PROJETO E IMPLEMENTAÇÃO DE UMA PLATAFORMA PARA PROCESSAMENTO DE IMAGENS

Silvino Benevides Magalhães

Prof. Dr. José Antão Beltrão Moura
Orientador

Prof. Msc. Galileu Batista de Sousa
Co-orientador

Dissertação apresentada na Coordenação de Pós-graduação em Informática - COPIN, da Universidade Federal da Paraíba CAMPUS II, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

M188p Magalhães, Silvino Benevides.
Projeto e implementação de uma plataforma para processamento de imagens / Silvino Benevides Magalhães. - Campina Grande, 1999.
97 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1999.
"Orientação : Prof. Dr. José Antão Beltrão Moura, Prof. M.Sc. Galileu Batista de Sousa".
Referências.

1. Processamento de Imagens. 2. Plataforma para Processamento de Imagem. 3. Satellite Image Processing Language - SIPL. 4. Dissertação - Informática. I. Moura, José Antão Beltrão. II. Sousa, Galileu Batista de. III. Universidade Federal da Paraíba - Campina Grande (PB). IV. Título

CDU 004.932(043)

PROJETO E IMPLEMENTAÇÃO DE UMA PLATAFORMA PARA PROCESSAMENTO DE IMAGENS

SILVINO BENEVIDES MAGALHÃES

DISSERTAÇÃO APROVADA EM 10.02.1999

PROF. JOSÉ ANTÃO BELTRÃO MOURA, Ph.D
Orientador

PROF. GALILEU BATISTA DE SOUSA, M.Sc
Co-Orientador

PROF. BERNARDO LULA JÚNIOR, Dr.
Examinador

PROF. GIUSEPPE MONGIOVI, D.Sc
Examinador

PROF. PEDRO SÉRGIO NICOLLETTI, M.Sc
Examinador

CAMPINA GRANDE – PB

Aos meus pais, que me fizeram ser perseverante e acreditar que a verdadeira riqueza do homem reside na sabedoria.

Sumário

Esta dissertação apresenta o projeto e a implementação de uma plataforma para processamento de imagens que utiliza SIPL, Satellite Image Processing Language, uma linguagem com alta carga semântica para utilização em processamento de grandes volumes de imagens. Extensões baseadas no conceito de expressões regulares foram criadas para permitir a recuperação de imagens com base em especificações temporais relativas e absolutas. A flexibilidade do mecanismo suporta, em princípio, qualquer esquema de nomeação de imagens usado na prática, sem ficar restrito a um esquema de recuperação particular. A linguagem é fortemente tipada, suporta todos os comandos estruturados e tem uma variedade, ainda crescente, de comandos de manipulação de imagens, sem aparentes penalidades em tempo de processamento.

Abstract

This thesis presents the design and implementation of an image processing environment, which uses SIPL, Satellite Image Processing Language, a language with highly dense semantics for large scale image processing. Extensions based on the regular expression concept was developed to allow images recovery based on absolute and relative temporal specifications. This flexibility allows the support of all practical images naming schemes. The language is strongly typed, offers structrured programming constructs and has a growing variety of commands for image manipulation with no apparent performance penalties.

AGRADECIMENTOS

Gostaria de agradecer:

- Em primeiro lugar, a meu orientador, José Antão Beltrão Moura pela assistência e ensinamentos em Engenharia de Software durante todo o decorrer do mestrado. A meu co-orientador, Galileu Batista de Sousa, pela rígida exigência com assistência que tornou possível este desafio, que de início até mesmo eu julgava intransponível, demostrando ser o amigo certo das horas certas e incertas.

- Aos demais professores da COPIN pelo fortalecimento e agregação de conhecimentos em outras áreas da informática.

- As minhæirmãs, Silene e Eveline e a amiga, Márcia Morais Ximenes Mendes pelo incentivo ao desafio.

- Ao amigo e orientador profissional, Prof. Francisco Ariosto Holanda e a Antônio Cruz Vasques pelo apoio logístico que tornou isto possível.

- Aos meus amigos do mestrado que tornaram nossa convivência mais alegre e menos saudosa.

Conteúdo

Índice de Figuras e Tabelas

Capítulo 1

Introdução

Processar imagens tem sido caracterizado como a operação de obter uma imagem a partir de outras. Estas operações visam realçar detalhes, remover ruídos, manipular propriedades no domínio da imagem, das cores ou da grandeza física a elas associada, para uma melhor visualização e interpretação [GV94].

Computacionalmente uma imagem é, na sua forma mais simplificada, uma matriz de inteiros, cada um representando uma função f(x, y) da intensidade luminosa, no ponto de coordenadas espaciais (x, y). Tipicamente o valor da função é proporcional ao brilho da imagem neste ponto. Imagens que possuem informações em intervalos ou bandas distintas de freqüências, necessitam desta função f(x, y) para cada banda, como acontece com as imagens coloridas. Para o Processamento Digital de Imagens (PDI) é conveniente que estas sejam representadas no formato matricial.

A representação matricial de uma imagem em PDI, permite que operações algébricas em imagens sejam reduzidas às operações de soma, e produto de uma matriz por um escalar; que são comuns na álgebra de matrizes[GV94] e álgebra de imagens[RWD90]. Na multiplicação de uma imagem por um escalar, a operação é dita unária. Operações unárias

transformam uma imagem em outra imagem. Operações binárias combinam duas imagens para formar uma nova imagem. Operações unárias com imagens são chamadas de filtragem[GV94].

Métodos de realce são essencialmente, a manipulação do contraste, filtragens e eliminação de ruídos. Para efetuar essas operações, o modelo de cores RGB (*Red, Green, Blue*) utilizado em monitores, é de difícil utilização, porque não se relaciona diretamente com as noções intuitivas de matiz de cor, saturação e brilho. Visando a manipulação direta destas variáveis, outros modelos foram desenvolvidos, tais como HSI (Hue, Saturation, Intensity) e HSV (Hue, Saturation, Value)[GW93].

A resolução espacial ou resolução geométrica da representação matricial, *m* linhas x *n* colunas, se expressa em termos absolutos, não fornece muita informação sobre a resolução real da imagem quando processada em um dispositivo físico, já que ficamos na dependência dos tamanhos físicos da janela de visualização e do pixel do dispositivo. Quando a resolução real da imagem é superior a resolução do dispositivo, operações de fator de escala, *zoom*, são aplicadas para a visualização de detalhes.

A representação gráfica da distribuição de valores de cinza e cores de uma imagem, denominada histograma, revela uma idéia instantânea sobre as características desta imagem. Histogramas podem ser obtidos da imagem toda ou de parte desta. Suas informações ajudam a decidir quais mudanças podem melhorar a qualidade de uma imagem.

Imagens de satélites parecem visualmente com baixo contraste, uma vez que o sistema visual humano só consegue discriminar cerca de 30 tons

de cinza, e mesmo assim quando estes se encontram bastante espalhados no intervalo de 0-255. Para que o analista humano possa extrair corretamente as informações contidas nestas imagens, o seu histograma comprimido precisa ser expandido para ocupar todo o intervalo possível. Este conceito é a base do chamado aumento de contraste e se constitui em uma das técnicas importantes mais poderosas e utilizadas em PDI para extração de informações.

O contraste de uma imagem é uma medida do espalhamento dos níveis de intensidade que nela ocorrem. A função de transferência especifica a transformação do contraste e sua ação não depende da intensidade dos pixels ao redor do pixel considerado, isto é, ela é uma operação pontual. Em PDI a função de transferência é implementada como uma tabela por questões de eficiência e é denominada de "Look-up Table" (LUT). Por meio deste artifício, evita-se especificar a função de transferência matematicamente e aplica-la pixel a pixel, para calcular as novas intensidades, o que seria extremamente vagaroso.

Todas as imagens possuem limites entre áreas com diferentes respostas à energia eletromagnética. Estes limites podem representar o contato entre objetos com respostas espectrais diferentes. Em uma imagem monocromática, esses limites representam portanto mudanças de um intervalo de intensidade para outro. Limites deste tipo são conhecidos como bordas. Ocupam geralmente áreas pequenas da imagem, sendo mais estreitos que largos e por variarem bastante em áreas pequenas, são chamados de feições de alta freqüência. Já limites gradacionais, que variam

mais uniformemente com a distância, sendo consequentemente menos nítidos, são chamados de feições de baixa freqüência.

A enorme mistura de freqüências em uma imagem dificulta sobremaneira a interpretação de feições com freqüências específicas. Para contornar esse problema e melhorar a aparência da distribuição espacial das informações, são utilizadas técnicas de filtragem espacial de freqüências. Estas consistem em realçar seletivamente as feições de alta, média ou baixa freqüências que compõem as imagens. Uma das técnicas de filtragem de freqüência é o processo conhecido por convolução. São três os tipos básicos de filtros de convolução: Filtros Passa-baixas, Passa-altas e Direcionais[GV94].

As técnicas de filtragem são transformações da imagem pixel a pixel, que não dependem apenas do nível de cinza de um determinado pixel, mas também do valor dos níveis de cinza dos pixels vizinhos, na imagem original, onde os pixels mais próximos contribuem mais na definição do novo valor de nível de cinza do que os pixels mais afastados. Uma matriz de filtragem denominada máscara ou janela, delimita a região de n x n pixels na qual os pixels vizinhos influem no novo valor do pixel. A máscara percorre toda a imagem pixel a pixel no processo de filtragem. A operação do filtro multiplica cada valor dos n pixels da imagem pelo respectivo valor dos n pixels da máscara. Em seguida substitui-se o valor do pixel central dessa área pelo valor calculado a partir desses n valores, o qual depende do tipo de filtro utilizado, obtendo-se desta forma o novo valor do pixel correspondente da imagem de saída.

1.1 Imagens de Satélites

As imagens enviadas por satélites são mais complexas, possuem algumas características que as diferenciam das outras imagens digitais [JBA89], entre as quais sua estrutura e resolução. Contém informações de georeferência (posição geográfica de latitude e longitude) e vários canais com informações distintas, entre as quais podemos citar temperatura e umidade, informações da superfície do globo como vegetação, que podem ser obtidas pela combinação de canais. Por terem georeferência, as operações algébricas com estas imagens fazem sentido, pois se referem ao mesmo objeto, porém em instâncias ou dimensões físicas diferentes. Esta estrutura depende ainda do tipo de satélite, se é ou não geoestacionário, e do propósito para o qual foi concebido, tendo, portanto, padrões variados.

Duas grandes categorias se destacam: as imagens meteorológicas e as de sensoriamento remoto. As primeiras estão interessadas nas grandezas físicas da atmosfera e da temperatura da superfície do globo terrestre, pois baseados nos dados embutidos nas imagens juntamente com outras medições aéreas, marítimas e terrestres é feita a analise de predição do tempo. As de sensoriamento remoto são utilizadas em estudos geológicos, geoquímicos, análise de vegetações, cartografia entre outros. Estas informações necessitam ser processadas, visualizadas e destacadas para uma boa interpretação pelo usuário.

Imagens capturadas de satélite têm tamanhos elevados e possuem uma alta taxa de aquisição, gerando desta forma um volume de armazenamento muito grande e de difícil gerenciamento. O grande tamanho se deve à

enorme área que representam, sendo ainda proporcional à resolução do sensor, que varia de frações de metros nos satélites militares ou destinados a análise mineral de solos, a quilômetros, em algumas imagens meteorológicas. A alta taxa de aquisição se deve a uma freqüência de captura superior a 24 imagens/dia/ satélite e vem da necessidade da análise interpretativa, representada pela comparação das imagens coletadas em intervalos de tempo regulares, onde a animação de um conjunto, enfatiza as características mutantes, facilitando a análise e sua interpretação.

Noutras ocasiões necessitamos compor imagens para ressaltar algumas características pouco ou nada visíveis em suas imagens originais. Estas imagens devem ser geradas em tempo real, pois suas composições são variadas e devem poder ser reproduzidas a qualquer instante de suas originais. A maioria destes procedimentos analíticos são rotineiros, o que favorece a utilização de uma linguagem em vez de uma ferramenta iterativa, evitando a geração de imagens intermediárias e um agravamento do armazenamento.

Para os satélites da série Meteosat cada imagem capturada tem cerca de 2Mb, que a intervalos de um quarto de hora, resulta num espaço de armazenamento em cerca de 5,8 Gb mensais, 69,6 Gb anuais, somente para um único satélite. Esta captura periódica tem como conseqüência um armazenamento de altíssimo volume, com problemas de gerenciamento e recuperação destas imagens. O processamento destas imagens requer software específico e atualizado, pois estes satélites possuem um ciclo de vida não muito longo e tecnologia sempre em evolução. Ademais outras peculiaridades estão presentes, entre as quais apresentamos:

- Estas imagens contem informações específicas sobre si em cabeçalhos com seções que variam em número e tamanho, que são utilizadas pela aplicação durante e após sua leitura, sendo peculiares a cada satélite.
- Nem todas as imagens coletadas apresentam a mesma importância e freqüência de utilização através do tempo. Imagens sem conteúdo significativo de algum fenômeno especial, motivo de alguma pesquisa ou estudo comparativo, só apresenta importância para visualização e predição do tempo referente ao período de sua captação, sua disponibilização torna-se reduzida para ser mantida em um banco de dados.

1.2 Apresentação

Esta dissertação apresenta o projeto e a implementação da plataforma de manipulação de imagens que tem embutida uma linguagem denominada **SIPL**, Satellite Image Processing Language. SIPL contém um esquema de recuperação de imagens que alia a flexibilidade de expressões regulares [Mar91] com estratégias de recuperação temporal.

Comandos estruturados estão definidos, bem como diversos operadores sobre imagens e outros tipos suportados, baseados em um ambiente de execução específico, adequado a este tipo de processamento. Procura-se manter a linguagem em um nível semântico de fácil utilização por programadores não especializados. A interface gráfica do ambiente dispõe de varias janelas gráficas onde o usuário pode atuar interativamente com as

imagens visualizadas. A utilização do mouse e de botões da interface estão associados a parte dos comandos disponíveis na linguagem.

1.3 Justificativa

Linguagens especializadas trazem aumento de produtividade já que são voltadas para as atividades da área fim, obtendo-se também produtos em menor tempo. Existem outras linguagens para processamento de imagens, mas não tão especializadas e com nível semântico idealizado para SIPL.

Programas comerciais existentes em hardware padrão para manuseio de imagens, manipulam imagens individuais, mas não suportam imagens de satélites nem possuem a idéia de projeto de visualização, contida em um programa que armazena um conjunto de operações representando uma rotina do usuário.

Suportando nativamente operações específicas da área fim, reduz-se o nível de complexidade e o esforço de codificação necessários à geração de programas. Manipulações conseguidas com os diversos operadores implementados permitem uma diversificação de produtos que podem ser obtidos com o uso da linguagem.

O sistema operacional UNIX possui poderosas *shells*, interpretadores de comandos que rodam como processos, e uma variedade de programas, através dos quais seria possível gerar arquivos, contendo nomes de arquivos de imagens pesquisados com auxilio de utilitários que se utilizam de expressões regulares, entre outros com capacidade de manipulação de

strings e de arquivos. O arquivo gerado seria posteriormente lido por um ambiente para PDI, como resultado de uma recuperação de imagens.

Um programa como este além de complexo é passível de manutenção freqüente, necessitaria de muitos parâmetros a serem passados na linha de comando tendo em vista a complexidade e as anomalias dos esquemas de nomenclatura utilizados na prática. Para a formação deste arquivo haveria muito redirecionamento entre os utilitários já que o resultado não seria possível de ser executado por um único utilitário. Este programa teria a mesma complexidade da implementação da recuperação temporal embutida em SIPL, obviamente processado em menor velocidade.

Como estas características são nativas do UNIX, estariam limitadas a este sistema operacional, ficando de fora o ambiente Windows entre outros, reduzindo a portabilidade da plataforma.

1.4 Objetivos

A plataforma projetada e implementada objetiva a utilização de hardware de baixo custo para o processamento de imagens em larga escala, utilizando-se de um novo conceito para recuperação de imagens, incluindo a idéia de projeto de visualização, que é a automatização de rotinas de usuários através do armazenamento de programas escritos em sua linguagem própria, onde há operadores específicos, adequados ao processamento de imagens meteorológicas, obtendo-se desta forma maior produtividade.

A portabilidade de seu código e a utilização de um interpretador facilitam a migração para plataformas distintas. Os diferentes formatos de imagens implementados facilitam a geração e diversificação de produtos obtidos. Os comandos da linguagem sintetizam operações complexas, transparentes ao usuários, onde o resultado equivalente destas operações em linguagem de programação genérica, demandaria um grande esforço de codificação e um programador especializado.

1.5 Contribuições

A não existência de uma linguagem para processamento de imagens que utilize computadores pessoais sob MS-Windows, manipule imagens georeferenciadas e de outros formatos comuns do mercado, deverá ser de incentivo a profissionais das áreas de meteorologia e computação a iniciarem esforços no sentido de ampliar e desenvolver as idéias aqui iniciadas.

Comandos que processam múltiplas operações complexas em processamento de imagens, incentivam a programação feita pelo usuário final, com aumento de produtividade e favorecendo a pesquisa, pois o programador é o próprio pesquisador.

Filtros definidos pelo usuário e filtragens múltiplas em várias freqüências, favorece a pesquisa e o aparecimento de novas imagens produtos.

A recuperação temporal de imagens é uma idéia nova, muito útil nas áreas afim e poderá vir a ser usada em outros contextos.

1.6 Organização da dissertação

A dissertação está organizada como a seguir. O capítulo 2 apresenta uma visão geral da plataforma e sua comparação com outras ferramentas que também oferecem recursos de programação. O capítulo 3 descreve a especificação da linguagem SIPL. A estrutura de programas é discutida, incluindo desde a declaração de variáveis e comandos, à utilização dos recursos de recuperação temporal de imagens. O capítulos 4 discute aspectos de implementação da linguagem e da interface gráfica. O capítulo 5 apresenta seções de revisão de objetivos, contribuições e sugestões de trabalhos futuros. Os apêndices A e B contém respectivamente as especificações léxicas e sintáticas da linguagem proposta e implementada. O apêndice C contém exemplos de programas em SIPL.

Capítulo 2

Visão Geral da Plataforma

Ao se analisar aplicações na área de PDI, identificam-se dois tipos de ferramentas:

- Bibliotecas contendo um conjunto de funções para manuseio de imagens a partir de uma linguagem hospedeira, normalmente C [KR88] ou C++ [Stro91]. Estas bibliotecas, eventualmente, são disponibilizadas através de ambientes de programação visual[KS92];
- Programas aplicativos com capacidade de manuseio de imagens individuais, em geral usando uma Interface Gráfica de Usuário e de forma interativa [Jasc97].

Sob a ótica da produtividade e eficiência pode-se constatar facilmente a inadequação dessas ferramentas à manipulação de grandes conjuntos de imagens. Em geral, essas ferramentas falham porque:

- Linguagem como C e C++ ainda demandam alta qualificação para que requisitos mínimos de produtividade possam ser alcançados, muito embora tenham alto desempenho, critério bastante relevante em aplicações dessa natureza;

- O manuseio individual é baseado em dispositivos apontadores, tais como *mouse*, leva a um trabalho artesanal e suscetível a erros, sendo inviável de ser aplicado a muitos dados de forma repetida. A seu favor pesa a baixa especialidade requerida para trabalhar com tais sistemas, bem como o aspecto da prototipação rápida.

SIPL objetiva reduzir a qualificação necessária, ao mesmo tempo que mantém um desempenho compatível com linguagem C e, em parte, a simplicidade de ambientes interativos. Isto é possível porque a linguagem é fundamentada na implementação de um RUNTIME SYSTEM (RTS) em C++ para manipulação de imagens e listas. Diferentemente de linguagens como C/Pascal, que têm um RUNTIME razoavelmente simples, SIPL tem um RTS bastante sofisticado para suportar seus tipos e operações embutidas.

O *runtime* de SIPL suporta os tipos mais comuns de operações sobre imagens, manipulação de cores e histogramas, filtros, leitura e gravação de diversos tipos e formatos de imagens, incluindo os de satélites meteorológicos; área para a qual julga-se ser SIPL muito apropriada. Independente do formato original da imagem, internamente cada imagem é armazenada em um padrão similar ao *Windows Bitmap Format* [BS94]. Muitas operações deste *runtime* não estão implementadas em SIPL, pois são mais adequadas a operações interativas de usuário, requerendo uma GUI mais sofisticada, aumentando em muito o tempo de desenvolvimento deste protótipo.

A idéia de projeto de visualização para um ambiente de processamento de imagens, sintetiza uma serie de operações a serem aplicadas à um

conjunto particular de imagens, com o objetivo de obter repetidas vezes, imagens produtos com determinadas características desejadas. Um meio de obter isto é através de procedimentos armazenados, onde um programa em uma linguagem apropriada contendo comandos e operadores específicos, fariam as operações necessárias ao processamento desejado.

Como as outras ferramentas, esta também trabalha de forma iterativa, muito embora operações desta natureza ainda necessitem de expansão, ela enfatiza o processamento repetitivo, o qual se constitui na principal característica da rotina de trabalho de quem processa imagens em larga escala. Projetar uma plataforma que fizesse todas as operações somente através de programas, seria ignorar as imagens individuais, ou os casos excepcionais, onde se faz necessária a participação direta do usuário. Portanto nos parece lógico também implementar comandos de ajustes que estariam disponíveis na interface gráfica, onde ações do mouse, botões, barra de rolagem e outros objetos da interface estariam associados a comandos da linguagem ou operações contidas em funções da biblioteca de imagens.

SIPL não objetiva uma competição em qualidade com outras linguagens existentes, já que imagens dependendo de sua origem, tem características peculiares, requerendo para seu tratamento software específico. Está voltada para tratamento de imagens georeferenciadas de satélites meteorológicos, que não são tratadas por software genéricos. Entretanto suportar diversos formatos de imagens já utilizados por software genéricos, foi contemplado no projeto por ser bastante funcional.

2.1 A Interface Gráfica de SIPL

Embora SIPL possa ser utilizada para a geração de imagens produtos e armazená-las para visualização posterior em outro ambiente, sua capacidade de exibição e de animação é uma característica altamente desejável, visto que a geração em tempo real não necessita de armazenamento extra e torna-a uma ferramenta valiosa na análise interpretativa de imagens, objetivo deste projeto.

Um protótipo da interface gráfica de usuário foi projetado e implementado, embora não se tenha disponibilizado ainda todos os comandos desejados sob a forma iterativa, dispomos de um ambiente gráfico para a visualização de imagens utilizando comandos disponíveis na linguagem.[1] Esta interface já é composta de várias janelas funcionais, contendo um mínimo de comandos iterativos necessários para utilização em meteorologia. A janela base é a única carregada inicialmente pelo ambiente, possui uma caixa de texto para a entrada do *script* a ser executado e um conjunto de botões que se destinam a execução, edição de *scripts* e saída do ambiente. Na sua base há um rodapé, onde é mostrado texto com as mensagens do ambiente.

Programas são executados pela entrada do nome do *script* na caixa de texto Script e o pressionamento do botão executar. Mensagens de erros quando detectadas são mostradas no rodapé da interface. Erros poderão ser corrigidos através de um editor de texto conectado ao botão editor. O

[1] Existem realmente dois ambientes, um que é uma pequena *shell* não gráfica e outro gráfico que usa SIPL como suporte para visualização e animação de imagens e se constitui no protótipo de uma plataforma de processamento de Imagens para meteorologia.

último botão presente na interface é o botão para finalizar SIPL. A figura 2.1 mostra a janela base da interface de SIPL.

O usuário pode solicitar a exibição do conteúdo de uma variável do tipo imagem, ação correspondente a *show* em SIPL, que mostra na janela gráfica a respectiva imagem. Se a variável for uma lista contendo imagens, o feito de *show* é uma animação do conjunto de imagens. Se o usuário utilizar o comando *zoom* em uma variável imagem, antes de sua visualização, o efeito na visualização será em uma escala ampliada, e assim sucessivamente.

Um editor de texto está acoplado ao botão editor da janela base da interface gráfica, tem o nome de *sipledit*, mas pode ser substituído por outro editor disponível no ambiente gráfico utilizado, bastando que o editor desejado tenha uma cópia ou no UNIX, um *link* para este nome. Tem dois menus *popup*, um destinado a arquivos, com funções de novo, abrir, salvar, salvar como e impressão, o outro, com funções de edição, incluindo copiar e colar. Não foi implementado a função desfazer, *undo*.

A janela de maior interesse é denominada *Animator*, figura 2.3, que é composta de uma área de desenho contendo barras de rolagem horizontal e vertical. Esta janela tem seu tamanho inicial ajustado automaticamente em função do tamanho da imagem carregada, entretanto está limitada a um tamanho inicial de 500 x 500 pixels. Se a imagem for maior que este limite inicial, as barras de rolagem tornam-se ativas, permitindo a visualização de regiões não mostradas, onde o tamanho das imagens está limitado pela memória da máquina. Este tamanho inicial pode ser modificado manualmente através do arraste de suas extremidades pelo mouse, onde o

novo limite é o tamanho dado pela resolução do dispositivo gráfico. Após a carga de imagens, as quais podem ter sido previamente tratadas por comandos da linguagem existentes no *script*, um clique no botão direito do mouse sobre a área de desenho mostra uma outra janela, denominada controles. Esta última contém botões para ativar outras janelas e operações de aumento e redução de escala das imagens visualizadas.

A janela de animação, figura 2.2, é disponibilizada somente quando mais de uma imagem foi carregada no ambiente. Controla a velocidade de animação, imagens por segundo e o intervalo de parada entre seqüências animadas, através de controles deslizantes, possuindo ainda botões de parada, avanço e recuo. Obviamente que a operação de animação não é concomitante com as de avanço ou recuo, já que a primeira representa um avanço automático por meio de temporizador na série de imagens, ficando estes últimos botões desativados, no que é evidenciado pelo tom esclarecido dos mesmos. Por outro lado, quando uma seqüência de imagens não está em animação, é possível avançar ou recuar uma imagem da série através de seus respectivos botões.

Os botões de ampliação e redução de escala assumem por omissão um fator 2, mas este pode ser alterado em sua janela própria. A imagem ampliada ou reduzida pode ser mostrada na janela de desenho, ou sua ampliação por um fator 2 em uma janela menor, denominada lente, a qual mostra uma área da janela de desenho apontada pelo mouse, e que acompanha dinamicamente o movimento deste.

Uma outra janela controla a tabela de cores, (LUT), utilizada para aumento do contraste e desta forma enfatizar as informações analisadas.

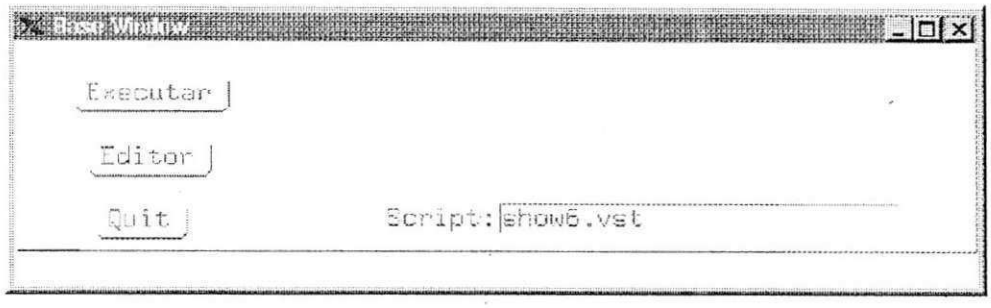


Figura 2.1 Janela Base

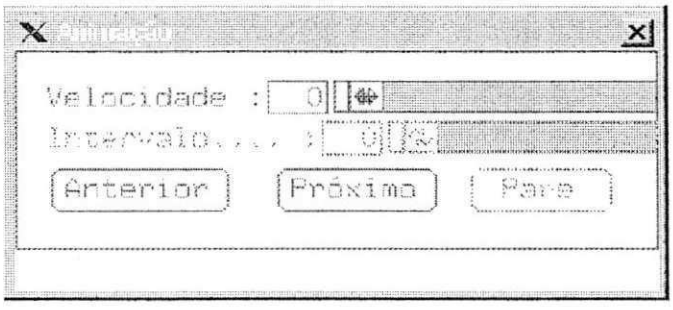


Figura 2.2 Janela Animação

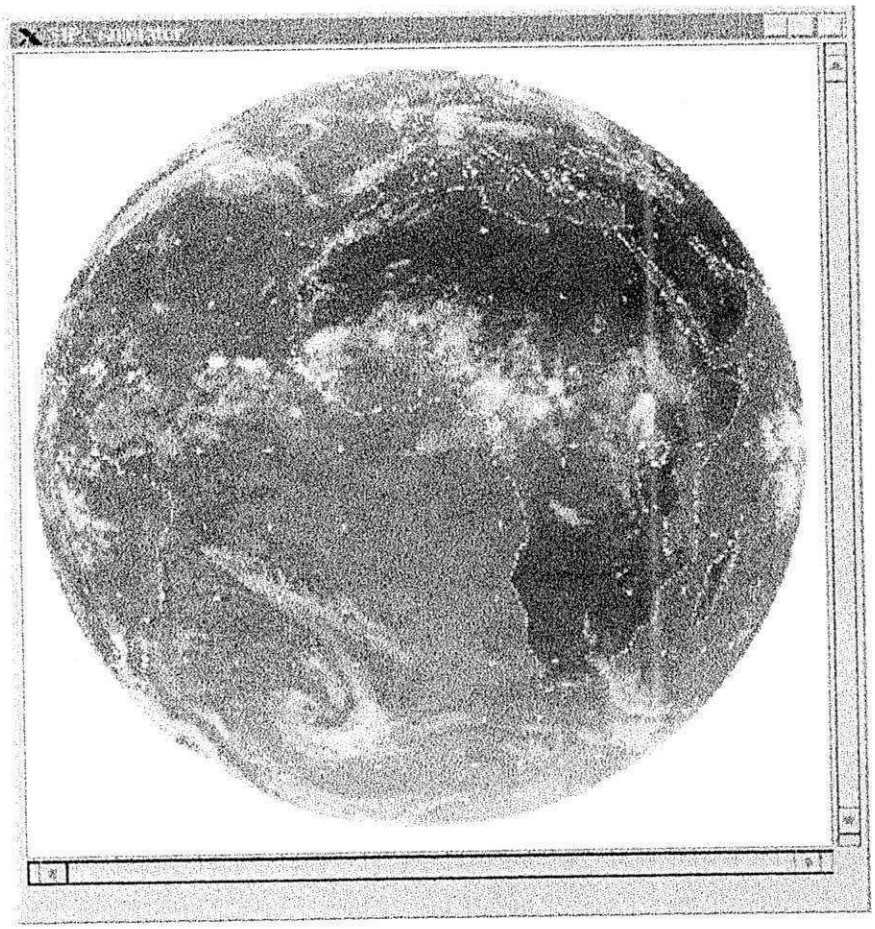


Figura 2.3 Janela Animator

2.2 Avaliação das Ferramentas Existentes

Operacionalizar o Processamento Digital de Imagens (PDI) através de linguagens específicas é uma atividade relativamente antiga [WR90]. Como exemplos podemos citar Khoros [RW91], NASA iAXE [Pow93] e MEDx [http://www-mrips.od.nih.gov/uguide/ugtoc.htm].

Khoros é um ambiente de prototipação rápida, baseado no paradigma *dataflow* que inclui uma linguagem visual, *Cantata*, de múltiplos propósitos, que inclui em sua biblioteca funções para tratamento de imagens. Por não ter propósitos específicos, as funções de sua biblioteca estão limitadas a fazerem o que normalmente as ferramentas gráficas iterativas[Jasc97] fazem.

iAXE é uma linguagem convencional que suporta imagens do satélite NOAA e outros formatos como IDIDAS, TGA, MCIDAS [BS94], entretanto é baseada em hardware específico e não possui regras de associação de nomenclatura temporal para as imagens suportadas, que torne possível a recuperação automática dessas imagens do meio de armazenamento. Suporta composição, múltiplas visualizações e informações estatísticas, mas é limitada a uma única carga de imagem por vez, portanto não suporta animação de uma seqüência de imagens nem processamento de imagens em larga escala. Sua função de leitura de imagens é única e requer a passagem de muitos parâmetros, o que permite ler imagens de uma forma genérica, entretanto exigindo do usuário a especificação detalhada de como deverá ser lida a imagem. São requeridos incondicionalmente os seguintes parâmetros:

- i= nome_arquivo.ext
- t= translação de dados. Suporta 6 tipos:
  - T – térmica
  - A – albedo. Albedo é percentual de refletância.
  - V – visível
  - P – pseudocolor
  - U – definido pelo usuário por meio de arquivo
  - B – binário, não há translação.
- f= formato. São suportados os formatos IDIDAS, TGA, MCIDAS, NOAA 1B.

- s= organização do armazenamento. Especifica o número de bytes por pixel (1,2,3,4) seguidos por uma letra indicando o entrelaçamento, (S) scanline, (P) pixel ou (C) canal seqüencial.
- rgb= canais de cores. Um só canal resulta em imagens em tons de cinza, 2 canais resulta na carga das cores vermelha e verde. O 3° canal inclui a cor azul.
- o= gráfico de sobreposição, tais como grade de latitude/longitude e linha costeira dos continentes.
- pf= arquivo de parâmetros, os podem definir parte ou todos os parâmetros para uma carga de imagem.
- skp= número *skip*. Salta número bytes. É utilizado para saltar cabeçalhos.

- bxy= *offset*. Deslocamento nas coordenadas do *buffer* de mémoria[x0, y0], especificando a origem de uma imagem na memória de visualização.
- fxy= *offset* no arquivo imagem. As coordenadas x,y especificam o início doas dados na imagem para visualização.
- nc= número de colunas na área de visualização quando de tamanho diferente da imagem.
- nr= número de linhas na área de visualização quando de tamanho diferente da imagem.
- rf= fator de redução de tamanho para a imagem visualizada.

Operações de escala e ângulo de visualizações são implementados como funções de mouse, enquanto as operações de composição estão limitadas às composições geradas entre os diferentes canais, filtros e definição de sub-àreas. SIPL utilizando especificações temporais e georeferência, caracteriza a instância de imagens, por se tratar de um mesmo objeto, diferindo no tempo de coleta e nas propriedades físicas representadas nos pixels, implementando desta forma operações algébricas em imagens.

A noção de *scripts* em MEDx se assemelha a SIPL em muitos aspectos. MEDx é um pacote de software portável para processamento de imagens voltado para pesquisadores de imagens medicas. Foi desenvolvido em associação com o National Institutes of Health-USA (NIH), e Bethesda, pela Sensor Systems Inc., uma empresa de engenharia e consultoria em tecnologia de imagens. Com ele pode-se mostrar imagens e menus de

controle, utilizando um ambiente gráfico como X Windows em estações de trabalho *Unix*.

Conforme detalhado nas especificação de SIPL no Capítulo 3, procuro-se dar suporte à importação de imagens em vários formatos a exemplo de MEDx, entretanto esta utiliza formatos proprietários utilizados em equipamentos da área médica.

É possível introduzir texto, criar animações e scripts como em SIPL, como também não necessita de hardware fora de padrão. Ambas utilizam pseudo cores fazendo uso de *Look-up Table* (LUT), implementação que simula a função de transferência descrita no capitulo 1, que transforma um dado valor de entrada a um correspondente valor de saída na visualização de cores.

O conceito de projeto de visualização também existe em MEDx, entretanto suas implementações em muito diferem. SIPL obtém suas imagens através de programas armazenados e não necessita de armazenamento de imagens intermediárias. Diferentemente de SIPL, que é voltada para PDI em larga escala e utiliza recuperação temporal de imagens, MEDx utiliza pastas e páginas. Pastas são reservatórios de todas as imagens, gráficos e texto relacionados com uma tarefa particular e deve corresponder a imagens e análises para um objetivo específico ou paciente. Os objetos existentes em uma pasta são armazenados em páginas. Uma página pode conter texto, gráfico e/ou uma imagem. Para executar qualquer tarefa em MEDx é necessária uma pasta. Uma pasta corresponde a um diretório que contém arquivos para páginas de uma pasta, sempre que uma pasta é salva um diretório é criado para conte-la e todas as páginas e

imagens são salvas no mesmo. Os arquivos para cada página consistem de duas partes separadas, os dados da imagem no formato VEST *canvas*, e outro contendo as descrições de visualização e tabela de mapeamento de cores. Páginas são usadas para agrupar imagens, anotações e parâmetros de visualização.

SIPL não é implementada tendo por base uma arquitetura distribuída como MEDx, que é estruturada como um conjunto de máquinas virtuais. Cada máquina executa uma tarefa particular.

Uma máquina é responsável pela interface com o usuário e apresentação dos resultados e usa Tcl/Tk[Ous94]. Uma outra máquina, referenciada como *xc*, é responsável pelas operações de processamento de imagens. Esta máquina gerencia todos os dados da imagem e qualquer informação auxiliar a ser armazenada com a imagem. A terceira máquina é a interface de aplicação do programador, chamada *ImageScript*. Esta máquina estende Tcl/Tk pela adição de centenas de funções para executar todas as operações disponível na interface de usuário. Estas máquinas se comunicam em uma rede por meio de um protocolo de mensagens projetado para executar rapidamente e eficientemente neste ambiente.

Devido as características de suas imagens, MEDx possui visualização adicional de imagens volumétricas. Volume é uma imagem com três dimensões, 3D. As dimensões tipicamente correspondem a comprimento, altura e profundidade. Contudo em MEDx este não é sempre o caso. Por exemplo, o volume de uma série temporal tem dimensões correspondendo a comprimento, altura e tempo. A dimensão temporal de SIPL é representada em imagens georeferenciadas distintas, havendo a necessidade

de nomenclatura e mecanismos de recuperação de imagens que diferem apenas no tempo e nos valores das grandezas físicas associadas. Como contém informações de suas posições espaciais, são verificadas automaticamente se representam o mesmo objeto em tempos distintos.

MEDx permite a programação de interfaces customizadas mediante programação em Tcl/Tk, com botões programáveis. Um botão programável é um objeto gráfico especial que executa algum procedimento ou ação que o ambiente seja capaz de executar. ImageScript é a *linguagem* de script de MEDx. Esta linguagem é uma extensão a Tcl/Tk que provê chamadas de funções e operações disponíveis na interface de usuário, provendo uma linha de comando para execução de processamento e visualização. Desde que Tcl provê construções genéricas de programação tais como controle de loop e procedimentos, é possível automatizar tarefas de rotina para processamento. É um ambiente de programação baseado em interpretação que facilita prototipação rápida para desenvolvimento. Entretanto programar em Tcl requer muitos conhecimentos sobre linguagens de programação e ambientes gráficos.

A linguagem utilizada por SIPL não permite a criação de interfaces customizadas, mas devido sua carga semântica, não demanda conhecimentos especializados em programação, permitindo desta forma uma utilização mais apropriada para o usuário final.

A arquitetura não distribuída de SIPL favorece a velocidade de processamento. A execução de scripts em MEDx segue a seqüência de eventos descrita abaixo:

1. O argumento é processado.

2. A mensagem apropriada é empacotada e enviada a xc (a máquina de processamento de imagens).
3. xc desempacota a mensagem, executa validação dos argumentos e executa as operações requeridas.
4. Uma mensagem é enviada de volta de xc com o resultado da operação.
5. A extensão de Tcl processa esta mensagem e retorna um código de resultado.

Há duas formas de executar um script em MEDx. A primeira semelhante a SIPL e mais fácil, é sem uma *shell*, onde o nome do script é digitado em uma caixa de texto com o precionamento do botão de execução. A segunda é com a utilização de uma shell iterativa, onde linhas são digitadas uma a uma, aguardando o efeito da interpretação de cada comando digitado. O interpretador de Tcl/Tk é geralmente conhecido como *wish*, a versão de ImageScript é chamada de *vestwish*.

Segue um exemplo de comando do script *DemoSelf.tcl* na shell de MEDx :

```
vestwish> source $PXHOME/folders/Demo/DemoSelf.tcl
```

A shell iterativa é recomendada para o desenvolvimento de scripts, onde é possível monitorar o que acontece após a execução de cada comando.

A UFPe tem um projeto para o manuseio de grandes volumes de imagens, onde a fonte destas são documentos históricos. Programas

existentes no mercado e conhecidos como GED, (G)erenciadores (E)letrônicos de (D)ocumentos, utilizam banco de dados para armazenamento dos índices de recuperação de documentos em meio eletrônico, entretanto as imagens destes residem em meio óptico, pois em geral o volume de armazenamento é muito grande e os documentos são caracterizados como legais, isto é, não podem sofrer modificações. Recuperar documentos desta forma é um dos objetivos do "Projeto Nabuco: Um ambiente para Processamento de Grandes Acervos de Imagens" e da tese de mestrado defendida em dezembro de 1994 "Nabuco: Uma Base de Dados para Documentos Históricos", vista em http://www.ufpe.br/~nabuco, é um exemplo deste esforço. SIPL com algum acréscimo à recuperação temporal, poderia ser utilizada para os mesmos propósito conforme será explicado na seção 3.12.

Todos esses programas têm a habilidade de manipular imagens individualmente. Infelizmente esta não é a principal atividade na maioria das instituições que trabalham com imagens. Normalmente é a combinação de um conjunto de imagens que leva à geração de um produto em PDI [SM96]. O gerenciamento de grandes volumes de imagens, incluindo busca e armazenamento, por si, é uma atividade complexa.

Laboratórios de meteorologia capturam várias centenas de imagens a cada dia. Organizar esse grande volume de dados e ainda ter acesso seletivo a qualquer subconjunto de imagens, tem sido um desafio nesses ambiente em várias instituições, entre as quais podemos citar a Fundação Cearense de Meteorologia–FUNCEME e o Instituto Nacional de Pesquisas Espaciais–INPE. O acesso aos sites http://www.funceme.br e http://www.cptec.inpe.br, dois

dos maiores repositórios de imagens de satélites meteorológicos do país, mostram as evidências de quão difícil é manipular um grande volume de imagens.

Capítulo 3

Especificação de SIPL

SIPL é uma linguagem de programação interpretada e esta voltada para o desenvolvimento de novos produtos de imagens em meteorologia. Vários fatores contribuem para o rápido desenvolvimento de programas por usuários iniciantes:

- Como imagem é um tipo diferencial e o principal da linguagem, a complexidade do tratamento de imagens é substituída de forma transparente para o usuário pela utilização de operadores algébricos nas operações com imagens. Desta forma SIPL disponibiliza soma de imagens, obtenção uma imagem contendo os valores mínimos ou máximos de um conjunto de imagens com comandos específicos, e outras operações com diversos comandos disponíveis, descritos na seção 3.8.
- Sendo interpretada é mais rápido o desenvolvimento de programas, pois a execução deste para com a indicação do primeiro erro encontrado, não havendo um processo de compilação em separado seguida de uma execução.

As seções que se seguem descrevem os tipos suportados, definição de constantes e variáveis, atribuição de valores, uso de expressões em SIPL e a

estrutura de *scripts* e programas que são uma seqüência eventualmente unitária destes últimos.

3.1 Tipos Suportados

Analisando as áreas que se utilizam de imagens em grande número, especialmente meteorologia, procurou-se capturar as atividades mais comuns e as entidades computacionais, tais como, tipos de dados e seus agregados, operadores e funções, que viessem a ser traduzidas, de uma forma melhor e simples, em comandos SIPL.

3.1.1 Tipos Primitivos:

- Int – Tipo primitivo que define constantes e variáveis inteiras. Os pixels das imagens são números inteiros no intervalo de 0 – 255. Sua magnitude eqüivale ao inteiro em C.
- Real – Tipo primitivo utilizado na definição de constantes e variáveis fracionárias. Operações envolvendo alteração de fator de escala de visualização e conversão de sistemas de cores, são exemplos de utilização de números de ponto flutuante..
- String – Este tipo primitivo é utilizado para a definição de literais e variáveis com o propósito de representar nomes de arquivos para imagens.

- Image – Obviamente este é o tipo primitivo fundamental e serve para definir as variáveis do programa que armazenam as imagens para o processamento. Internamente variáveis imagens são objetos da classe imagem.

3.1.2 Tipos de Agregados

Lista é um tipo composto por um encadeamento de valores de um mesmo tipo. Listas também podem conter outras listas. Podemos ter desta forma, lista de listas em quaisquer dimensões e profundidades, obviamente limitadas pela memória da máquina. Este tipo composto é utilizado para definir constantes e variáveis contendo agregados de qualquer um dos tipos primitivos.

Constantes listas são inicializadas sintaticamente de forma semelhante a inicializadores de estruturas em C. Operações com listas estão limitadas as operações de seu tipo primitivo.

Listas suportam indexação e podem ser de tamanho variável. Um *array* de inteiros, reais, *strings* ou imagens será portanto uma lista de dimensão única com N elementos. Uma matriz é uma lista de duas dimensões de N colunas e M linhas. Lista de tamanho variável tem a primeira de suas dimensões vazia. Um bom exemplo para da utilização de listas é a soma de uma lista de imagens.

3.2 Identificadores

Em SIPL identificadores são nomes utilizados para referenciar variáveis e *scripts*, sendo definidos pelo programador. Um identificador é uma combinação de letras e dígitos, onde o primeiro deles deve ser uma letra. O símbolo sublinha (_) é permitido em um identificador e é considerado uma letra.

3.3 Declaração de Variáveis

As declarações de variáveis em SIPL devem ter seus tipos indicados para que o programa faça corretamente a alocação de espaço de memória requerido, e posteriormente estas informações sejam usada na consistência de tipos de suas atribuições e nas expressões que as utilizam. A declaração de uma variável tem a forma:

Tipo_especificador lista_de_variáveis

Tipo_especificador é um dos quatro tipos primitivos suportados. Lista_de_variáveis é uma lista de identificadores separadas por vírgulas, representando as variáveis do *script*, terminadas por ponto e virgula (;). Por questões de facilidade de leitura cada declaração aparece em linhas separadas. São exemplos de declarações:.

```
INT ivar1, ivar2, array[10];
```

```
REAL var1, matriz[ ][4];
IMAGE img1, limg1[];
STRING nome, arquivos[];
```

As três últimas linha de declarações contém exemplos de listas de tamanho variáveis ou listas livres.

3.4 Expressões em SIPL

A linguagem suporta os operadores aritméticos normais: adição (+), subtração (-), multiplicação (*), e divisão (/). O operador módulo (%) é usado para computar o resto da divisão de dois inteiros. Estes são os operadores binários aritméticos que se aplicam a dois operandos. O operador unário de negação (-) está também disponível e é aplicado para a operação simples de produzir o negativo aritmético de seu operando.

Como nas expressões aritméticas normais, uma expressão em SIPL é avaliada de acordo com a precedência de seus operadores. A precedência ou prioridade de um operador determina a ordem de avaliação de uma expressão aritmética. A tabela 3.1 mostra a ordem de precedência, onde o operador unário menos tem a mais alta precedência enquanto soma e subtração tem a mais baixa e igual precedência. A multiplicação, divisão e o operador módulo tem precedência intermediária.

Uma expressão envolvendo operadores de igual precedência é resolvida de acordo com a referência da coluna associatividade da tabela 3.1. Associatividade refere-se a ordem (ou direção) na qual operadores

possuindo a mesma precedência são executados. Parênteses alteram a ordem de precedência. A expressão 2 + 3 * 4 + 5 é avaliada da seguinte forma. Multiplicação tem a maior precedência dos três operadores, portanto é avaliada primeiro. A expressão agora é reduzida a 2 + 12 + 5. Os dois operadores de adição tem igual precedência e associam da direita para esquerda. Então, 2 e 12 são adicionados primeiro resultando em 14, para em seguida 14 e 5 somarem, produzindo 19 como resultado final.

Se no exemplo acima desejamos executar ambas as somas antes da multiplicação, então é indicado o uso de parênteses envolvendo a sub-expressão. A expressão deverá ser escrita como (2+3) * (4+5), e avaliada como 5 * 9, ou 45.

O operador da divisão funciona normalmente, exceto na situação onde ambos operadores são inteiros, quando o resultado é truncado na sua parte fracionária.

O operador exponencial (^) é utilizado para reais nos moldes da função pow(x, y) em C. A base x não pode ser negativa.

Expressão aritméticas e booleanas são suportadas. Dependendo do tipo primitivo envolvido, algumas operações podem não fazer sentido, tais como multiplicação e divisão de strings. Outras utilizam tipos mistos tais como expressões numéricas contendo inteiros e reais, onde o inteiro é convertido para real durante o cálculo da expressão.

A função "exponenciação" é definida em SIPL como fator de alteração na escala de visualização de imagens, *zoom*, significando que uma imagem com "expoente" 2.0 terá sua escala de visualização dobrada, enquanto que "expoentes" negativos reduzirão esta escala.

Operações com imagens podem envolver inteiro e reais e resultam sempre em novas imagens. Estas operações são realizadas ponto a ponto e o tratamento de exceções é realizado dentro do domínio das cores. Somar imagens significa realizar a soma das cores pixel a pixel. Multiplicar uma imagem por um inteiro fará com que todos os pixels desta imagem sejam multiplicado por este inteiro, alterando portanto as cores desta imagem. Filtros utilizam matrizes de inteiros, portanto a operação de filtragem em SIPL pode se r realizada pela multiplicação de uma imagem por uma lista de inteiros. Apresentamos a seguir trechos de um programa em SIPL para aplicar o filtro laplaciano de ordem 3, mostrado na máscara da figura 3.1:

```
INT   filtro[3][3];
IMAGE img1, img2;
.
filtro := { { {0,1,0}, {1,-4,1}, {0,1,0} } };
img2 := img1 * filtro;
```

0	1	0
1	–4	1
0	1	0

Figura 3.1 Máscara do filtro laplaciano de ordem 3

operador	Descrição	Associatividade
–	Menos Unário	Direita para esquerda
*	Multiplicação	Esquerda para direita
/	Divisão	"
%	Módulo	"
+	Adição	Esquerda para direita
–	Subtração	"

Tabela 3.1 Precedência e Associatividade de operadores

3.5 Operador de Atribuição

Operador de atribuição permite a atribuição de algum valor novo a uma variável do programa. A forma mais simples desta atribuição é:

Variável := expressão;

O efeito do operador é avaliar a expressão à direita do operador (:=) e o valor do resultado é atribuído à variável na esquerda. Se os tipos primitivos numéricos diferem o tipo da expressão é convertido, *cast*, para o tipo da variável antes de executar a atribuição. Se um número real é atribuído a um inteiro sua parte fracionaria é truncada.

Seguem-se exemplos simples usando listas de inteiros com atribuição de valores, operações algébricas, inclusive utilizando-se indexação:

INT arraydgt[10], mat_a[2][3], mat_b[2][3], mlista[2][2][3], list[ ];

atribuindo valores:

```
arraydgt :=  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
mat_a    :=  { { 1, 2 } , { 3, 4 } , { 5, 6 } };
mat_b    :=  { { 7, 2 } , { 8, 3 } , { 9, 4 } };

mlista   := { { { 1, 2, 3 } ,{  4,  5,  6 } },
              { { 7, 8, 9 } ,{ 10, 11, 12 } } };
list     :=  { 7, 2 };  # Um vetor aberto
```

alterando valores por meio de indexação:

```
mlista[1,2,3] := 0;
mlista[1,1]   := { 10, 20, 30 };
```

mlista conterá agora

```
{ { {10, 20, 30}, { 4,  5,  0} },
  { { 7,  8,  9}, {10, 11, 12} } }
```

atribuição com expressão de listas:

mat_a := mat_a + mat_b;

mat_a resultará em { { 8, 4 } , { 11, 7 } , { 14, 10 } }

A linguagem implementa a conversão automática (*cast*), de variáveis inteiras e reais nas operações de atribuição e tem operadores de *append* a listas.

3.6 Declaração de Scripts

Para que um *script* possa referenciar outro *script* é necessário declaração destes durante a seção de declarações. Isto se faz necessário para a alocação de memória para os mesmos na tabela de *scripts* e para que durante a analise do mesmo, sejam realizados o cálculo do endereço de chamada e a verificação de referências inválida.

Sua forma é:

USES lista_de_scripts;

Onde Lista_de_scripts é uma seqüência de *scripts* separados por vírgulas.

3.7 Estrutura de um Programa

Um programa em SIPL é composto por um conjunto de *Script*s. Um *Script* representa o equivalente de uma função em C. *Script*s não podem ser aninhados e têm argumentos passados implicitamente, conforme detalhado mais adiante.

A estrutura de um *script* é composta de um rótulo, uma seção de declarações e um bloco que forma o corpo do *script*, no qual se encontram os comandos e expressões da linguagem. O apêndice C contém alguns exemplos de programas em SIPL, inclusive fazendo referência a outros *scripts* e a utilização de comandos de condição e repetição.

O rótulo é o nome pelo qual um *script* é referenciado no corpo de um

outro *script*. Na seção de declarações é feita a declaração de variáveis e dos *scripts* que são referenciados em seu corpo.

Variáveis são sempre locais e definidas no início do *Script*, não existindo declarações dentro de blocos. Registros de Ativação são construídos dinamicamente, permitindo recursividade. A execução de um programa inicia-se no primeiro *Script* do texto fonte. Todo *Script* retorna implicitamente uma imagem.

O exemplo na figura 3.2 ilustra, um programa para apresentar uma animação de todas as imagens horárias armazenadas em disco no diretório `/usr/images`, com nomes iniciando em **`MXI ou mxi`**, obtidas entre os dias `31/03/97` e `02/04/97` e que estão armazenadas em formato `GIF`. Nas próximas seções será detalhando o preciso funcionamento do *script* `Animator`, contendo outro *script* simples, **`MXIS`**, representado um *maskfile*, definido na seção 3.6, contendo a máscara, o caminho de pesquisa e o tipo de imagem associado a classe de imagens contida na máscara.

```
Script Animator
  uses MXIS;
  begin
    show ($MXIS.31d.3m.97y -> 02d.04m.1997y+1h);
  end.

Script MXIS
  begin
    [Mm][Xx][Ii]yymdd.hhh,  /usr/images,  GIF
  end.
```

Figura 3.2 Um Programa em SIPL

3.8 Comandos Principais

SIPL possui comandos que são comuns a outras linguagens tais como, atribuição, condicional, repetições com estilo ***while...do***, retorno, leitura e armazenamento de imagens, estes suportando vários formatos (BMP, GIF, SunRaster, GRIB, PDUS e PCX). Operadores aritméticos, lógicos e relacionais estão disponíveis. Há outros comandos específicos da linguagem para aplicação às imagens tais como: gerar o *grayscale* [GV94] de uma imagem, ou seja convertê-la em tons de cinza, alterar a escala de visualização da imagem original, extração de cores, valor máximo e mínimo ponto a ponto em uma seqüência de imagens, operações aritméticas e aplicação de filtros passa-alta e passa-baixa em imagens.[2] SIPL suporta agregados de variáveis, assim operações aritméticas em conjunto de variáveis são possíveis, inclusive em agregados de imagens.

Descrição dos principais comandos de manipulação de imagens de SIPL:

- **SHOW** – Apresenta uma imagem ou seqüência de imagens. Se o número de imagens for maior que um faz a animação das imagens.
- **GRAY** – Converte imagens coloridas em tons de cinza.
- **ZOOM** – Altera a escala de visualização das imagens mostradas pelo comando SHOW.

[2] Matrizes de filtragem definidas pelo usuário também são suportadas.

- **EXTR, EXTG** e **EXTB** – Extrai cores de uma imagem. O letra final representam as cores (R)ed para vermelho, (G)reen para verde e (B)lue para azul.
- **MIN** – Retorna a cor mínima de uma seqüência imagem.
- **MAX** – Retorna a cor máxima em uma seqüência de imagens.
- **SRAW, SRAS, SGIF** e **SB** – Salvam imagens nos formatos Raw, SunRaster, Gif e Windows Bitmap respectivamente.
- **RRAW, RGIF, RB, RPCX, RRAS, RGRIB, RPDUS** – Lê imagens nos formatos Raw, Gif, Windows Bitmap, Pcx, SunRaster, Grib e Pdus. Estes dois últimos são imagens com georeferência.
- **REP** - Substituição de tons de cinza por faixas de cores, LUT (Look Up Tables).

A versão inicial de SIPL não suporta definição e chamadas de *Scripts* com declarações e mecanismos de passagem de parâmetros. A chamada de um *Script* por outro supõe passagem e retorno implícito de argumentos. Mesmo sendo uma limitação, esta abordagem representa uma construção bastante característica na área de aplicação da linguagem.

3.9 Especificação Temporal

Foi definido um esquema de recuperação temporal de imagens, doravante chamado de *Maskfile*, que resolve um problema operacional complexo que ocorre em instituições que manipulam grandes quantidades de imagens, cada uma oriunda de uma fonte (satélite, radar, etc.), com determinada periodicidade e de diferentes sensores. A questão é determinar, a partir da especificação da fonte e de um intervalo de datas, que imagens satisfazem a esses critérios.

Tipicamente, existem duas formas clássicas de abordar o problema. A primeira é definir uma base de dados para fazer o mapeamento em questão, seja armazenando as próprias imagens, ou indicando nomes de arquivos que contêm as imagens satisfazendo os critérios de busca. O projeto da base de dados é complexo, pois a forma de especificação do mapeamento é estritamente anômala e heterogênea, não seguindo estruturas comuns de projeto de bases de dados relacionais. Seria necessário, após conseguirmos projetar tal esquema, criar múltiplas chaves de acesso à base, o que apenas muda o foco do problema, dado que, nesse novo cenário, a aplicação deveria descobrir que esquema de indexação usar em cada caso.

A segunda abordagem para a questão é criar um esquema de armazenamento onde a partir do nome do arquivo contendo a imagem seja possível determinar todos os parâmetros de caracterização da referida imagem. Novamente, o problema é que a falta de regularidade dos parâmetros conduz a esquemas esdrúxulos de nomeação e dificultam significativamente o projeto de aplicações. Decorre, então, uma situação

onde cada aplicação implementa sua própria técnica de reconhecimento do mapeamento, aplicável apenas às imagens que ela manipula.

Um *Script* na sua forma mais simples contém apenas um *Maskfile*, representando um conjunto de constantes e variáveis para compor nomes de imagens em disco. *Maskfile* lembra uma expressão regular em que os operadores de concatenação, alternativa e *closures* são adicionados de especificações abertas representando unidades de tempo. Um *Script* que contém um *maskfile* recebe uma data, substitui nas "variáveis" de tempo e recupera a imagem (se existir) do disco, de acordo com o formato e diretório indicados. Veja o segundo *Script* na figura 3.2.

A especificação de um *Maskfile* contém constantes entre colchetes, as quais representam a parte fixa de uma posição no nome da imagem. Várias letras entre os mesmos colchetes indicam alternativas para a posição. No caso do exemplo citado as partes fixas seriam as alternativas iniciadas por "MXI" e "mxi". A parte fixa de uma especificação identifica, normalmente, uma classe de imagens.

Variáveis são caracteres especiais, que representam especificadores de tempo, caracterizando a parte variável do nome. Os especificadores de tempo utilizados são: ano, mês, dia, hora, quinzena, pêntada e semana respectivamente representados por Y, M, D, H, Q, P, W. Dia, quinzena, pêntada e semana são mutuamente exclusivos, ou seja, somente um desses podem coexistir em uma máscara válida.

Existem muitas variações envolvendo cada uma dessas variáveis. Por exemplo, se apenas dois Y aparecem em uma especificação, eles indicam um ano no século corrente; sendo três Y, indicam um ano no atual milênio.

Até quatro Y são aceitos, neste caso com o significado óbvio. Um caso mais esdrúxulo acontece com o mês, e é decorrente de uma prática comum de nomeamento de imagens em várias instituições no Brasil. Se dois M estão presentes, eles indicam uma variação de 01 a 12, mas quando apenas um M está presente ele casa com 1 a 9 para os meses entre janeiro e setembro; para os meses outubro, novembro e dezembro, o mapeamento é com as letras O, N e D, respectivamente. Um outro caso importante é que podemos ter dois a quatro H em uma especificação, onde o terceiro, e eventualmente o quarto, indicam a fração da hora a que se refere a imagem.

Tomemos por base novamente a figura 3.2, cuja especificação temporal é **[Mm][Xx][Ii]YYMDD.HHH**. Sua parte variável significa duas posições para o ano (**YY**), uma posição para o mês (**M**), duas posições para o dia (**DD**) e três posições para a hora e fração (**HHH**). Se invocado com a data `12/04/97-17:50h`, o algoritmo de implementação buscaria em disco as imagens MXI97412.175 e mxi97412.175.

Maskfile resolve tão bem o problema, que foi implementado separado e independente de **SIPL**, para ser um mecanismo de recuperação temporal de imagens.

3.10 Invocação de *Maskfile*

Em SIPL existe um comando genérico *Maskvar* responsável por ativar os *Scripts*, incluindo *Scripts Maskfile*. Cada *Script* recebe implicitamente uma data. *Maskvar*, em princípio, chama um *Script* passando a mesma data que (o *Script* no qual está inserido) recebeu do seu chamador. No caso mais

geral, entretanto, Maskvar se transforma num mecanismo potente de chamada repetitiva de um mesmo *Script*, variando a data para cada chamada. Assim *Maskvar* retorna uma lista de imagens, eventualmente unitária.

Sintaticamente, um comando *Maskvar* é composto por:

- O *Script* a ser chamado
- Uma data opcional passada para a primeira chamada ao *Script*
- Uma data opcional passada para a última chamada ao *Script*
- O incremento de data entre duas chamadas

O *Maskvar* **$MXIS.31d.3m.97y -> 02d.04m.1997y+1h** no programa da figura 3.1 chama repetidamente o *Script* **MXIS**, passando como unidade de tempo cada uma das "datas" entre 31 de março e 02 de abril de 1997, aumentando uma hora em cada chamada. Os argumentos antes e depois do operador -> referem-se às datas de início e fim da varredura e o último argumento ao fator de incremento de tempo. Poderíamos refinar mais (ou menos) cada uma das especificações de data acima, ou seja, seria possível fazer as chamadas iniciando às 31/03/97-18:00h e terminando em 02/04/97-14:00h. Neste caso deveríamos usar:

$MXIS.31d.3m.97y.18h -> 02d.04m.1997y.14h+1h.

3.11 Maskvar Absoluto e Relativo

Cada especificador de tempo em um *Maskvar* é considerado absoluto quando não possui sinal e relativo quando sinalizado. Como cada *Script*

recebe uma data, o especificador absoluto sobrepõe (localmente) o tempo recebido, enquanto o relativo indica um deslocamento em relação a esse tempo. Argumentos de tempo não modificados explicitamente, permanecem com o valor recebido. Se dentro de um *Script*, desejarmos recuperar todas as imagens horárias **MXI** das últimas duas semanas, deveríamos usar o operador *Maskvar*:

$MXIS.-2w+3h.

Este é um exemplo de um *Maskvar* relativo. Supondo que a data do sistema seja 29/01/98, este novo *Maskvar* MXIS ajusta a data inicial em 14/01/98, a data final de varredura é a própria data do sistema, com incremento de 3 horas.

3.12 Outras Utilizações em Recuperações de Imagens

Uma outra forma possível para a utilização deste esquema de recuperação temporal, seria a recuperação de documentos históricos convertidos em imagens, tal como o Projeto Nabuco da UFPe [http://www.ufpe.br/~nabuco]. Muito embora a recuperação de documentos baseada em banco de dados seja mais adequada, devido a utilização de múltiplos índices de pesquisa, como no caso real do Diário Oficial da União[http://www.dou.gov.br/], SIPL poderia resolver esta recuperação de imagens documentais com uma simples inclusão de mais um especificador temporal nas regras de validação de *maskfile*, para permitir a emulação de

páginas distintas de um documento. Especificadores temporais não utilizados na recuperação de imagens de satélites, tal como segundos que tem representação interna em sua classe com um inteiro longo, poderia ser utilizado como um número seqüencial para as páginas deste documento, onde a parte fixa de uma máscara estaria associada ao título do documento. Poderíamos ainda utilizar ou não os outros especificadores de tempo se quiséssemos associar ao documento a data na qual o mesmo fora escrito.

O próximo Capítulo trata da implementação de SIPL. Primeiramente são discutidos os aspectos do *frontend*, com as fases de análises léxica, sintática e semântica. Em seguida é tratado os aspectos de implementação do *backend*, especialmente seu ambiente de execução e interpretador.

Capítulo 4

Aspectos de Projeto e Implementação

SIPL é implementada através de um interpretador e sua interpretação é facilmente justificada. A obtenção das imagens depende fundamentalmente de entrada e saída, uma atividade inerentemente lenta. Operações sobre imagens também são bastante demoradas e consumidoras de CPU, requerendo para isto um runtime de alto desempenho. A combinação destes fatores faz com que o tempo de tradução seja, normalmente, uma pequena parcela do tempo de execução do programa. Um interpretador também se mostra viável, pela maior facilidade de construção de um protótipo e pela maior portabilidade em relação a um compilador propriamente dito.

Operações com imagens na plataforma projetada, fazem uso de uma classe de imagens, que encapsula todas os métodos necessários à manipulação de imagens, bem como garante o desempenho necessário, pois se trata de uma implementação em C++.

As características da linguagem, entretanto, oferecem total possibilidade de implementação através de um compilador. As variáveis e funções tipadas e a uniformidade semântica dos operadores são suficientes para garantir ambas as metodologias de implementação. Na realidade, este interpretador faz toda a fase de análise e geração de código intermediário antes de iniciar a execução, é portanto um interpretador de código

armazenado. Para o armazenamento do código intermediário em disco, a exemplo de Java[App97] que utiliza *bytecode*, poderia ser feito com a implementação de um módulo com tal capacidade de mapeamento, no qual também se daria a transferência de código, em ambas as direções, entre as memórias principal e secundária da máquina. Para a geração de um compilador seria necessário módulos de otimização e geração de código.

O modelo de construção adotado para esta implementação foi o incremental por ser o mais adequado a uma linguagem nova, facilitando seu crescimento e funcionalidade. A cada nova funcionalidade acrescida testes exaustivos foram efetuados, evitando-se dessa forma o acumulo de erros que seriam mais difíceis de retirar em estágios mais avançados.

A maior parte do código desenvolvido para implementar SIPL utiliza o paradigma da orientação a objetos, codificado em C++. Procurou-se usar ferramentas sempre que possível. Assim para análise léxica e sintática foram adotados Flex [Pax90] e Bison [DS94], respectivamente., formando o *frontend* de SIPL conforme sua estrutura representada na Figura 4.1. Os apêndices A e B contém as especificações léxicas e sintáticas utilizadas por estas ferramentas para a construção de seus respectivos analisadores.

Como nas fases de análise não de pode determinar o valor dos operandos nas expressões, a semântica de SIPL faz o tratamento de exceções tais como divisão por zero e verificação de índices de listas fora de limite.

Diversas classes foram projetadas e implementadas para a construção da linguagem. A tabela 4.1 mostra o objetivo de cada classe.

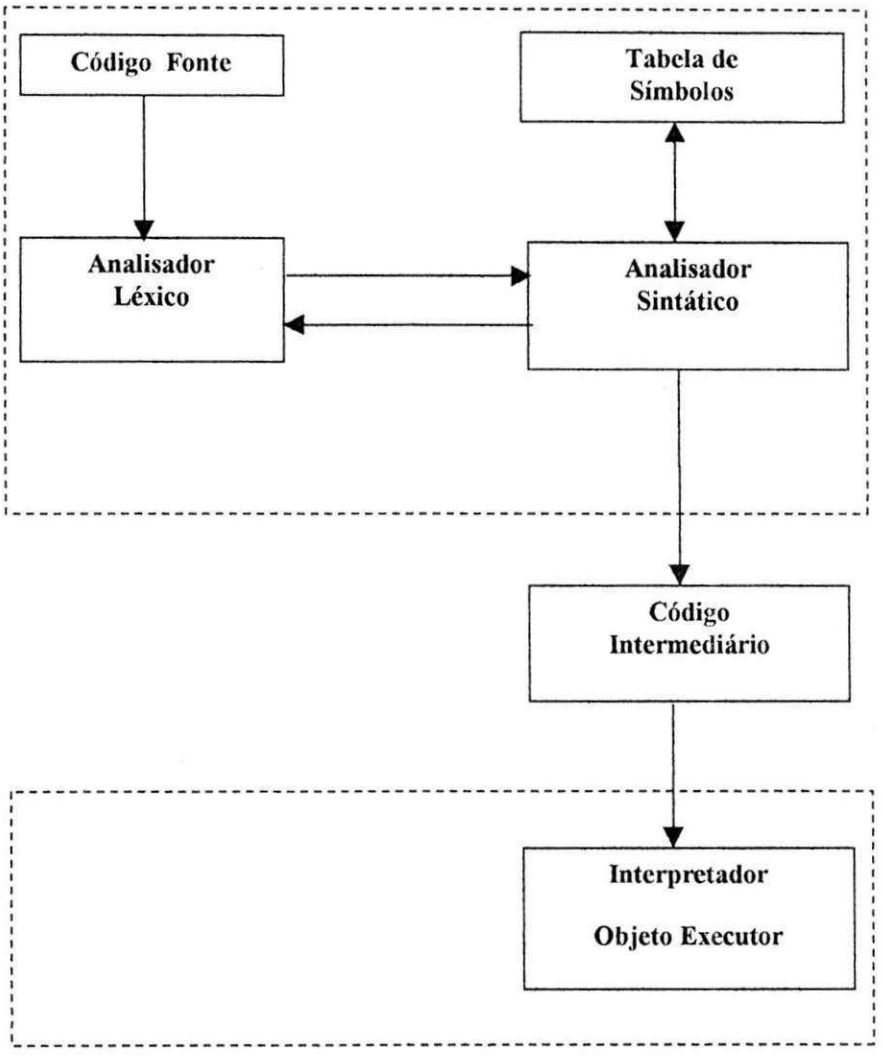


Figura 4.1 Estrutura do Interpretador para SIPL

Classe	Objetivo
`Symbol`	Armazena o nome, escopo léxico, tipo, deslocamento no *frame* de alocação de variáveis do *script*.
`Tabsym`	Implementa tabela de símbolos utilizando *hash*.
`Framvar`	Armazena a variável de acordo com seu tipo.
`Scrframe`	Aloca dinamicamente espaço no *frame* de variáveis do *script*. (*array* de objetos FramVar).
`List`	Implementa a agregação de variáveis, constantes, expressões, e comandos.
`Element`	Contém unidade de um agregado.
`Treenode`	Implementa nós de uma árvore sintática utilizada na representação intermediária da linguagem.
`Script`	Contém elementos para controle, alocação e execução de cada *script*.
`Tabscript`	Implementa tabela de *scripts*.
`IpDate`	Contém elementos relativos ao tempo e manipula variáveis temporais.
`Executor`	Executa *scripts* da linguagem.

Tabela 4.1 Classes implementadas e objetivos

Para o tratamento dos símbolos (identificadores ou variáveis) são utilizadas as classes `Symbol`, `TabSym` e `FramVar`. As classes `Element` e `List`, são a base para a agregação de variáveis, constantes, expressões,

comandos, e que, em conjunto com a classe `TreeNode`, são utilizadas nas árvores da representação intermediária. `TabScript` e `Script` para tratamento dos *scripts*. A classe `IPDate` trata os objetos temporais e por fim, temos a classe Executor, sendo a responsável pela alocação de variáveis, a execução e controle do programa.

4.1 Analisador Léxico

O analisador léxico foi construído a partir de especificações de Expressões Regulares em Flex. Para simplificar a implementação, ainda que a um custo de aumento no tamanho do Autômato Finito gerado, não há uma consulta a uma tabela de palavras-chaves [FH95] a cada reconhecimento de um identificador.

Um problema potencial para programadores SIPL, é que foram usados vários abreviadores ou sinônimos, para alguns elementos léxicos. No capítulo anterior, seções 3.9, 3.10 e 3.11 usamos especificadores de tempo associados com **D, M** e **W**. Estes símbolos são abreviações léxicas de **DAY, MONTH** e **WEEK**, respectivamente. Objetivando a simplificação de *Maskvar* parece razoável manter as abreviações, considerando-as como palavras reservadas da linguagem.

O analisador léxico ignora a distinção entre caracteres maiúsculos e minúsculos e mantém variáveis de controle de posição de *tokens* para informação sobre erros.

4.2 Analisador Sintático

A gramática de SIPL é LALR(1) e, portanto, adequada a uso de Bison sem quaisquer modificações, exceto por ambigüidades que são implicitamente corrigidas pelo comportamento padrão do Bison. O esquema de tradução é dirigido pela sintaxe e em um único passo. Os identificadores são incluídos na tabela de símbolos com informações de seu escopo e tipo nesta fase.

Ações semânticas são executadas após o reconhecimento de cada sentença, seguindo o estilo de compilação imposto por ferramentas LALR. As verificações semânticas se estendem desde a consistência de tipos primitivos ou agregados em listas, até a tipagem de argumentos dos comandos e operadores. Uma parte importante dessas ações diz respeito à coerência semântica de *Maskfiles*. Na ocorrência de erro a análise para e uma mensagem descreve o tipo de erro e sua localização, usando variáveis de posição oriundas do analisador léxico.

A técnica de tradução usada por Bison é ascendente e baseada na utilização de atributos sintetizados na pilha do analisador [FB88]. Para evitar o uso de atributos herdados, a semântica utiliza listas temporárias de identificadores para a atribuição de seus tipos. Um aspecto interessante é que o mesmo esquema de suporte a listas para a linguagem (implementado através de uma classe C++) pôde ser usado na sua implementação, que será discutido com maiores detalhes na seção 4.6.

A análise sintática e semântica de um programa é feita uma única vez, gerando uma representação intermediária que é submetida ao módulo interpretador.

4.3 Verificação de Tipos e Compatibilidade de Operações

Para suportar conversão de tipos(*cast*), operações com listas e imagens a verificação de tipos é um ponto chave da implementação. Indexação a torna ainda mais complexa, pois além de verificarmos a compatibilidade dos tipos envolvidos, devemos verificar se o tipo da estrutura do dado referenciado pelo índice de um símbolo corresponde ao mesmo tipo de estrutura em uma expressão. O exemplo seguinte esclarece este caso de verificação.

Suponhamos a declaração de uma lista de inteiros a qual é atribuída posteriormente os valores 1, 2, 3, 4, 5, 6, 7 e 8 :

- `INT LIST[2][2][2];`
- `LIST := {{{1,2}, {3,4}}, {{5,6}, {7,8}}};`

Portanto as referências abaixo são:

- `LIST[1,2,2]` é do tipo inteiro e contém o valor 4
- `LIST[2,2]` é do tipo lista de inteiros, contendo os valores 7 e 8
- `LIST[1]` é uma lista de inteiros contendo os valores 1, 2, 3 e 4

então atribuindo corretamente novos valores:

- `LIST[2,2] := {700,800};`
- `LIST[1,2,2] := 400;`

s \ e	inteiro	real	string	imagem
inteiro	✓	to-int		
real	to-real	✓		
string			✓	
imagem				✓

Tabela 4.2 Atribuições permitidas símbolo (s) := expressão (e)

Nas atribuições com listas são necessários que ambos tipos primitivos sejam iguais e que as dimensões de ambos os lados referenciem o mesmo tipo de dado. Um símbolo sem índice referencia sua dimensão inteira. Conversão de tipos de real para inteiro e de inteiro para real estão indicadas acima por to-int e to-real respectivamente.

Compatibilidade de tipos nas operações algébricas

A ordem dos operandos não pode ser invertida nas tabelas abaixo, significando que a troca dos operando quando não explicitada, indica operação inválida.

Adição

todos os tipos primitivos (inteiro, real, string e imagem)
Inteiro + real (converte operando inteiro para real)
Real + inteiro (converte operando inteiro para real)
Imagem + inteiro (operação resulta em imagem) *

* a operação desloca domínio de cores em direção ao branco.

Subtração

Inteiro
Real
Imagem
Inteiro – real (converte segundo operando)
Real – inteiro (converte segundo operando)
Imagem – inteiro (operação resulta em imagem) *

* a operação desloca domínio de cores em direção ao preto.

Multiplicação e divisão

Inteiro
Real
Inteiro (* | /) real (converte segundo operando)
Real (* | /) inteiro (converte segundo operando)
Imagem (* | /) inteiro (operação resulta em imagem) *

* a operação de filtragem da imagem.

Módulo

válida somente para inteiros.

Exponenciação

válida somente para reais e imagens.

base deve ser real e positiva ou imagem.

expoente deve ser real.

Em SIPL a operação "exponenciação" de imagens é utilizada para alterar o fator de escala de apresentação das mesmas.

Operadores | e & binários

Operando1 deve ser imagem.

Operando2 pode ser imagem ou inteiro.

Operações Booleanas

Operandos devem ter tipos primitivos iguais.

Append a Listas

Símbolo deve ser lista aberta, índice nulo. Tipo da estrutura e dimensões são verificadas.

4.4 Representação Intermediária

A representação intermediária da linguagem é feita com a utilização de árvores sintáticas nos moldes clássicos, acrescidos de nós contendo listas. Os operadores e palavras reservadas são nós intermediários representando sub-expressões, comandos e argumentos [ASU86].

Novamente foi utilizado o mesmo projeto de listas para montagem de árvores. Para isto SIPL suporta listas onde cada elemento pode ser um comando, constituindo-se no agregador de comandos, ou seja o comando composto ou bloco de comandos. Dessa forma torna-se trivial a construção da árvore. Devido a carga semântica da linguagem, espera-se que os programas e, consequentemente, as representações intermediárias equivalentes tenham tamanhos relativamente pequenos, podendo estas permanecerem na memória para execução.

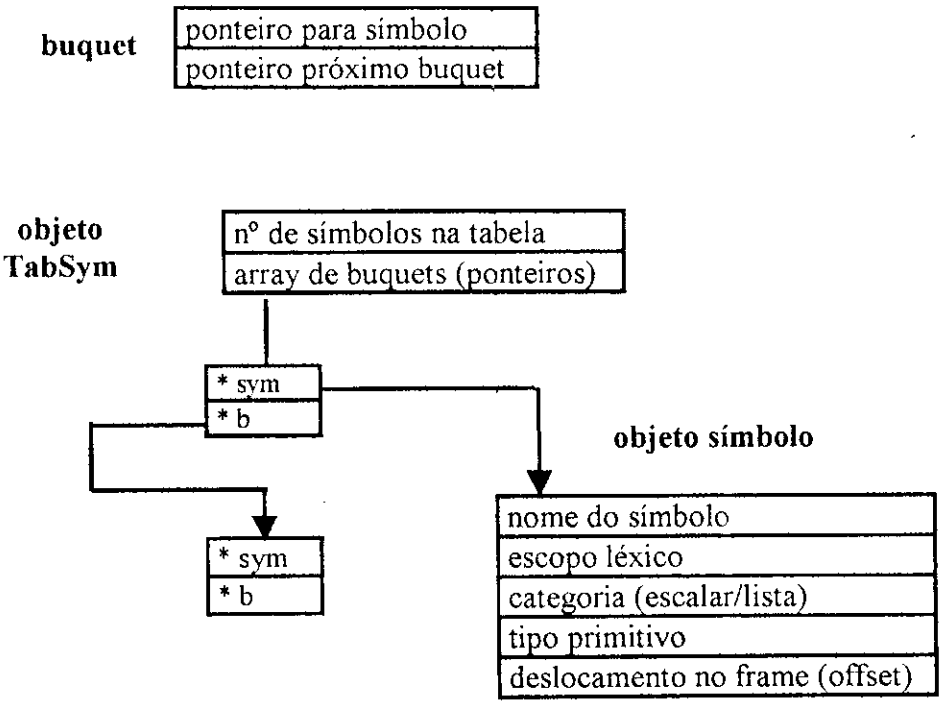
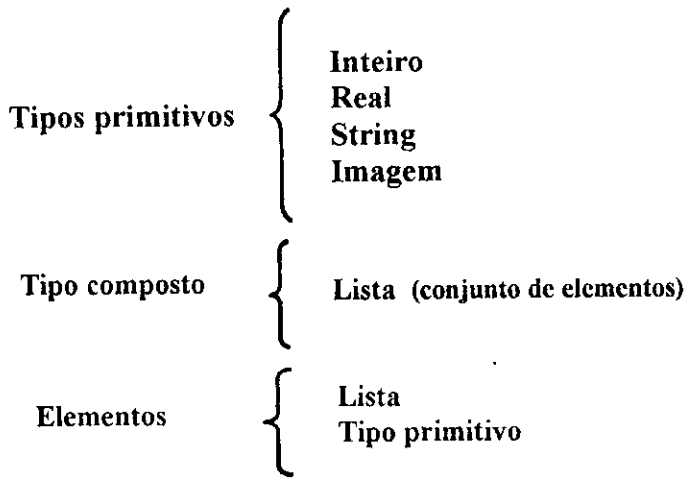


Figura 4.2 **Símbolos**



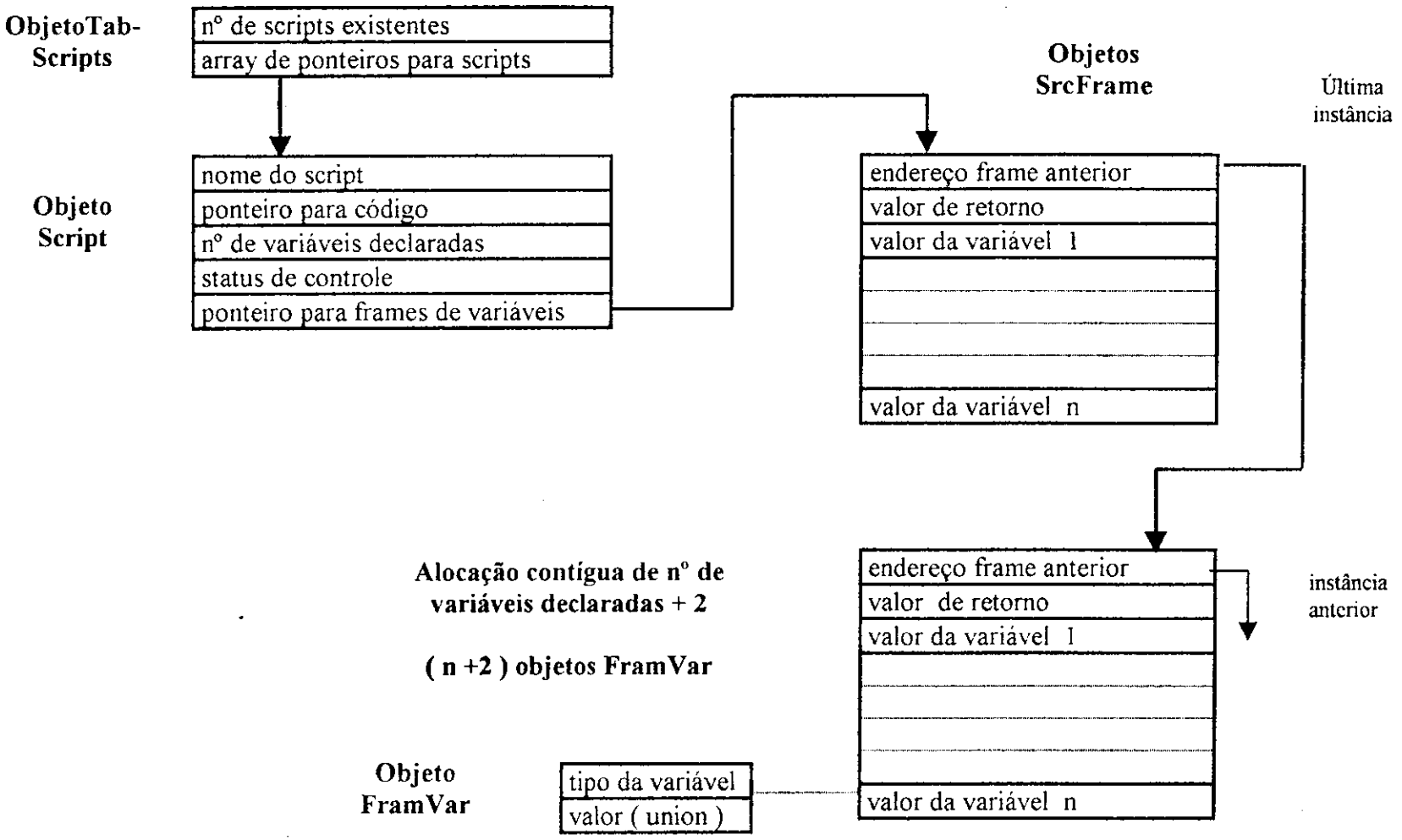


Figura 4.3 Alocação dinâmica de variáveis com registros de ativação

4.6 Projeto de listas

Listas são estruturas dinâmicas de armazenamento de dados. O conceito abstrato de uma lista é geralmente implementado por uma lista de nós, onde cada elemento da lista abstrata corresponde a um nó da lista ligada. Cada nó contém um campo *info* e um apontador para o próximo nó, *next*. Estes conceitos abstratos de uma "lista" e um "elemento" são ambos representados por apontadores. Foram analisados diferentes métodos propostos para a implementação de listas gerais [TLA89]. O *método do apontador* permite criar listas recursivas. O *método da cópia* garante que um nó apareça apenas em um contexto. No *método do cabeçalho*, o mais utilizado, um nó de cabeçalho é posicionado no inicio de todo grupo de nós, sendo semelhante ao método do apontador no sentido que uma lista é representada por um apontador para ela.

A implementação de listas em SIPL deriva deste último método, onde um objeto lista encabeça sempre um grupo de nós, ou seja um grupo de objetos elemento. Como as listas de SIPL necessitam suportar vários tipos, em um objeto elemento o campo *info* é substituído por uma estrutura *tipo-valor*, onde valor é uma união representando os diferentes tipos suportados, e tipo é o indicador do valor utilizado na união. As figuras seguintes 4.4 e 4.5 mostram as representações destes objetos.

Objeto Elemento

tipo do dado
união valor
apontador para próximo

Figura 4.4 Objeto Elemento

Objeto Lista

tamanho da lista
vínculo a símbolo (true/false)
apontador 1° elemento da lista
apontador último elemento da lista
apontador elemento corrente

Figura 4.5 Objeto Lista

A utilização de apontadores auxiliares internos no objeto lista, evita o uso de apontadores externos quando elementos desta lista precisam ser obtidos seqüencialmente, permitindo a implementação de métodos tais como `getfirst` e `getnext`, que garantem o isolamento requerido pela programação orientada a objetos.

Programando em SIPL só podemos declarar listas de tipos primitivos, mas a implementação da linguagem usa maciçamente listas de outros elementos, que tanto são utilizados em sua construção como também em sua execução. Boa parte do processo semântico refere-se a manuseio de listas.

Listas simples se constituem em uma seqüência e nesta noção de seqüência, listas também são utilizadas no código intermediário gerado. O comando composto, que é uma seqüência de comandos, é na realidade uma lista de árvores, onde cada elemento da lista representa estes comandos, que por sua vez são árvores, as quais contém o código respectivo de cada comando. Um programa em SIPL é uma seqüência de *scripts*, portanto uma lista de *scripts*, onde cada elemento desta lista contém a árvore de código para cada *script*. Assim, temos, internamente, lista de variáveis, lista de expressões e lista de comandos, *scripts*, entre outras.

Listas em SIPL não são usadas somente para agregação genérica, operadores aritméticos foram também sobrecarregados em listas e seus constituintes. Obviamente que operações com lista dependem de seu tipo primitivo, portanto estas operações estão restritas às mesmas regras aplicadas aos tipos primitivos que a compõe. Listas também suportam indexação. Um índice de uma variável é também uma lista contendo zero

ou mais constantes inteiras. Um índice contendo uma lista vazia representa listas abertas, ou seja sua dimensão não é fixa. Somente uma dimensão não fixa é permitida para um símbolo contendo uma lista e esta deve ser sempre a primeira das dimensões. Como podemos notar esta implementação é bastante complexa, principalmente a verificação de compatibilidade em expressões, pois operações com imagens permitem combinação de tipos heterogêneos tais como, imagem + inteiro e imagem **x** inteiro. A definição desta classe genérica de manipulação de listas é um ponto chave na implementação desta linguagem.

A figura 4.6 representa uma lista de três dimensões, L[3][2][2], onde o quadrado e círculo simbolizam os objetos lista e elemento respectivamente.

Na figura 4.7 mostra a representação intermediária do comando de atribuição, o qual pode ser um elemento de uma lista comandos que representa um bloco ou comando composto.

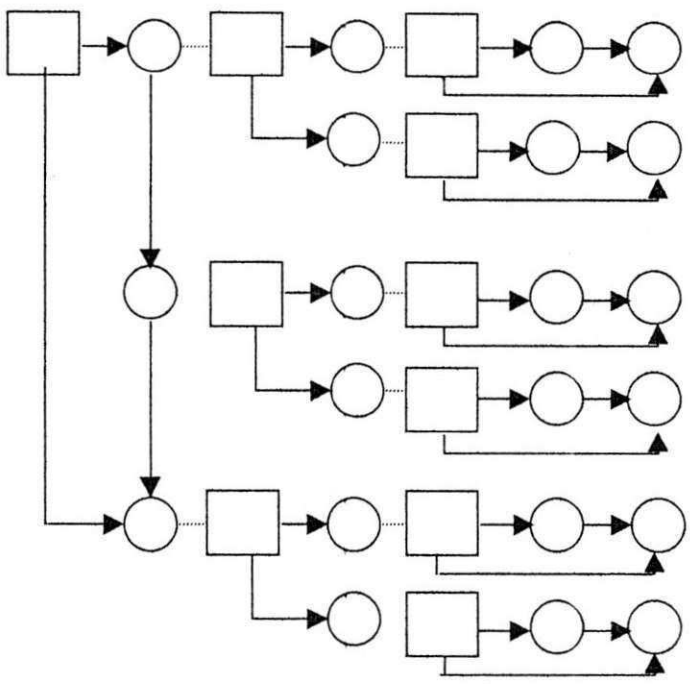


Figura 4.6 Representação de Lista [3][2][2]

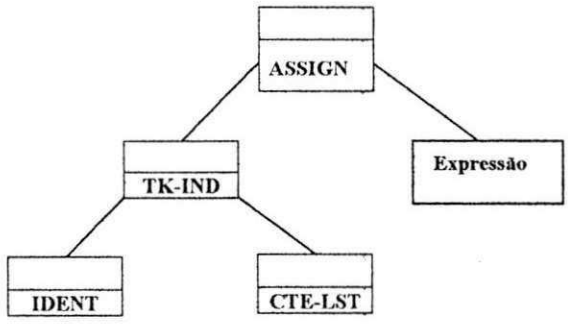


Figura 4.7 Representação intermediária de um comando

4.7 Fase de execução

O interpretador é implementado por um objeto da classe Executor. A estrutura do programa é representada por uma tabela de *Scripts*. Uma entrada nesta tabela contém o nome do *Script*,[3] a representação intermediária dos seus comandos, o número de variáveis definidas e um apontador para o Registro de Ativação (conforme explicado na próxima seção).

A representação de um *Script* é uma lista de comandos, onde cada comando, eventualmente, pode ter pendurado a si, outra lista de comandos, criando a noção de uma árvore. A execução é uma travessia dessa árvore guiada pela estrutura lógica do programa. A figura 4.9 mostra a representação intermediária de um programa completo.

A estrutura é tal que um *loop*, por exemplo, tem pendurado a si, dois outros comandos, um representando uma expressão lógica e o outro um comando composto. Este por sua vez contém uma lista de comandos. Ou seja, usando recursividade na classe Executor, a interpretação fica bastante simplificada.

4.8 Ambiente de Execução

Os endereços de memória para as variáveis são alocados dinamicamente durante a execução de um *Script*. Para cada *Script*, o analisador semântico gera um número seqüencial distinto, denominado *Script_number*. Dentro de

[3] *Scripts* e Variáveis estão em espaços de nomes distintos.

um *Script*, um outro número é associado a cada variável. Trata-se do *var_number*. Todos os endereços de variáveis são assim formados por um par: (*Script_number*, *var_number*), registrado na tabela de símbolos.

Ao iniciar a execução de um *Script* é criado um Registro de Ativação (RA) para a instância (um RA contém os seguintes campos: um apontador para o RA anterior associado ao *Script*, uma entrada para cada variável e mais uma para a valor a ser retornado pelo *Script*). O módulo Executor, então, armazena o RA atualmente na tabela de *Scripts* em um campo do RA recém criado, estabelecendo este como o novo RA da tabela de *Scripts*. O processo inverso acontece durante o retorno.

O acesso a variáveis é então feito indiretamente através da indexação na tabela de *Scripts*, usando *Script_number*, deslocado por *var_number*. Como o RA na tabela de *Scripts* varia durante a execução do programa, esta abordagem garante o suporte a recursividade.

4.9 Suporte a Datas

Especificadores de tempo representam as diferentes unidades de tempo para a composição de uma data. Foi implementada uma classe para dar suporte e permitir operações com data. A figura 4.8 representa um objeto para data.

Esta classe, IPDate, tem operadores aritméticos e lógicos, suporta operações sobrecarregadas sobre as várias unidades de tempo, incluindo, segundos, hora, dia, mês, ano, semana, quinzena e pêntada. Intervalos de tempo decorrido entre frações de datas ou datas pode ser obtido, bem como conversões entre os calendários gregoriano e juliano e outros métodos que suportam incremento e decremento de unidades temporais. A

implementação de IPDate foi simplificada pela retirada dos ajustes de calendário realizados anteriores ao século 18 (baseia-se no Calendário Gregoriano), consistindo e validando datas somente de 1° de janeiro do ano de 1800 em diante. Trata também data de dois dígitos pertencentes a séculos distintos, estando preparada para a virada do milênio (*bug* do ano 2000) mesmo com a utilização de anos com somente dois dígitos.

4.10 Implementação de *Maskvar* e *Maskfile*

A implementação de *Maskvar* demanda a existência da classe IPDate para a manipulação de datas. Esta classe encapsula toda a atividade requerida por *maskvar*. Um objeto desta classe, denominada IPDate, figura 4.8, é passado implicitamente como parâmetro para todo *Script*, exceto possivelmente o *Script* inicial, que neste caso usa a data do sistema.

Com IPDate a implementação de *maskvar* é feita apenas através da variação da data usando os operadores dessa classe.

A implementação de *maskfile* é simplesmente a captura dos parâmetros em aberto na especificação usando o objeto IPDate recebido. Feito isto temos um nome de imagem a ser procurado em disco, o que é feito através de rotinas do sistema operacional hospedeiro.

Objeto IPDate

dia
mês
ano
hora
minuto
segundo
tipo-data

Figura 4.8 Objeto Data

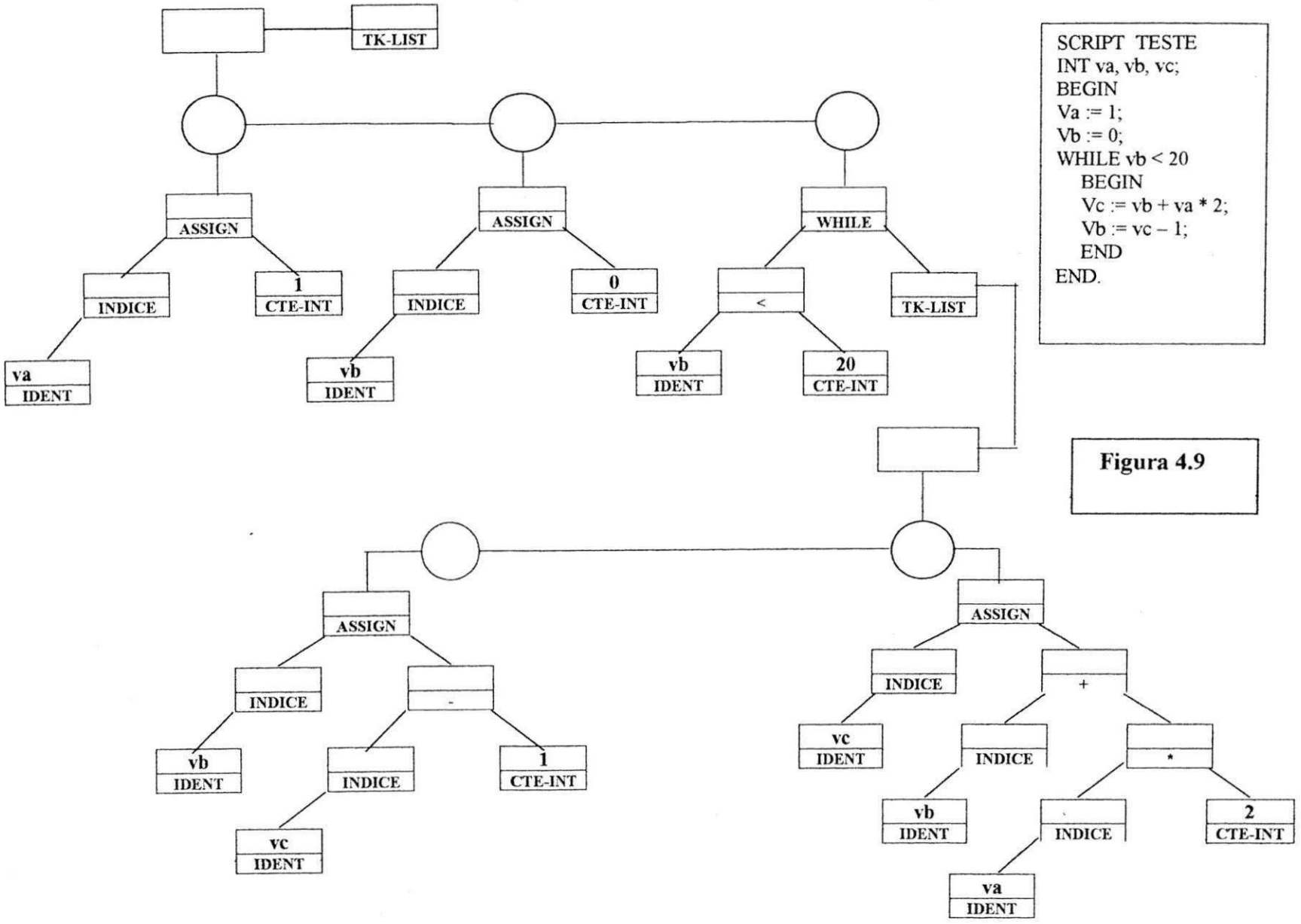

TK-LIST
ASSIGN
INDICE
va
IDENT
1
CTE-INT
ASSIGN
INDICE
vb
IDENT
0
CTE-INT
WHILE
<
vb
IDENT
20
CTE-INT
TK-LIST
SCRIPT TESTE
INT va, vb, vc;
BEGIN
Va := 1;
Vb := 0;
WHILE vb < 20
BEGIN
Vc := vb + va * 2;
Vb := vc – 1;
END
END.
ASSIGN
INDICE
vb
IDENT
-
INDICE
vc
IDENT
1
CTE-INT
ASSIGN
INDICE
vc
IDENT
+
INDICE
vb
IDENT
*
INDICE
va
IDENT
2
CTE-INT

Figura 4.9

4.11 Interface Gráfica

Adotando mais uma vez a filosofia de hardware de baixo custo e software padrão, esta interface foi implementada utilizando classes codificadas em C++ com chamadas direta a funções em *xlib*, permitindo a execução em qualquer ambiente gráfico do *Unix* através de um método que faz a conecção com o servidor gráfico utilizado no ambiente.

Dentro de várias tentativas testadas, o ambiente gráfico do RUNTIME preexistente[SM96], foi alterado para suportar ambientes gráficos do *Unix*, com a inclusão de novas classes implementadas diretamente em *Xlib*, tomando como base o RUNTIME preexistente e XCLASSES servindo de modelo de chamadas diretas a *Xlib*. Chegaram a ser construídas as classes `Xwindow`, `Xframe`, `Xcanvas`, `Xbutton`, `Xedit` e `Xtext`, utilizando-se uma hierarquia simples. Entretanto a imaturidade do código criado comparado a XCLASSES e o esforço de codificação requerido para criar uma funcionalidade e interação harmoniosa entre os diferentes objetos necessários à interface gráfica, provocaram seu abandono.

A tentativa de utilização de ambientes gráficos do *Unix*, tal como *Motif*[OSF90], também foi abandonada. *Motif* está implementado em C e a utilização de suas bibliotecas[Dou95] na codificação em C++ mostrou-se bastante complexa para utilização na construção da interface gráfica, além do que a utilização direta de Xlib reduziria o *overhead* de se encapsular funções do Motif.

A disponibilidade dos objetos gráficos de XCLASSES, que é implementada em C++ utilizando diretamente *Xlib*, juntamente com a funcionalidade de seus objetos e a depuração seu código, finalizaram a

escolha adequada para a implementação da interface gráfica de SIPL.

A interface final utiliza objetos das classes `XcFrame`, `LgCanvas`, `LgEditLut`, `XcPanel`, `XcButton`, `XcEdit` e `XcText`.

Entretanto implementar um editor de texto diretamente em Xlib é uma tarefa altamente complexa, de forma que este software, que não tem o processamento pesado das imagens e é executado de forma isolada, foi implementado utilizando-se objetos da Interface *Motif*, disponível em praticamente todas as plataformas Unix. Da forma como este editor foi implementado, permite que o usuário possa utilizar o editor de sua preferência, que é carregado através da chamada a função *system* da biblioteca C padrão, com o nome de Sipledit.

As classes utilizadas para a interface foram:

- Xcframe – Classe responsável pela criação da janela *Top Level* da aplicação. É derivada da classe XcOpaque que é a classe base de janelas gráficas. É a responsável pela inicialização do ambiente gráfico, fazendo a conexão com o servidor gráfico, *X-Server*, utilizado. Em termos de decoração da janela, a aparência vai depender do gerenciador de janelas em uso.
- LgCanvas – Classe de janelas gráficas derivada das classes XcFrameCmd e XcDraw. Possui internamente uma área de armazenamento de um conjunto de imagens. Seus objeto define uma região de desenho é o responsável pela exibição das imagens.
- LgEditLut – Classe que implementa a função transferência, usada na colorização das imagens mostradas no objeto LgCanvas.

- XcPannel – Classe de janela com a função de agrupar objetos gráficos de controle tais como botões, caixas de textos, etc.
- XcButton – Classe de botões contendo um rótulo, em que cada objeto tem procedimentos associados, sendo responsável por determinada ação executada pelo usuário de forma iterativa. A interface de Sipl utiliza vários destes objetos.
- XcEdit – Classe de caixa de edição, cada objeto recebe um texto digitado pelo usuário. Um objeto XcEdit recebe o nome do script (programa) a ser executado por SIPL.
- XcText – Classe de caixa de apresentação de texto. Um objeto XcText é utilizado no rodapé, *footer*, da janela principal para apresentar mensagens ao usuário, tais como mensagens de erro.

Capítulo 5

Conclusões e Trabalhos Futuros

A importância desta plataforma pode ser traduzida por trazer para o hardware de baixo custo uma linguagem voltada ao Processamento Digital de Imagens, especialmente quando grandes volumes de dados estão envolvidos.

O difícil problema da recuperação de imagens em instituições que utilizam maciçamente imagens capturadas de satélites, é em muito atenuado com a utilização da recuperação temporal de imagens proposta.

A disponibilidade da linguagem que se propôs desenvolver justifica-se por vários aspectos, alguns de ordem prática como a produtividade e destinada ao usuário final, outros de cunho investigativo. Dessa última visão o trabalho serve, por exemplo, como base para pesquisas sobre derivação de arquiteturas específicas programadas em linguagens de quarta geração.

5.1 Revisão dos Objetivos e como foram alcançados

Um projeto complexo como este, envolvendo duas áreas que requerem grande esforço de codificação como compiladores e processamento de

imagens não seria possível se, grande parte do *runtime* já não estivesse definido e codificado[SM96], embora a interface gráfica original fosse o *MS-Windows*. MEDx [http://www-mrips.od.nih.gov/uguide/ugtoc.htm] é um exemplo desta complexidade, seu desenvolvimento em um projeto modular, levou cerca de cinco anos de trabalho de uma equipe formada por programadores especializados em processamento de images.

Para sua implementação diversas ferramentas foram necessárias: gcc, g++, xgbd, Emacs, Bison, Flex, e diversos utilitários do sistema operacional UNIX de apoio a programação. Por reunir tudo isso e executar sob o hardware padrão mais utilizado na atualidade, o Linux foi o sistema operacional escolhido para o desenvolvimento. Foi instalada e adotada a interface gráfica *Motif*[OSF90] para completar o ambiente de desenvolvimento.

Embora operações com imagens ainda sejam poucas, os objetivos propostos foram alcançados, exceto a investigação sobre tecnologias de ligação dinâmica, visando a inserção de módulos reconhecidos pelo sistema, ao que se tem atribuído o jargão de tecnologia *plug and play*. O tempo necessário para a inclusão e conclusão deste trabalho excederia em muito o tempo máximo disponível para uma tese de mestrado.

Houve mudanças no projeto inicial, que não contemplava a utilização de variáveis dinâmicas. Estas foram implementadas visando uma maior flexibilidade na linguagem, permitido recursividade. Realizar esta mudança após a implementação de agregados constituiu-se em uma tarefa um pouco árdua, pois o código que faz a verificação da consistência na indexação de listas, é um código longo e complexo, e teve que mudar de classe. A

implementação de agregados para suporte a animação de conjuntos de imagens não só foi implementada, como também estendida, visto que a linguagem usa maciçamente listas em sua própria implementação, e foi a que demandou maior esforço na codificação.

5.2 Contribuições

Uma linguagem com uma elevada carga semântica, que tem a imagem como seu principal tipo de dado, operadores algébricos para sua manipulação da forma mais conveniente para um pesquisador, proporcionará uma alta produtividade e apresentará vantagens em pesquisas em relação a outras linguagens que demandem alta especialização.

A possibilidade de tratar imagens com filtros definidos pelo usuário, filtragens múltiplas em várias faixas, torna ainda maior o interesse por SIPL. A pesquisa de imagens de satélites meteorológicos está voltada para a investigação de muitos dos fenômenos meteorológicos causadores de catástrofes, tais como El Niño, furacões e outros, que ainda precisam ser melhor estudados, visando descobrir suas ações geradoras e, se possível, determinar formas de serem evitados.

Utilizando-se imagens de satélites de altas resoluções, combinações de imagens de diferentes canais, seguindo padrões pesquisados e pré determinados, podem ser utilizadas para a visualização de alterações no meio ambiente, na vegetação e até mesmo determinar seu tipo. Este exemplo seria valioso para investigações utilizadas no combate às drogas.

A recuperação temporal de imagens é uma característica que não encontramos nas linguagens e softwares utilizados em PDI, de forma que se constitui em uma inovação relevante introduzida por este trabalho.

5.3 Trabalhos Futuros

A disponibilização de SIPL em domínio público para que usuários possam avaliar, sugerir alterações e, principalmente, indicar meios de aumentar o suporte a operações com imagens, resultará em melhorias e aumento da aplicabilidade. Pensa-se também, que a introdução da passagem de parâmetros em *Scripts* seja inevitável, nesse caso seria também razoável pensar em uma maneira de ativação de subrotinas em outras linguagens.

Mecanismos de otimização local de código já estão sendo gradativamente introduzidos na implementação de SIPL, mas devem ser melhorados. Isso é particularmente importante porque o custo de realizar operações desnecessárias sobre imagens é muito maior que sobre escalares.

O código de SIPL é altamente portável por não incluir aspectos particulares de C++ para o ambiente em que foi desenvolvido. Por outro lado, a interface de usuário só pode ser facilmente recompilada em ambiente Unix-like. Uma das maneiras de evitar o problema é reimplementar em Java e usar a ponte Java-C-C++ para garantir portabilidade total entre plataformas.

Apêndice A - Especificações Léxicas da Linguagem SIPL

```
DIGIT           [0-9]

ID              [a-zA-Z_][a-zA-Z_0-9]*

{DIGIT}+        { yylval.ival = atoi(yytext);
                  return CTE_INT; }

{DIGIT}+"."{DIGIT}*
                {
                  yylval.fval = atof(yytext);
                  return CTE_FLT;
                }

"IMAGE"           return IMAGE;
"INT"             return INTEGER;
"REAL"            return FLOAT;
"STRING"          return STRING;
"USES"            return USES;

"SCRIPT"          return SCRIPT;
"BEGIN"           return START;
"END"             return END;
"WHILE"           return WHILE;
"TO"              return TO;
"DO"              return DO;
"FOR"             return FOR;
"IN"              return IN;
"IF"              return IF;
"THEN"            return THEN;
"ELSE"            return ELSE;
"RETURN"          return RETURN;
```

```
/* especificadores de tempo */

"HOUR"     |
"H"              { yylval.ival = 0;
                   return HOUR;      }

"DAY"      |
"D"              { yylval.ival = 1;
                   return DAY;       }

"WEEK"     |
"W"              { yylval.ival = 2;
                   return WEEK;      }

"PENTAD"   |
"P"              { yylval.ival = 3;
                   return PENTAD;    }

"FIFTEEN"  |
"F"        |
"Q"              { yylval.ival = 4;
                   return FIFTEEN;   }

"MONTH"    |
"M"              { yylval.ival = 5;
                   return MONTH;     }

"YEAR"     |
"Y"              { yylval.ival = 6;
                   return YEAR;      }

"->"               return UNTIL;

/* I-O de imagens */

"SHOW"             return SHOW;

"SAVERAW"  |
```

```
"SR"                return SRAW;

"SAVEGIF" |
"SG"                return SGIF;

"SAVEBMP" |
"SB"                return SBMP;

"SAVERAS" |
"SS"                return SRAS;

"READRAW" |
"RR"                return RRAW;

"READGIF" |
"RGIF"              return RGIF;

"READBMP" |
"RB"                return RBMP;

"READPCX" |
"RPCX"              return RPCX;

"READRAS" |
"RS"                return RRAS;

"READGRIB" |
"RG"                return RGRIB;

"READPDUS" |
"RP"                return RPDUS;

/* operações com imagens */

"MIN"               return MINV;
"MAX"               return MAXV;
"GETVAL"            return GVAL;
"RGB"               return RGB;
```

```
"EXTR"              return EXTR;
"EXTG"              return EXTG;
"EXTB"              return EXTB;
"BINARY"   |
"BIN"               return BIN;

"ZOOM"              return ZOOM;
"SUM"               return SUM;
"REGION"            return REG;
"REPLACE"           return REP;
"EXPAND"            return EXPAND;

/* formatos de imagens */

"BMP"           { yylval.ival = 0;
                  return FORMAT;    }

"GIF"           { yylval.ival = 1;
                  return FORMAT;    }

"PCX"           { yylval.ival = 2;
                  return FORMAT;    }

"RAS"           { yylval.ival = 3;
                  return FORMAT;    }

"GRIB"          { yylval.ival = 4;
                  return FORMAT;    }

"PDUS"          { yylval.ival = 5;
                  return FORMAT;    }

"RAW"           { yylval.ival = 7;
                  return FORMAT;    }

{ID}            { strcpy (yyident, yytext);
                  yylval.charptr = strdup(yyident);
                  return IDENT;     }
```

```
"$"{ID}          { strcpy (yyident, &yytext[1]);
                   yylval.charptr = strdup(yyident);
                   return MSKVAR;   }

\"[^\"]*\"       { strcpy (yyctech, yytext);
                   yylval.charptr = strdup(yyctech);
                   return CTE_STR;  }
/* operadores */

"AND"              return AND;
"OR"               return OR;
"NOT"              return NOT;

">"                return GT;
">="               return GE;
"<"                return LT;
"<="               return LE;
"="                return EQUAL;

":="               return ASSIGN;

"+"                return PLUS;
"-"                return MINUS;
"*"                return TIMES;
"/"                return DIV;
"%"                return MOD;
"^"                return POWER;
"<<"               return APPEND;

"("                return OPENP;
")"                return CLOSEP;
"{"                return OPENKEY;
"}"                return CLOSEKEY;
"["                return OPENB;
"]"                return CLOSEB;
";"                return SEMICOL;
","                return COMMA;
```

```
\.                  return POINT;


"#"[^\n]*          /* Eat comments */

[\n]               { vsl_line++; }
[\r]
[ \t]+
.                   return ERROR;
```

Apêndice B - Especificações Sintáticas da Linguagem SIPL

Nas descrições abaixo { } significa uma ação semântica diferente da ação semântica implícita de Bison $$ = $1.

```
%token SCRIPT START   END     WHILE    DO       FOR
%token IF     ELSE    THEN    RETURN   TO       IN
%token SHOW   BIN     ZOOM    REG      ERROR    REP
%token USES   EXTR    EXTG    EXTB     MINV     MAXV
%token SRAW   SGIF    SBMP    SRAS     RRAW     RGIF
%token RBMP   RPCX    RRAS    RGRIB    RPDUS    SUM
%token RGB    UNTIL   GVAL    PLUS     MINUS    TIMES
%token DIV    MOD     POWER   UMINUS   EQUAL    GT
%token GE     LT      LE      NE       AND      OR
%token NOT    OPENP   CLOSEP  OPENKEY  CLOSEKEY OPENB
%token CLOSEB SEMICOL TK_MOD  COMMA    POINT    INTEGER
%token FLOAT  IMAGE   STRING  CTE_LIST TK_LIST  TK_IND
%token TK_CMD TK_MODS TO_INT  TO_FLT   TO_IMG   MSKFILE
%token   <ival>     CTE_INT FORMAT HOUR DAY WEEK PENTAD
%token   <ival>     FIFTEEN MONTH  YEAR
%token   <fval>     CTE_FLT
%token   <charptr>  IDENT CTE_STR MSKVAR
%token   <symptr>   ASSIGN APPEND
%type   <ival>     types signal index time_esp
%type   <listptr>  typelist Lof_ident Lof_cmds Lof_masks
%type   <Node>     exp term power sfactor factor variable
%type   <Node>     maskvar body_script  maskfile cmask
%type   <Node>     Lof_exp indexer expb comand comp_cmd
%type   <Node>     exprel
%type   <t_m>      time_mod increment
```

```
programa  :seq_scrip { }
          ;
seq_scrip : seq_scrip script
          | script
          ;
script    : SCRIPT IDENT { } Decs { } bodyscript POINT { }
          ;
```

```
bodyscript : comp_cmd  {  }
           | START maskfile END  {  }
           ;

maskfile   : cmask  COMMA  CTE_STR  COMMA  FORMAT  {  }
           ;

decs       : decs typedecs
           | decs maskdecs
           | /* empty */
           ;

typedecs   : types
             Lof_ident  {  }  SEMICOL
           ;

maskdecs   : USES  Lof_masks  SEMICOL  {  }
           ;

Lof_masks  : Lof_masks COMMA MSKVAR    {  }
           | MSKVAR   {  }
           ;
types      : INTEGER {  }
           | FLOAT   {  }
           | STRING  {  }
           | IMAGE   {  }
           ;

typelist   : typelist OPENB index  CLOSEB {  }
           | /* empty */
           ;

index      : CTE_INT  {  }
           | /* empty */  {  }
           ;
Lof_ident  : Lof_ident COMMA IDENT  typelist {  }
           | IDENT  typelist  {  }
           ;

comp_cmd   : START  Lof_cmds END {  }
           ;

Lof_cmds   : Lof_cmds comand SEMICOL {  }
           | comand SEMICOL  {  }
           ;
```

```
comand     : IDENT APPEND exp {  }
           | WHILE expb DO comand  {  }
           | IF expb THEN  comand  {  }
           | IF expb THEN  comand  ELSE comand  {  }
           | IDENT  indexer ASSIGN  {  }  exp  {  }
           | SRAW  exp  TO  CTE_STR  {  }
           | SRAS  exp  TO  CTE_STR  {  }
           | SGIF  exp  TO  CTE_STR  {  }
           | SBMP  exp  TO  CTE_STR  {  }
           | RETURN exp  {  }
           | comp_cmd    {  }
           | SHOW OPENP exp CLOSEP  {  }
           ;

expb       : expb OR termb
           | termb
           ;

termb      : termb AND factb
           | factb
           ;

factb      : NOT factb
           | exprel
           | OPENP expb CLOSEP  {  }
           ;
exprel     : exp NE exp  {   }
           | exp LT exp  {   }
           | exp GT exp  {   }
           | exp LE exp  {   }
           | exp GE exp  {   }
           | exp EQUAL exp {   }
           ;

exp        : exp PLUS  term  {   }
           | exp MINUS term  {   }
           | term   {   }
           ;

term       : term TIMES power  {   }
           | term DIV   power  {   }
           | term MOD   power  {  .}
           | power
           ;

power      : sfactor POWER power {   }
```

```
          | sfactor
          ;

sfactor   : signal factor  {    }
          ;

factor    : CTE_INT  {    }
          | CTE_FLT  {    }
          | CTE_STR  {    }
          | variable
          | OPENKEY  Lof_exp CLOSEKEY {    }
          | RRAW OPENP exp COMMA exp COMMA exp CLOSEP {   }
          | RRAW OPENP exp COMMA exp COMMA exp COMMA exp
                  CLOSEP {    }
          | RBMP OPENP  exp CLOSEP  {    }
          | RGIF OPENP  exp CLOSEP  {    }
          | RPCX OPENP  exp CLOSEP  {    }
          | RGRIB OPENP exp CLOSEP  {    }
          | RPDUS OPENP exp CLOSEP  {    }
          | RRAS OPENP exp  CLOSEP  {    }
          | MINV OPENP exp  CLOSEP  {    }
          | MAXV OPENP exp  CLOSEP  {    }
          | GVAL OPENP exp  COMMA exp CLOSEP {   }
          | RGB  OPENP exp  COMMA exp COMMA exp CLOSEP {   }
          | EXTR OPENP exp  CLOSEP  {    }
          | EXTG OPENP exp  CLOSEP  {    }
          | EXTB OPENP exp  CLOSEP  {    }
          | BIN  OPENP exp  COMMA exp CLOSEP  {    }
          | ZOOM OPENP exp  COMMA exp CLOSEP  {    }
          | REG  OPENP exp  COMMA exp COMMA exp CLOSEP {   }
          | OPENP exp CLOSEP  {    }
          ;

cmask     : CTE_STR {    }
          ;

variable  : IDENT {    }
          | maskvar
          ;

indexer   : OPENB Lof_exp CLOSEB  {    }
          | /* empty */  {     } .

maskvar   : MSKVAR   {    }
          | MSKVAR UNTIL time_mod UNTIL increment {    }
          | MSKVAR POINT time_mod UNTIL time_mod UNTIL
```

```
                increment {     }
            ;

time_mod    : time_mod POINT signal CTE_INT time_esp {    }
            | signal CTE_INT time_esp  {    }
            ;

increment   : signal CTE_INT  time_esp  {    }
            ;

signal      :  PLUS    {  }
            |  MINUS   {  }
            |  /* empty */ {  }
            ;

time_esp    :  HOUR     {  }
            |  DAY      {  }
            |  WEEK     {  }
            |  PENTAD   {  }
            |  FIFTEEN  {  }
            |  MONTH    {  }
            |  YEAR     {  }
            ;

Lof_exp     : Lof_exp COMMA  exp  {    }
            | exp {    }
            ;
```

Apêndice C – Exemplos de Programas

```
SCRIPT while_ex
        INT ic1, ic2;
        STRING mystr, str1, str2;
BEGIN
        ic1 := 5;
        ic2 := 0;
        str1 := "INICIANDO";
        str2 := " e FINALIZANDO";
        WHILE ic1 > ic2
          DO
              BEGIN
#                 ajuste do controle
                  ic2 :=ic2 + 1;
#                 algum processamento útil
              END;
        mystr := str1 + str2;
END.

SCRIPT SINGLE-IMAGE
   IMAGE MGIN;
BEGIN
        MGIN := RGIF ( "vst100.gif" );
        SHOW  ( MGIN );
# ou opcionalmente
```

```
# SHOW ( RGIF ("vst100.gif" ) );
END.

SCRIPT MOVIE
  IMAGE XIMG[3], LIMG[];
  BEGIN
# inicio de um bloco ou comando composto
    LIMG := ($MMIS.15d.10m.98y -> 03d.12m.98y);
    XIMG[1] := MAXV ( LIMG );
    XIMG[2] := MINV ( LIMG );
    XIMG[3] := XIMG[1] + XIMG[2];
    SHOW ( ZOOM( XIMG, 2.0) );
# fim do bloco
  END.

SCRIPT MMIS
# script maskfile chamado por MOVIE
  BEGIN
    [M][M][I]yymdd.hhh, /usr/local/images, GRIB
  END.
```

Referências Bibliográficas

[App97] Andrew W. Appel *Modern Compiler Implementation in Java.* Cambridge University Press, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles Techniques and Tools.* Addison-Wesley, 1986.

[Boe88] Barry W. Boehm, *A Spiral Model of Software Development and Enhancement*, Computer, May, pages 61-72, 1988.

[BS94] C. Wayne Brown and Barry J. Shepherd, *Graphics File Formats: Reference and Guide*. Prentice Hall, 1994.

[Dou95] Douglas A. Young. *Object-Orinted Programming with C++ and OSF?/Moftif.* Prentice Hall, 1995.

[DS95] Charles Donnelly and Richard Stallman. *Bison - The YACC – compatible Parser Generator.* Free Software Foundation, Inc., May 1995.

[FB88] Charles Fischer and Joseph Le Blanc. *Crafting a Compiler,* Benjamim Cummings, 1988.

[FH95] Chis Fraser and David Hanson. *A retargetable C Compiler Design and Implementation,* Addisson-Wesley, 1995.

[GV94] Gomes,J. e Velho, L., *Computação Gráfica: Imagem.* Instituto de Matemática Pura e Aplicada Sociedade Brasileira de Matemática SBM. Rio de Janeiro, 1994.

[GW93] Gonzales, R. C. and Woods, R. E. *Digital Image Processing.* Addison-Wesley. World Student Series, 1993.

[Hol90] Allen I. Holub. *Compiler Design in C.* Prentice-Hall International, 1990.

[Jasc97] _, Paint Shop Pro version 4, Jasc Inc, 1997.

[JBA89] G. J. Jedlovec K. B. Batson, R. J. Atkinson, C. C. Moeller, W. P. Menzel, and M. W. James, *Improved capabities of the Multispectral Atmospheriic Mapping Sensor (MAMS).* Nasa Tech. Memo 100352. Marshall Space Flight, 1989.

[KS92] D. Koelma, A. Smeulders, *An Image Processing Library based on Abstract Image Data-types in C++.* Project AWI92-1691 Faculty of Mathematics and Computer Science, University of Amsterdam, 1992.

[KBS92] D. Koelma, R. van Balen and A. Smeulders, *SCIL-VP: a multi-purpose visual programming environment.* ACM/SIGAPP *Symposium on Aplied Computing*, pages 1188-1198, 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (ANSI C).* Prentice Hall, 1988.

[LDG95] Brian T. Lewis L. Peter Deutsch and Theodore C. Goldstein. Clarity Mcode: *A Retargetable Intermediate Representation for Compilation. Technical report*, Sun Microsystems, 1995.

[OSF90] *OSF/Motif Programmer's Guide, OSF/Motif Programmer's Reference*, Prentice Hall, 1990.

[Ous94] John K. Ousterhout. *Tcl and Tk Toolkit,* Addison Wesley, 1994.

[Pow93] John S. Powers. *The iAXE Image Processor.*

NOOA/NESDIS/Interactitive Processing Branch, Maryland, 1993.

[Pax90] Vern Paxson. Flex-*A fast Scanner generator.* Free Software Foundation, Inc., 1990.

[RW91] J. R. Rasure and C. S.Williams., *An Integrated Data Flow Visual Language and Software Developments Environment.* Journal of Visual Languages and Computing, pages 217-246, 1991.

[RWD90] G. X. Ritter, J. N. Wilson and J. L. Davidson. *Image algebra: an overview.* Computer Vision, Graphics, and Image Processing, 49:297-331, 1990.

[SM96] Galileu B. de Sousa e Silvino B. Magalhães, Smartech VSAT Tutorial, Relatório Interno, Smartech Soluções Computacionais, 1996.

[McCo93] Steve McConnel, *Code Complete,* Microsoft Press, 1993.

[Stro91] Bjarne Stoustrup, *The C++ Programming Language, Second Edition,* Addison-Wesley, 1991.

[Kow83] Tomasz Kowaltowski, Implementação de Linguagens de Programação, Guanabara Dois, 1983.

[TLA89] Aaron M. Tanenbaum, Yedidyah Lamgsam, and Moshe J. Augenstein. *Data Structures Using C.* Prentice Hall, 1989.

[TS86] Jean P. Tremblay and Paul G. Sorenson, *The Theory and Practice of compiler Writing,* MacGrawHill International, 1986.

[WR90] C.S. Williams and J. R. Rasure., *A Visual Language for Image Processing.* Workshop on Visual Languages, pages 86-91, 1990.

[XCLASSES] Galileu Batista de Sousa, Francisco Pinto Oliveira Júnior, *Xclasses, Uma Biblioteca de Objetos Gráficos XWindows.*

Relatório Interno Funceme, 1993.

[MEDx] http://www-mrips.od.nih.gov/uguidc/ugtoc.htm, setembro, 1998.

[UFPe] http://www.ufpe.br/~nabuço, maio, 1998.

[DOU] http://www.dou.gov.br/, setembro, 1998.