquer outro elemento global ser liberado pela função ou processo que o estava utilizando é que o mesmo, alterado ou não, poderá ser manipulado por outro processo, por outra função. Assim, uma vez referenciados, um registro, uma variável, memória, arquivos, etc., os mesmos passam a ficar indisponíveis.

Desta forma, para iniciar um processo que utiliza **IO** (entrada e saída de dados), a sintaxe dos programas muda radicalmente. A função **Start** passa a ter um argumento, denominado de “mundo”, ou seja, uma variável de controle das unidades de **IO** do computador.

Assim, um programa que envolve **IO** inicia e termina como segue:

```
1 module testeMundo
2 import StdEnv, StdIO
3
4
5
6 Start world
7
8
9 =world|
10
```

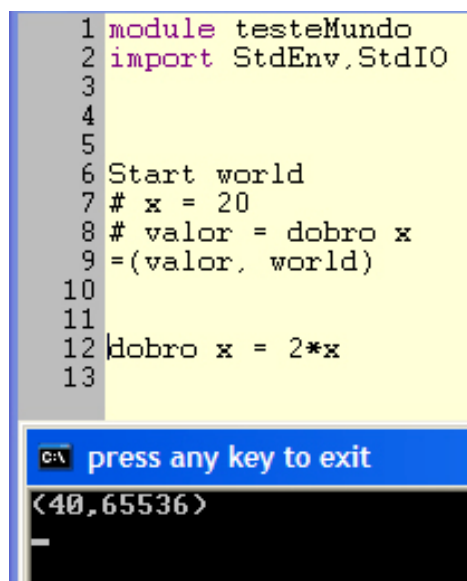


Obs. A resposta devolve o valor 65536, que é o status atual de seu mundo, o qual, depois de “aberto” foi retornado sem nenhuma modificação em suas unidades de **IO**.

Caso neste processo fosse realizada uma ação de leitura de um arquivo qualquer, este arquivo seria monitorado por esta variável, alterando seu valor para um valor que traduza, para o compilador da linguagem, esta ocorrência. Desta forma, a variável **world** inicial seria alterada. Assim, como a variável de entrada e a de saída possuem o mesmo nome e conteúdos diferentes, ocorreria, aí, uma avaliação destrutiva, mas controlada.

Entre iniciar a função **Start** e seu fim, cada linha de programa deve ser precedida do símbolo **#**⁶⁷. O mesmo significa Let (lê-se rash). O programa, com a técnica de Tipos Únicos, passa a tomar algumas características procedurais, tais como:

- O programa é executado linha a linha, top->down. Desta forma, se uma linha utiliza uma função ou variável, as mesmas deverão ter sido criadas anteriormente, ou seja, em linhas precedentes.
- As variáveis e os resultados das funções passam a ser globais no escopo da função **Start**. As variáveis, o resultado fica do lado esquerdo da igualdade e a chamada da função do lado direito, como mostra o exemplo da figura abaixo.



```
1 module testeMundo
2 import StdEnv, StdIO
3
4
5
6 Start world
7 # x = 20
8 # valor = dobro x
9 =(valor, world)
10
11
12 dobro x = 2*x
13
```

C:\> press any key to exit

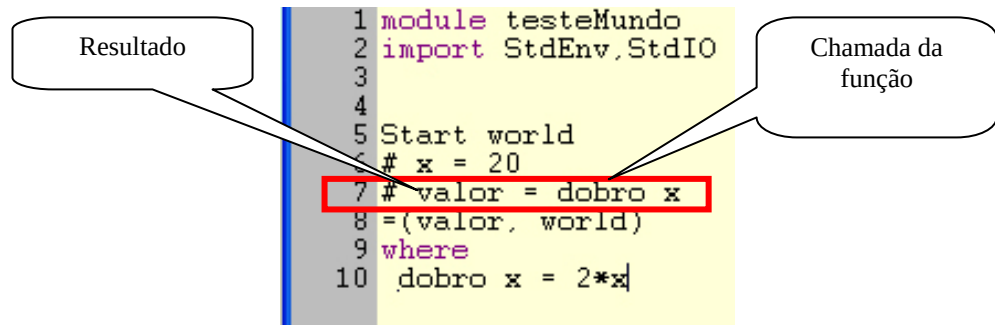
<40,65536>

Tipos únicos

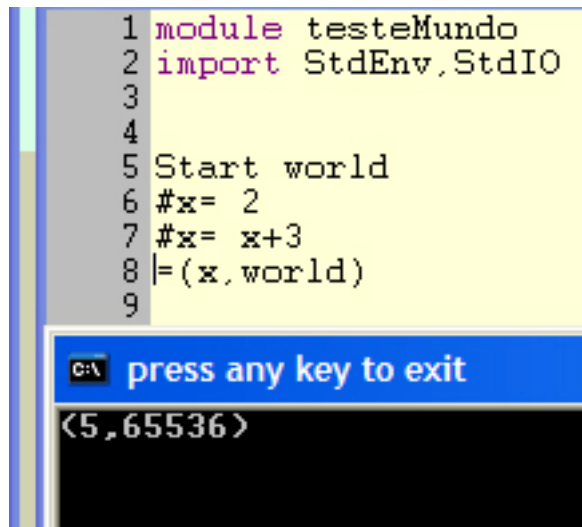
Observe que na resposta tem-se uma tupla com o resultado no primeiro elemento, e, no segundo elemento, o valor do mundo, que é o mesmo do exemplo anterior, já que, novamente, nenhuma operação de **IO** foi procedida.

Na técnica de tipos únicos, se o “mundo” foi “aberto”, deve-se, após realizar as ações desejadas, “devolver” o mundo atualizado. Assim, após a igualdade devolve-se o conteúdo de valor. Observe que a função dobro foi definida fora do corpo da função **Start**, de uma forma global, ou seja, podendo ser utilizada por outras funções. Poder-se-ia definir a função dobro localmente. Quem especifica as variáveis e funções globais é a palavra reservada **where**.

⁶⁷ Pode-se programar sem utilizar # em cada linha, mas, neste trabalho, adotar-se-á esta convenção, esta sintaxe.



- Pode-se realizar avaliações destrutivas de variáveis, tal como: $x = x + 3$.



Abrindo, manipulando e salvando Arquivos

Abrir e salvar arquivos de textos e de dados é uma tarefa imprescindível em interfaces multimídia. Para tanto, implementou-se uma biblioteca, a **StdArq**, a qual possui várias funções de manipulação e criação de arquivos e diretórios. A mesma é apresentada no CD em anexo, já locada na biblioteca **StdEnv** do diretório da Linguagem Clean 2.1 também neste mesmo CD. No corpo desta dissertação serão mostrados os conceitos de abrir e salvar arquivos de texto e de dados, como processar um texto extraindo os *tokens* do mesmo, como converter uma estrutura em formato texto em uma estrutura de listas para manipulação em Clean e seu processo inverso, como criar novos tipos algébricos de dados, como declarar tipos de dados mais complexos, como criar bibliotecas e utiliza-las em outros aplicativos, como programar utilizando a técnica de tipos únicos e outros conceitos relevantes para o entendimento das implementações multimídia e das interfaces visuais mostradas no próximo capítulo.

Leitura de Arquivo de Dados

Como a leitura de arquivos será feita, nos aplicativos, em uma interface visual, a mesma deverá ser implementada com a técnica de tipos únicos para ser utilizada localmente ou como biblioteca.

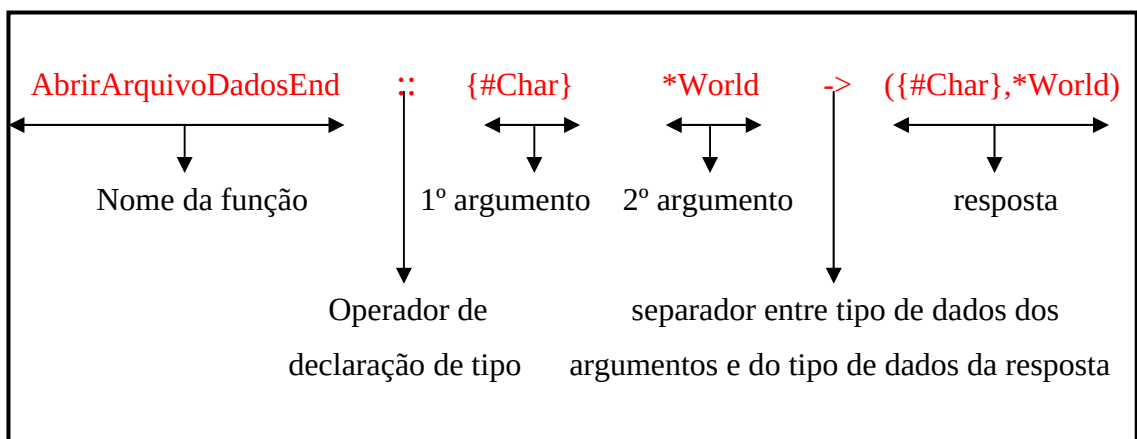
AbrirArquivoDadosEnd endFixo comp,

Para tornar o programa mais legível e poder colocar uma função em uma biblioteca, deve-se declarar o tipo da mesma.

Declarar o tipo de uma função, em Clean, possui a seguinte sintaxe:

nomeDaFuncao :: tipoDosDadosDosArgumentos -> tipoDeDadoDaResposta

o que, nesta função fica:



O que significa:

- Função com dois argumentos de entrada
 - Um do tipo String (**{#Char}**), o endereço;
 - Um do tipo ***World**, o status de **IO** do computador. O Asterisco indica que a variável do argumento **World** pode ser modificada.
- Resposta = Tupla com dois elementos:
 - Uma String (**{#Char}**) contendo o conteúdo do arquivo lido;
 - A variável do tipo **World** modificada pela ação de se ter feito uma operação de leitura em arquivo.

```
//função de ler arquivo dado endereço fixo
AbrirArquivoDadosEnd :: {#Char} *World -> ({#Char},*World)
AbrirArquivoDadosEnd endFixo comp
#(ok, file1, comp)= fopen endFixo FReadData comp
|ok == False = ("",comp)
#(conteudo1, file1)= freads file1 20000000
#(ok, comp)= fclose file1 comp
|ok == False == ("",comp)
=(conteudo1,comp)
```

Na função criada, tem-se:

- 1- Abertura de arquivo (**fopen**), dado um endereço (**endFixo**), para leitura de dados (**FReadData**). Como resposta, tem-se a confirmação de que o arquivo foi ou não aberto na variável **ok**. Esta variável devolve **True** se o arquivo foi aberto com sucesso e **False** se o arquivo não for encontrado pelo endereço fornecido. Caso a resposta seja **False**, a abertura é desconsiderada (**|ok == False = ("",comp)**).
- 2- Lê-se arquivo de até 20MBytes (**freads file1 20000000**) de dados. Caso o arquivo seja menor, o Clean desconsidera sem alocar memória extra. Como resultado desta ação, o conteúdo é devolvido como primeiro elemento da tupla de resposta (**(conteudo1, file1)**). Como a técnica de tipos únicos não permite que um mesmo arquivo possa ser manipulado por dois processos ou funções ao mesmo tempo, este arquivo só será liberado após ser fechado. Como já se tem uma cópia do mesmo para manipulação, e o intuito da função é apenas fazer isto, deve-se fechar o arquivo para liberá-lo (**fclose file1 comp**).
- 3- O resultado é retornado em forma de uma tupla com dois elementos, sendo o primeiro o conteúdo do arquivo (**(conteudo1,comp)**).

Caso se deseje ler o arquivo a partir do gerenciador de arquivos do Windows, ao invés de fornecer o endereço do arquivo, pode-se ler o mesmo pelo gerenciador. Para tanto, acrescenta-se duas linhas ao programa.

```
//função de abrir arquivo dados via explorer
AbrirArquivoDadosExpl :: *World -> ({#Char},{#Char},*World)
AbrirArquivoDadosExpl comp
#(nomeString, comp)= selectInputFile comp
|nomeString == Nothing = (""," ",comp)
#(ok, file1, comp)= fopen (fromJust nomeString) FReadData comp
|ok == False = (""," ",comp)
#(conteudo1, file1)= fread file1 200000
#(ok, comp)= fclose file1 comp
|ok == False = (""," ",comp)
=(conteudo1,fromJust nomeString,comp)
```

Onde:

- ***selectInputFile*** abre o gerenciador de arquivos.
- a segunda linha somente testa se o usuário cancelou a abertura do arquivo, caso tenha cancelado, a abertura é cancelada.
(***nomeString == Nothing = (""," ",comp)***).
- Na resposta, tem-se como segundo elemento da tupla: ***fromJust nomeString*** que retorna o endereço do arquivo. A função ***fromJust*** é utilizada porque, caso o arquivo seja aberto com sucesso, o endereço retorna um tipo com o construtor de dados ***just*** seguido do endereço, e, neste caso, deseja-se eliminar o construtor de dados para pegar apenas o endereço. A função ***fromJust*** faz isto.

Para ler arquivos de texto, basta trocar a função ***FReadData*** pela função ***FReadText***, conforme pode ser visto na biblioteca ***StdArq*** no CD em anexo.

A diferença fundamental em se utilizar funções de leitura ou escrita de dados em vez de texto é que na função de ler e escrever texto os caracteres especiais de texto são interpretados de forma diferente, como por exemplo, os caracteres 10 e 13 que são caracteres de saltar linha e retorno de carro, o que pode levar a um erro de leitura e/ou de gravação quando utilizados (o que é o caso de um arquivo MIDI, onde se utiliza todos os 256 códigos de um Byte).

Escrita em Arquivo de dados

Gravar arquivos é semelhante ao processo de se ler. A diferença está em se utilizar a função ***selectOutputFile*** para abrir o gerenciador de arquivos para gravar dados e utilizar a função ***FWriteData*** e ***fwrites*** para escrever o arquivo, no lugar das funções de leitura. A figura abaixo mostra as funções de gravar arquivo de dados via gerenciador de arquivos ou através de um endereço. As funções de gravar em arquivo texto podem ser vistas na biblioteca ***StdArq*** no CD em anexo.

```

//função de gravar arquivo dado endereço fixo
GravarArquivoDadosEnd :: {#Char} {#Char} *World -> *World
GravarArquivoDadosEnd endFixo conteudo comp
#(ok,file1, comp)= fopen endFixo FWriteData comp
|ok == False = comp
#file1 = fwrites conteudo file1
#(ok, comp)= fclose file1 comp
|ok == False = comp
=comp

|

//função de gravar arquivo texto endereço fixo
GravarArquivoTextoEnd :: {#Char} {#Char} *World -> *World
GravarArquivoTextoEnd endFixo conteudo comp
#(ok,file1, comp)= fopen endFixo FWriteText comp
|ok == False = comp
#file1 = fwrites conteudo file1
#(ok, comp)= fclose file1 comp
|ok == False = comp
=comp

```

Criando uma biblioteca

Para se criar uma biblioteca, necessita-se de dois arquivos:

1. Um arquivo, um módulo, contendo as implementações das funções (com extensão .icl);
2. Um arquivo, um módulo, contendo apenas as definições dos tipos de dados das funções que serão disponibilizadas na biblioteca (com extensão .dcl).

Conforme mostrado no item - “Utilizando Clean pela primeira vez”, um módulo de trabalho inicia pela palavra reservada **module** seguido pelo nome do módulo (o qual deve ser igual ao nome do arquivo), e, na linha seguinte, uma sequência de abertura de bibliotecas, iniciada pela palavra reservada **import**. Assim, a *figura abaixo* mostra o módulo contendo duas das funções de leitura e escrita em arquivo de dados para um arquivo nomeado **StdArq1**.

```

1 module StdArq1
2 import StdEnv, StdIO
3
4 AbrirArquivoDadosEnd :: {#Char} *World -> ({#Char},*World)
5 AbrirArquivoDadosEnd endFixo comp
6 #(ok, file1, comp)= fopen endFixo FReadData comp
7 |ok == False = ("",comp)
8 #(conteudo1, file1)= fread1 file1 20000000
9 #(ok, comp)= fclose file1 comp
10 |ok == False = ("",comp)
11 =(conteudo1,comp)
12
13 GravarArquivoDadosEnd :: {#Char} {#Char} *World -> *World
14 GravarArquivoDadosEnd endFixo conteudo comp
15 #(ok,file1, comp)= fopen endFixo FWriteData comp
16 |ok == False = comp
17 #file1 = fwrites conteudo file1
18 #(ok, comp)= fclose file1 comp
19 |ok == False = comp
20 =comp
21
22 Start comp
23 #(conteudo,comp)=AbrirArquivoDadosEnd "C:\\teste.mid" comp
24 =(conteudo,comp)
25

```

Função de teste para devolver o conteúdo de um arquivo de dados no endereço: "C:\\teste.mid".

Após testar todas as funções implementadas, para criar uma biblioteca, deve-se seguir os seguintes passos.

1. Acrescentar no cabeçalho do módulo a palavra reservada **implementation** antes da palavra **module**, ficando: **implementation module StdArq1**

```

1 implementation module StdArq1
2 import StdEnv, StdIO
3

```

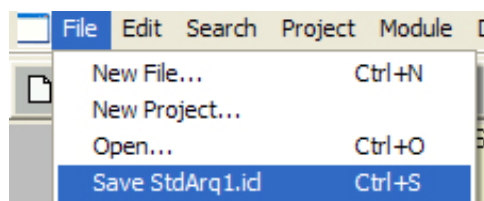
2. Retirar a chamada da função **Start** do módulo ou colocar como comentário. Comentário de bloco inicia por **/*** e finaliza por ***/**. Comentário de linha inicia por duas barras: **//**

```

23 /*
24 Start comp
25 #(conteudo,comp)= AbrirArquivoDadosEnd "C:\\teste.mid" comp
26 =(conteudo,comp)
27 */

```

3. Salvar o arquivo com o mesmo nome (**Save StdArq1.icl**), por cima do arquivo antigo.



4. O arquivo de implementação está pronto. Falta o arquivo de definição dos tipos. Para tanto, deve-se retirar o nome **implementation** e substituir por **definition**.

```
1 definition module StdArq1
2 import StdEnv, StdIO
```

5. Deve-se, agora, deixar no módulo apenas as declarações de tipo das funções que se deseja utilizar na biblioteca.

```
1 definition module StdArq1
2 import StdEnv, StdIO
3
4 AbrirArquivoDadosEnd :: {#Char} *World -> ({#Char},*World)
5
6 GravarArquivoDadosEnd :: {#Char} {#Char} *World -> ({#Char},*World)
7
```

6. Feito isto, basta salvar o arquivo de definição com a extensão **.dcl**.

Feito isto o arquivo fica cor de rosa indicando o sucesso da operação.

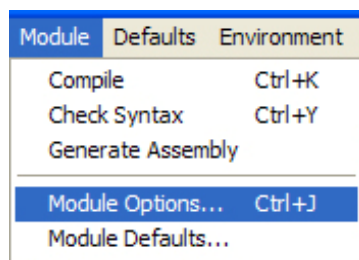
```
1 definition module StdArq1
2 import StdEnv, StdIO
3
4 AbrirArquivoDadosEnd :: {#Char} *World -> ({#Char},*World)
5
6 GravarArquivoDadosEnd :: {#Char} {#Char} *World -> ({#Char},*World)
7
8
9
```

Muitas vezes, para o iniciante em Clean, declarar tipos de funções é uma tarefa difícil, e, com isto, o mesmo fica impossibilitado de criar bibliotecas.

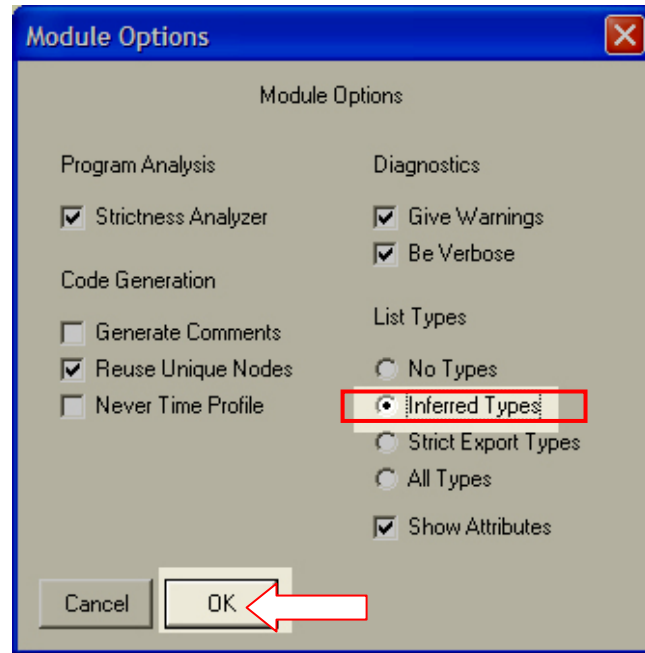
Para tanto, o Clean disponibiliza um recurso para o compilador mostrar os tipos de dados das funções implementadas.

Assim, no exemplo anterior, ao eliminar as declarações de tipo, pode-se configurar o compilador do Clean para mostrar os tipos das mesmas. Para tanto, deve-se seguir os seguintes passos:

- 1- Ativar o menu **Module** e clicar na opção **Module Options...**



- 2- Na janela aberta, ativar a opção **Inferred Types** e clicar em OK



- 3- Ao mandar compilar o programa, o compilador do Clean infere os tipos e apresenta em uma janela específica a este fim.

```

1 module StdArq1
2 import StdEnv, StdIO
3
4 AbrirArquivoDadosEnd endFixo comp
5 #(ok, file1, comp)= fopen endFixo FReadData comp
6 |ok == False = ("",comp)
7 #(conteudo1, file1)= freada file1 20000000
8 #(ok, comp)= fclose file1 comp
9 |ok == False = ("",comp)
10 =(conteudo1,comp)
11
12 GravarArquivoDadosEnd endFixo conteudo comp
13 #(ok,file1, comp)= fopen endFixo FWriteData comp
14 |ok == False = comp
15 #file1 = fwrites conteudo file1
16 #(ok, comp)= fclose file1 comp
17 |ok == False = comp
18 =comp
19
20 Start comp=comp
21

```

Types

```

"StdArq1"
AbrirArquivoDadosEnd :: {#.Char} *a -> *({#.Char},*a) | FileSystem a
GravarArquivoDadosEnd :: {#.Char} {#.Char} *a -> *a | FileSystem a
Start :: *World -> .World

```

Observe que as declarações de tipo são diferentes das declarações feitas nos exemplos. Isto se dá porque o Clean define os tipos de uma forma mais geral. Assim, basta copiar as declarações de tipo e acrescentá-las às funções para a criação das bibliotecas.

Separando *Tokens*⁶⁸ de uma *String* ou de um arquivo texto

⁶⁸ **Token** em computação é um segmento de texto ou símbolo que pode ser manipulado por um parser, que fornece um significado ao texto; em outras palavras, é um conjunto de caracteres (de um alfabeto, por exemplo) com um significado coletivo. <http://pt.wikipedia.org/wiki/Token>

Ao abrir um arquivo texto ou de dados, na maioria das vezes deseja-se separar os Tokens do mesmo, seguindo um critério ou expressão regular. Assim, baseado na linguagem MLWorks, implementou-se uma função que separa os tokens (*strings*) de um arquivo: a **StringTokens**. Para se separar um *token*, é necessário que se defina qual é o caractere de separação. A seguir, na figura abaixo é mostrada esta função, a qual, também, se encontra na biblioteca **StdArq** no CD em anexo.

```
StringTokens :: (Char->Bool) {#Char} -> [{#Char}]
StringTokens f str = acum f str 0 "" []
where
    acum f str n ac li
        | n == size str && ac<>" " = reverse [ac:li]
        | n == size str = reverse li
        | f str.[n] && ac<>" " = acum f str (n+1) "" [ac:li]
        | f str.[n] = acum f str (n+1) "" li
        | otherwise = acum f str (n+1) (ac +++ {str.[n]}) li
```

Observe que na declaração do tipo de função, têm-se dois argumentos:

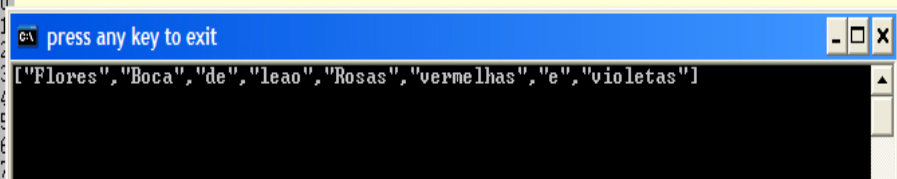
- O primeiro (**(Char->Bool)**) é outra função que deve testar se um caractere é um caractere de separação, caso seja, deverá retornar o booleano True, caso contrário, False.
- O segundo é a *string* da qual se deseja conhecer os *tokens*.

O exemplo mostrado na figura a seguir mostra como utilizar esta função para separar todas as palavras de uma *string*, utilizando como separador: o caractere de espaço, a vírgula, o ponto final, o ponto e vírgula, dois pontos, hífen, abrir e fechar de parênteses, abrir e fechar de colchetes, abrir e fechar de chave e o caractere de saltar linha ('\n').

```

1 module lexico
2 import StdEnv
3
4 StringTokens :: (Char->Bool) {#Char} -> [{#Char}]
5 StringTokens f str = acum f str 0 "" []
6 where
7     acum f str n ac li
8         | n == size str && ac<>"" = reverse [ac:li]
9         | n == size str = reverse li
10        | f str.[n] && ac<>"" = acum f str (n+1) "" [ac:li]
11        | f str.[n] = acum f str (n+1) "" li
12        | otherwise = acum f str (n+1) (ac ++ {str.[n]}) li
13
14
15 isSeparador :: Char -> Bool
16 isSeparador x = isMember x ['|','.',',',';',':', '-', '(', ')', '[', ']', '{', '}', '\n']
17
18
19 Start = StringTokens isSeparador "Flores: Boca-de-leao,Rosas (vermelhas) e violetas."
20
21
22
23 ["Flores", "Boca", "de", "leao", "Rosas", "vermelhas", "e", "violetas"]
24
25
26
27

```



A função **StringTokens** aplica, mapeia, uma função *booleana* de teste de caractere em cada caractere de uma *string*. Enquanto esta função, no caso **isSeparador** devolve falso (*False*) um *token* é formado, o mesmo é reconhecido quando a função *booleana* retorna verdade (*True*), no caso, quando encontra um separador. Feito isto o separador é eliminado (quantos existirem em sequência) e o reconhecimento de um novo *token* é iniciado, repetindo-se o processo até que a *string* retorne vazia.

Anexo X

Interfaces Visuais e Multimídia

Neste anexo, será mostrado como implementar interfaces visuais NDI e SDI⁶⁹, de tal forma a permitir que se criem aplicativos multimídia interativos nesta linguagem, a partir, novamente, dos conceitos básicos e fundamentais. Criar interfaces é um processo relativamente complexo em qualquer linguagem, principalmente se deseja implementar sistemas interativos, com leitura de posicionamento de mouse, *plotagem* de *bitmaps*, enfim, implementar recursos multimídia. Tornar tal processo claro, legível e reproduzível é um dos objetivos deste anexo em conjunto com os Capítulos 4 e 6. Desta forma, este anexo apresentará exemplos aplicativos onde passo a passo os conceitos de criação e manipulação de interfaces visuais serão inseridos e sedimentados. A seguir, será apresentado:

1. Como implementar interfaces NDI;
2. Como implementar interfaces SDI;
3. Como implementar interfaces SDI com Menus;
4. Como implementar interfaces SDI com Botões;
5. Como implementar interfaces utilizando registros no estado local de processos;
6. Como acrescentar barra de ferramentas (*ToolBar*) nas interfaces;
7. Como *plotar Bitmaps* em qualquer ponto da interface;
8. Como rastrear e manipular os eventos de mouse em áreas específicas da interface;

⁶⁹ NDI = No Document Interface, SDI = Single Document Interface. Existe, também a MDI = Multiple Document Interface, a qual não será objeto de estudo neste trabalho.