Universidade Estadual de Campinas
Instituto de Matemática, Estatística e Ciência da Computação
Departamento de Ciência da Computação

Integrador FIG-LaTeX

Aparecido Nilceu Marana

Orientador: Prof. Dr. Tomasz Kowaltowski

Novembro de 1990

Dissertação apresentada ao Departamento de Ciência da Computação (DCC-IMECC-Unicamp) como parte dos requisitos exigidos para a obtenção do título de mestre em Ciência da Computação.

Integrador FIG-LaTeX

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. **Aparecido Nilceu Marana** e aprovada pela Comissão Julgadora.

Campinas, 13 de Dezembro de 1990.

Prof. Dr. Tomasz Kowaltowski
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Aos meus pais
Nilso e Lourdes.

Agradecimentos

Ao concluir este trabalho agradeço:

- Ao Prof. Dr. Tomasz Kowaltowski, pela orientação atenciosa e segura;
- Às Prof. Eliana X. Linhares e Neuza K. Kakuta, pelo estímulo durante os anos de graduação;
- À amiga Inês Gasparotto, pela companhia valiosa durante os primeiros passos neste caminho;
- Aos amigos Toninho e Gorete, pelo apoio em Campinas;
- Aos amigos do curso de mestrado, pelos bons momentos que me propiciaram;
- Ao CNPq, pelo suporte financeiro.

Abstract

TEX is a computerized typesetting system developed by Donald E. Knuth designed to handle especially well mathematical texts. Leslie Lamport's LATEX system was developed as its extension for relatively unexperienced users. TEX does not include any facilities for drawing pictures, and those existing in LATEX are very restricted. It's described in this work the integration of the language FIG into the LATEX system which allows the user to place the specifications of his drawings within the LATEX text. These specifications are automatically translated into LATEX or PostScript commands, so that the drawings are inserted into the resulting text.

Sumário

TEX é um sistema de composição computadorizada de textos desenvolvido por Donald E. Knuth que apresenta excelentes resultados ao compor textos, especialmente *textos matemáticos*. Com o intuito de auxiliar os usuários de TEX, Leslie Lamport desenvolveu sobre ele o sistema LATEX. Infelizmente, TEX não dispõe de recursos gráficos e aqueles existentes em LATEX são muito restritos. Descreve-se, neste trabalho, a integração da linguagem FIG ao sistema LATEX. A partir desta integração, o usuário passa a especificar as figuras em FIG no próprio texto, no local onde deseja inseri-la. Tais figuras são convertidas em comandos LATEX ou PostScript. No final do processamento, obtêm-se as figuras automaticamente inseridas no texto que, então, pode ser submetido a LATEX.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Com a disseminação da informática nos mais diversos setores da sociedade, aumenta cada vez mais o seu contingente de usuários. Torna-se, então, de fundamental importância todo e qualquer esforço no sentido de possibilitar que estes usuários (muitas vezes leigos) possam se servir, sem muitos transtornos, dos recursos e facilidades por ela oferecidos.

Atualmente, existe uma gama enorme de aplicações nas quais os computadores prestam valiosos serviços, que vão desde as pesquisas espaciais/nucleares, até a resolução de simples operações aritméticas.

Nas aplicações envolvendo textos de um modo geral, têm papel relevante os sistemas de composição gráfica, sem os quais, por exemplo, seria muito mais trabalhosa a composição deste texto. Esses sistemas permitem a um autor ter controle completo sobre sua obra, uma vez que ele passa a executar o trabalho de tipografia sobre ela.

Um desses sistemas foi desenvolvido por Donald Knuth e denominado TEX [16]. TEX é um programa sofisticado projetado para compor textos, principalmente "textos matemáticos". O texto composto graficamente com TEX adquire um aspecto profissional, especialmente se for empregado um dispositivo de saída de alta resolução, com tecnologia laser, por exemplo.

Sendo a Tipografia uma arte que pode levar anos para ser assimilada, autores sem experiência podem cometer erros elementares de formatação, dificultando, assim, o entendimento das idéias que desejam transmitir. Visando auxiliar autores usuários de TEX, Leslie Lamport desenvolveu sobre esse sistema, um trabalho não menos importante, ao qual denominou LATEX [19].

LATEX adiciona a TEX uma coleção de comandos que simplificam a composição, por permitir que o autor se concentre na estrutura do texto, no seu conteúdo, ao invés de se concentrar nos comandos de formatação. Com

LaTeX pode-se fazer, num nível superior de abstração, quase tudo o que se faz com TeX.

Durante a elaboração de um texto, seja ele matemático ou não, comumente depara-se com a necessidade de se utilizar uma figura com o intuito de ilustrar uma idéia qualquer. Caso um usuário queira processar no sistema TeX um texto contendo figuras, ele terá problemas, uma vez que TeX não possui recursos gráficos.

Diante desta situação, o usuário deve utilizar uma ferramenta que lhe permita gerar suas figuras numa linguagem interpretável pelo dispositivo no qual o texto será impresso e depois inserir os arquivos contendo as especificações das figuras no arquivo onde se encontra o resto do texto. Isto pode ser feito em TeX através do comando `\special` [5]. No entanto, esta possibilidade restringe-se aos contextos em que o *driver* interpreta o comando `\special` como sendo uma macro que permite a inserção no texto fonte escrito em TeX (ou em LaTeX), de comandos destinados ao dispositivo da saída.

No sistema LaTeX este problema é amenizado, pois Lamport definiu novas fontes com as quais torna-se possível simular uma razoável categoria de figuras. Com isso, o usuário não precisa necessariamente se deslocar a outros ambientes a fim de gerar suas figuras, já que em vários casos LaTeX é suficiente para compor completamente os textos, inclusive as figuras.

Mesmo sendo um grande avanço em relação a TeX, logo se percebe que especificar figuras utilizando os recursos gráficos de LaTeX não é das tarefas mais fáceis, principalmente se estas possuem uma estrutura hierárquica complexa, como a que apresenta-se na Figura 1.1.

A linguagem utilizada para a especificação de figuras em LaTeX é relativamente pobre e freqüentemente necessita-se manipular coordenadas absolutas, fato este que contribui para elevar, consideravelmente, o grau de dificuldade durante a especificação.

Verificando-se tais transtornos, constatou-se haver necessidade de ferramentas que facilitassem a especificação de figuras e que permitissem a inserção automática dessas figuras em textos a serem compostos no ambiente TeX ou LaTeX. Essas ferramentas deveriam, preferivelmente, levar em conta o fato de que em ambiente TeX a composição gráfica é caracterizada pela estruturação lógica e não pela estruturação visual, obtida através de sistemas que seguem a linha *"WYSIWYG"* (*What You See Is What You Get*).

Simpatizantes à parte, ambas as abordagens possuem vantagens e desvantagens. Todavia, dever-se-ia manter o estilo de composição característico

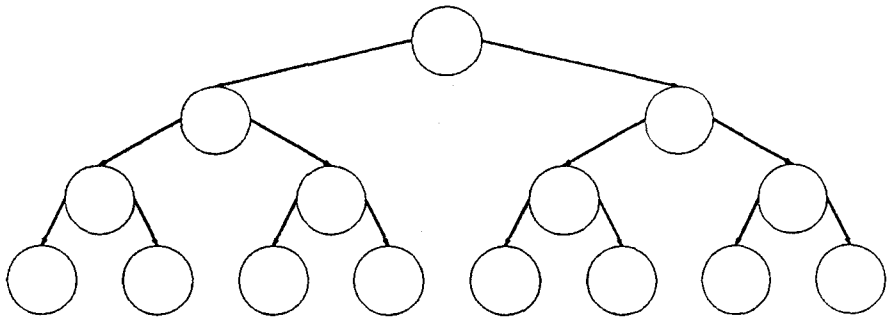


Figura 1.1: Árvore binária completa de altura 3.

de TEX, especificando-se logicamente as figuras e não visualmente. Utilizando-se a linguagem FIG [17,22] satisfaz-se esta premissa. FIG é uma linguagem própria para especificação não interativa de figuras bidimensionais que, embora sendo de propósito geral, pode ser de grande valia se integrada a sistemas de composição gráfica de textos, tais como TEX ou LATEX.

Este texto visa, então, apresentar aspectos da implementação de uma versão da linguagem FIG e mostrar alguns modelos que permitem a inclusão automática de figuras especificadas nesta linguagem, no ambiente TEX.

Capítulo 2

A Linguagem FIG

Este capítulo contém uma descrição sumária da linguagem FIG, uma vez que em [22] ela está descrita com minúcias.

Durante a implementação, a linguagem sofreu pequenas alterações que visaram simplificá-la e ajustá-la ao contexto da integração ao ambiente TEX. Tais modificações encontram-se no Apêndice B. No Apêndice E existem exemplos de especificações feitas em FIG de algumas figuras encontradas neste texto.

2.1 Idéias Gerais

FIG é uma linguagem de programação cujo objetivo é possibilitar a especificação de figuras bidimensionais de forma não interativa. Sua característica principal é a possibilidade de definição e uso de figuras de maneira análoga a procedimentos, modelando, assim, o desenho sem utilizar estrutura de dados. Isto facilita a construção de desenhos que tenham uma estrutura hierárquica complexa. As figuras permitem parametrização flexível e podem retornar valores através de um mecanismo especial. A "instanciação" das figuras pode ser modificada através de transformações geométricas embutidas na linguagem e o posicionamento pode ser feito de três modos: por um movimento implícito associado a "instanciação" consecutiva de diferentes figuras, por posicionamento relativo a outras figuras já "instanciadas" e por coordenadas absolutas.

FIG possui um conjunto de primitivas que constituem o "alicerce" de quaisquer novas figuras a serem especificadas e, como pode ser visto na próxima seção, sua sintaxe é muito semelhante à sintaxe da linguagem Pas-

```
Picture Arvore ;
 Figure Sub_Arvore ;
  Parameters{ h = 0 } ;      ! h : altura da arvore
  Results{ p1 , p2 , p3 } ;
  Begin
   If h = 0
      Then {r : Circle 0.75 ;
            p1 := r.lb ;
            p2 := r.tc ;
            p3 := r.rb }
      Else {t1 : Sub_Arvore h-1 ;
            t2 : Sub_Arvore h-1 With .p1 At t1.p3 + [1,0] ;
            rz : Circle 0.75 At [t1.p3.x + 0.5,t1.p2.y + 1] ;
            Arrow rz.lc , t1.p2 ;
            Arrow rz.rc , t2.p2 ;
            p1 := t1.p1 ;
            p2 := rz.tc ;
            p3 := t2.p3 }
  End ;
 Begin       !  corpo do programa principal
    Sub_Arvore 3 ;
 End Picture
```

Figura 2.1: Especificação em FIG da árvore binária completa de altura 3.

cal, fato este que contribui sobremaneira para sua assimilação, já que Pascal encontra-se bastante difundida.

Veja na Figura 2.1 a especificação em FIG do desenho apresentado na Figura 1.1.

2.2 Descrição Sumária

Em FIG, toda figura ao ser "instanciada" retorna, automaticamente, determinados resultados. Estes resultados são pontos estratégicos do menor retângulo que envolve a figura. A orientação do retângulo é determinada

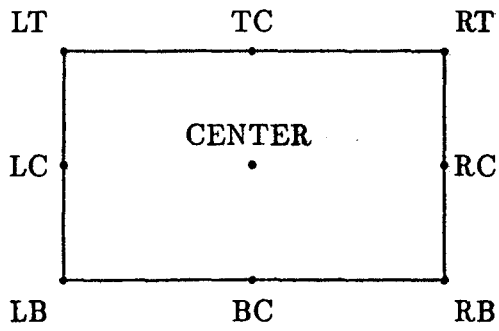


Figura 2.2: Retângulo Envolvente.

pelo sistema de coordenadas no qual a figura está inserida. Refere-se normalmente a este retângulo por *retângulo envolvente*. Os resultados fornecidos automaticamente são:

- LT – Left Top
- LC – Left Center
- LB – Left Bottom
- RT – Rigth Top
- RC – Rigth Center
- RB – Rigth Bottom
- TC – Top Center
- BC – Bottom Center
- CENTER – Center

Na Figura 2.2 pode-se visualizar estes pontos.[1]

Em FIG existem, entre outras, as variáveis pré-definidas *cc* (current coordinates) e *cd* (current direction). A variável *cc*, após ser inicializada como

[1] A especificação em FIG desta figura está exibida no Apêndice E.

sendo a origem, vai assumindo automaticamente novos valores com o decorrer do posicionamento (explícito ou implícito) das figuras. A variável *cd* é inicializada com o valor *east* e indica a direção que está sendo utilizada na especificação da figura. Assim como *cc*, *cd* é empregada principalmente pelo mecanismo de posicionamento implícito.

Veja o comando de "instanciação" de figuras descrito em 2.2.3 para obter maiores detalhes sobre estas variáveis.

Um programa escrito em FIG constitui-se das seguintes partes: cabeçalho, definição de figuras e corpo do programa principal, nesta ordem.

2.2.1 O cabeçalho

O cabeçalho é composto pelo nome do desenho e, opcionalmente, pela definição (em centímetros) da unidade de comprimento a ser utilizada. Se a unidade de comprimento não for explicitamente definida, ela é considerada como sendo 1 cm.

Exemplo:

```
Picture Arvore ;
   UnitLength = 3 ;
```

2.2.2 A Definição de Figuras

A definição de uma figura é análoga à declaração de um procedimento na linguagem Pascal, porém, não se permite aninhamento, ou seja, definições dentro de definições.

Toda figura tem um movimento implícito próprio e é definida num sistema local de coordenadas, cujos eixos são paralelos aos eixos do sistema global. O sistema local é utilizado unicamente para a definição da figura. Para que uma figura seja efetivamente desenhada, ela precisa ser "instanciada" direta, ou indiretamente, no sistema global, quando, então, sua posição, tamanho e orientação no plano do desenho ficam bem determinados.

A definição de uma figura decompõe-se em cinco partes, a saber: nome, parâmetros, combinações, resultados e corpo da figura.

Nome:

Identificador através do qual a figura é declarada.

Parâmetros:

Ao definir uma figura, o programador deve especificar quais são seus

parâmetros, desde que haja algum. Se não existirem, esta seção pode ser omitida. Os parâmetros em FIG são passados apenas por valor e podem ser de qualquer um dos tipos existentes na linguagem. Durante a "instanciação" de uma figura, pode-se omitir alguns parâmetros, desde que estes tenham um valor-padrão, ou que os não omitidos constituam uma das combinações definidas. Um valor padrão pode ser associado a um parâmetro durante a declaração dos parâmetros.

Combinações:

Uma combinação é um subconjunto do conjunto de parâmetros da figura. Para cada combinação de parâmetros, existe uma especificação alternativa para a figura em questão. No momento da "instanciação", o programador especifica os parâmetros que naquele instante lhe são mais convenientes. Com estes parâmetros deve ser possível a identificação de pelo menos uma das combinações. Se duas ou mais combinações forem identificadas, é escolhida a que foi declarada em primeiro lugar. Caso não seja possível identificar nenhuma combinação apenas com os parâmetros passados, tenta-se identificar alguma delas utilizando-se também os parâmetros que não foram passados, mas que têm valor padrão. Se nem assim for possível identificar uma combinação, então estar-se-á diante de um erro. Após identificada uma combinação, os comandos referentes a ela são utilizados na construção da figura.

Resultados:

Em FIG os parâmetros são passados apenas por valor. Se uma figura produz um resultado que deve ser visível fora do seu contexto, este resultado deve ser atribuído a uma variável que é declarada na seção `Results`. A linguagem fornece automaticamente o retângulo envolvente de toda figura instanciada. Para que os resultados de uma figura possam ser utilizados, a sua "instanciação" precisa ser rotulada.

Corpo da Figura:

O corpo da figura se enquadra nas regras do corpo do programa principal, descrito na seção 2.2.3, com uma única diferença: o corpo da figura deve terminar com "`End ;`", enquanto que o corpo do programa principal deve terminar com "`End Picture`".

Exemplo de definição de uma figura em FIG:

```
Figure exemplo ;
  Parameters{a,b = 1,c} ;
  Combinations{ 1 : a,b ;
                2 : b,c ;
                3 : a,c
              } ;
  Results{pto1,pto2,area} ;
  Begin
     Case exemplo
        { 1 : { ..... } ;
          2 : { ..... } ;
          3 : { ..... } ;
        }
  End ;
```

2.2.3 O Corpo do Programa

O corpo do programa é constituído por uma seqüência de comandos separados, necessariamente, por ponto-e-vírgulas. Tais comandos, por sua vez, compõem-se de variáveis, constantes, expressões e dos próprios comandos.

Variáveis, Constantes e Expressões

Em FIG existem duas categorias de variáveis e constantes: as pré-definidas e as definidas pelo usuário. As variáveis e constantes pré-definidas são globais, enquanto que as demais são visíveis apenas localmente. Não há necessidade de se declarar os tipos das variáveis, pois estes são determinados pelo compilador, baseado no contexto em que elas aparecem pela primeira vez. Contudo, ao final da compilação, toda variável deve ter o tipo definido, senão estar-se-á diante de um erro. Os tipos suportados por FIG são: *numérico*, *ponto no plano*, *cadeia de caracteres*, *booleano* e *direção*. O tipo *direção* é um tipo enumerado, sendo possíveis os valores: *north*, *south*, *east* ou *west*.

As variáveis e constantes podem ser combinadas em expressões, através de operadores e de funções embutidas na linguagem. Tais operadores e funções estão descritos no Apêndice A.

Comandos de Atribuição

- Atribuição a variáveis

  variável := *expressão* ;

- Atribuição a constantes

  constante :== *expressão* ;

Comandos de Repetição

- `Repeat` *expressão* `times` { *comandos* } ;

  O resultado da avaliação da expressão é truncado.

- `For` *variável* `from` *expressão1* `to` *expressão2* `step` *expressão3* `do` { *comandos* } ;

  A declaração de `step` *expressão3* é opcional (o valor-padrão para *expressão3* é 1).

- `While` *expressão* `do` { *comandos* } ;

Comandos de Decisão

- `If` *expressão* `then` { *comandos1* } `else` { *comandos2* }

  A declaração de `else` { *comandos2* } é opcional.

- `Case` *expressão*

  { *rótulo numérico* : { *comandos1* } ;
  rótulo numérico : { *comandos2* } ;
  ⋮
  rótulo numérico : { *comandosn* }
  }

  O valor de *expressão* é truncado para poder ser comparado com os rótulos numéricos, que devem ser constantes numéricas inteiras sem sinal. Este comando também é utilizado no corpo das figuras que possuem combinações. Nestes casos, a expressão deve ser composta apenas pelo nome da figura.

Comando de Finalização

- `Exit`

  Este comando interrompe a execução dos comandos da figura na qual esteja inserido.

Comando de "Instanciação" de Figuras

A forma geral deste comando é a seguinte:

Rótulo Operações Nome-da-Figura Parâmetros Posicionamento

Todas as partes são opcionais, exceto, é óbvio, a referente ao nome da figura.

Rótulo: é um identificador sucedido pelo caractere ":". Ele identifica a "instância" da figura de modo a permitir acessos futuros a seus resultados. Isto significa que os resultados de uma determinada "instância" só podem ser utilizados se esta for rotulada. Por exemplo, para utilizar o resultado *res* de uma figura cuja "instância" foi rotulada por *rot*, basta escrever *rot.res*.

Operações: As operações são transformações geométricas embutidas na linguagem, que permitem ao usuário modificar as figuras. Uma figura pode ser transformada por uma ou mais operações. Neste caso, devem vir separadas pelo caractere"|". A operação que for especificada por último, será a primeira a ser aplicada. É importante notar que as transformações geométricas, em geral, não são comutativas, portanto, a ordem em que são declaradas é relevante. As operações existentes em FIG são:

- `Rotate` *ângulo , ponto*: rotaciona a figura no sentido anti-horário em *ângulo* graus sobre o ponto *ponto*. Se *ponto* for omitido, então será utilizado o ponto *[0,0]* como padrão.
- `Scale` *fx , fy , ponto*: amplia/reduz a figura utilizando *fx* e *fy* como fatores de escala para as abscissas e ordenadas, respectivamente. O fator *fy* pode ser omitido; se isto ocorrer, é utilizado o valor de *fx* em seu lugar. O ponto *ponto* é utilizado como referência para fazer a redução/ampliação. Se for omitido é utilizado o ponto *[0,0]*.

- `Reflect` *ponto1* , *ponto2*: espelha a figura em torno da linha definida pelos pontos *ponto1* e *ponto2*.

Nome-da-Figura: é simplesmente o nome com o qual a figura foi definida.

Parâmetros: Os parâmetros são uma lista de expressões associada à lista de parâmetros formais. Essa associação pode ser feita de duas formas: usando sintaxe posicional ou não posicional. Na forma não posicional, o programador determina, através do símbolo "=", o parâmetro formal ao qual deve ser associada determinada expressão. Os parâmetros podem vir em qualquer ordem. Já na forma posicional, as expressões devem ocupar na lista, a mesma posição ocupada pelo parâmetro formal ao qual deve ser associada. Como já foi dito, alguns ou mesmo todos os parâmetros podem ser omitidos. Na forma não posicional, logicamente, basta não associá-lo a nenhuma expressão, enquanto que na posicional, as posições dos parâmetros omitidos devem ficar vazias. Se na forma não posicional forem associadas duas expressões a um mesmo parâmetro formal, então estar-se-á diante de um erro.

Posicionamento: Se o posicionamento for omitido, então o compilador utilizará o mecanismo de posicionamento implícito de figuras, ou seja, o ponto de concatenação da figura (que depende da direção corrente *cd*) coincidirá com o ponto determinado pelas coordenadas correntes *cc*. As regras para determinar o ponto de concatenaçaõ das figuras são as seguintes:

1. O ponto de concatenação das primitivas *Line*, *Arrow* e *Arc* é o seu resultado explícito *First*.
2. Para as demais figuras, o ponto de concatenação é o *LC*, *RC*, *TC* ou *BC* do seu retângulo envolvente, caso a direção corrente seja *east*, *west*, *south* ou *north*, respectivamente.

Se o posicionamento não for omitido, deve ter a seguinte sintaxe:
`with` *.res* `at` *ponto* .

O identificador *res* deve ser, necessariamente, um resultado implícito ou explícito do tipo ponto da figura que está sendo "instanciada", enquanto que *ponto* pode ser um ponto qualquer do plano.

A omissão do posicionamento pode ser parcial. Quando "`with` *.res*" ou "`at` *ponto*" forem omitidos, não simultaneamente,o compilador uti-

lizará "`with .center`[2]" ou "`at cc`", respectivamente, em seus lugares.

Como pode ser notado, o posicionamento nada mais é do que a aplicação da operação de translação às figuras.

Após o posicionamento de uma figura, o valor de *cc* é atualizado segundo as regras:

1. Se a figura for a primitiva *Line*, *Arrow* ou *Arc*, o seu resultado *Last* é atribuído à *cc*;
2. Para as demais figuras, primitivas ou não, é atribuído à *cc* o valor *RC*, *LC*, *TC* ou *BC* do seu retângulo envolvente, caso a direção indicada pela variável *cd* seja *east*, *west*, *north* ou *south*, respectivamente.

[2]Embora a utilização do ponto de concatenação da figura seja mais coerente, já que este ponto é utilizado nos casos em que há omissão total de posicionamento, na prática, a utilização do ponto center demonstrou ser bastante útil.

Capítulo 3

Esquemas de Implementação do Integrador FIG-LaTeX

O objetivo principal desse trabalho é fornecer meios para que se possa inserir, num texto a ser composto em ambiente TeX, figuras especificadas através da linguagem FIG. Para tanto, foram criados alguns modelos, sendo que a diferença mais significativa entre eles é a possibilidade de se obter ou não um resultado final que seja independente de dispositivo de saída.

3.1 Modelo 1: *Utilizando recursos gráficos de* LaTeX

Como foi dito, ao desenvolver LaTeX, Lamport agregou-lhe alguns recursos gráficos para tornar possível a especificação de uma gama razoável de figuras sem ser necessária a utilização de outras ferramentas. Tais recursos estão disponíveis no ambiente *picture*.

3.1.1 Recursos Gráficos de LaTeX

Em LaTeX, todas as figuras devem ser especificadas no ambiente *picture* e podem ser compostas de texto, segmentos de reta, setas, círculos, circunferências e outros objetos que são posicionados de acordo com suas coordenadas (x,y).

A unidade de comprimento usada para determinar posições no ambiente *picture* é o valor de `\unitlength`. Além das posições, todas as medidas são especificadas em termos de `\unitlength`, cujo valor padrão é um ponto (1/72 de uma polegada). Para mudar a escala de uma figura, basta atribuir

um novo valor para \unitlength através do comando \setlength.

O ambiente *picture* tem um par de argumentos que especifica o tamanho da figura (em termos de \unitlength). Esse ambiente produz uma caixa cuja largura e altura são dadas por este par de argumentos. A posição padrão da origem é o canto esquerdo inferior (*cei*) desta caixa, contudo, o ambiente *picture* pode ter, opcionalmente, um segundo par de argumentos para determinar as coordenadas do *cei* da caixa.

São os comandos \put e \multiput que posicionam os objetos numa figura. Os objetos que podem ser utilizados no ambiente *picture* são: \makebox, \framebox, \dashbox, \line, \vector, \shortstack, \circle, \circle*, \oval e \frame. Dentre estes, os que mais interessam para a implementação deste modelo são:

- \line(h_declive,v_declive){comp}

- \vector(h_declive,v_declive){comp}
  Desenham um segmento de reta tendo seu ponto de referência determinado pelo comando \put$(x_0,y_0)$ e seu declive pelas coordenadas (h_declive,v_declive), onde h_declive e v_declive são inteiros entre -6 e 6 para o objeto \line e -4 e 4 para \vector, sendo mdc(h_declive,v_declive) = 1 .

  No caso de \vector, a ponta da seta é desenhada no extremo oposto ao ponto de referência.

  O "comprimento horizontal" do segmento é comp, desde que h_declive não tenha valor zero. Se isto ocorrer, comp será o valor do "comprimento vertical" do segmento de reta.

- \circle{diam}

- \circle*{diam}
  Desenham uma circunferência e um círculo, respectivamente, com diâmetro tão próximo quanto possível de diam. O centro da circunferência (ou círculo) é o ponto (x_0,y_0) determinado pelo comando \put$(x_0,y_0)$.

  A maior circunferência que pode ser desenhada tem diâmetro aproximadamente igual a 1/2 polegada. Para o círculo, esse valor pode ser no máximo 1/5 de uma polegada.

- \makebox(l,h)[pos]{txt}
  Este comando cria uma caixa de largura l e altura h sem, contudo, desenhar seu contorno e posiciona o texto txt em seu interior, utilizando

o argumento pos, que é opcional. Se pos for omitido, o texto é centralizado na caixa. O canto esquerdo inferior da caixa é determinado pelas coordenadas (x_0,y_0) do comando put(x_0,y_0).

Veja na Figura 3.1, a especificação feita em LaTeX do desenho mostrado na Figura 1.1.

Note-se que o número de comandos necessários para a especificação de uma árvore de altura 4 seria, aproximadamente, o dobro do que foi utilizado para a árvore de altura 3, enquanto que na especificação via linguagem FIG (veja a figura 2.1) , bastaria "instanciar" a figura `Sub_Arvore` passando `h=4` como parâmetro. Certamente, com a utilização do comando `\multiput` esta especificação tornar-se-ia menos extensa. Ainda assim, tal especificação seria mais complexa se comparada àquela feita através da linguagem FIG.

3.1.2 Descrição do modelo

No modelo que se propõe nesta seção, as figuras especificadas em FIG são traduzidas para LaTeX. Com isso, por exemplo, o programa FIG apresentado na Figura 2.1 pode ser automaticamente transformado naqueles comandos LaTeX mostrados na Figura 3.1. O produto final obtido é independente de dispositivo de saída e, além dessa importante característica, possibilita-se ao usuário uma maior abstração durante a definição de figuras, pois, fazê-la diretamente em LaTeX pode ser deveras tedioso.

As primitivas da linguagem FIG dependem dos recursos gráficos utilizados e, devido às limitações dos recursos de LaTeX, elas acabam ficando um pouco restritas, caracterizando-se na maior desvantagem deste modelo, uma vez que, por exemplo, um segmento de reta especificado em FIG com uma determidada inclinação pode ficar deturpado, caso tal inclinação não seja suportada por LaTeX.

Por razões óbvias, este esquema não permite a integração de FIG diretamente a TeX, mas apenas a LaTeX.[1]

Na Figura 3.2 é apresentada uma representação gráfica do esquema de implementação para este modelo. Nesta representação, os retângulos denotam arquivos, enquanto que as circunferências denotam programas. A semântica associada a esta representação é a seguinte: inicialmente o usuário constrói um arquivo (.FLT) contendo texto e comandos LaTeX intercalados com especificações de figuras feitas através da linguagem FIG. Este arquivo

[1] A integração a TeX poderia ser realizada através dos recursos gráficos a ele incorporados via PiCTeX [26].

```
\begin{picture}(15.10, 5.60)( 0.95, 0.95)
   \put( 1.50, 1.50){\circle{1.000}}
   \put( 3.50, 1.50){\circle{1.000}}
   \put( 2.50, 3.00){\circle{1.000}}
   \put( 2.00, 3.00){\vector(-1,-2){ 0.50}}
   \put( 3.00, 3.00){\vector(1,-2){ 0.50}}
   \put( 5.50, 1.50){\circle{1.000}}
   \put( 7.50, 1.50){\circle{1.000}}
   \put( 6.50, 3.00){\circle{1.000}}
   \put( 6.00, 3.00){\vector(-1,-2){ 0.50}}
   \put( 7.00, 3.00){\vector(1,-2){ 0.50}}
   \put( 4.50, 4.50){\circle{1.000}}
   \put( 4.00, 4.50){\vector(-5,-3){ 1.50}}
   \put( 5.00, 4.50){\vector(5,-3){ 1.50}}
   \put( 9.50, 1.50){\circle{1.000}}
   \put(11.50, 1.50){\circle{1.000}}
   \put(10.50, 3.00){\circle{1.000}}
   \put(10.00, 3.00){\vector(-1,-2){ 0.50}}
   \put(11.00, 3.00){\vector(1,-2){ 0.50}}
   \put(13.50, 1.50){\circle{1.000}}
   \put(15.50, 1.50){\circle{1.000}}
   \put(14.50, 3.00){\circle{1.000}}
   \put(14.00, 3.00){\vector(-1,-2){ 0.50}}
   \put(15.00, 3.00){\vector(1,-2){ 0.50}}
   \put(12.50, 4.50){\circle{1.000}}
   \put(12.00, 4.50){\vector(-5,-3){ 1.50}}
   \put(13.00, 4.50){\vector(5,-3){ 1.50}}
   \put( 8.50, 6.00){\circle{1.000}}
   \put( 8.00, 6.00){\vector(-4,-1){ 3.50}}
   \put( 9.00, 6.00){\vector(4,-1){ 3.50}}
\end{picture}
```

Figura 3.1: Especificação em LaTeX da árvore binária completa de altura 3.

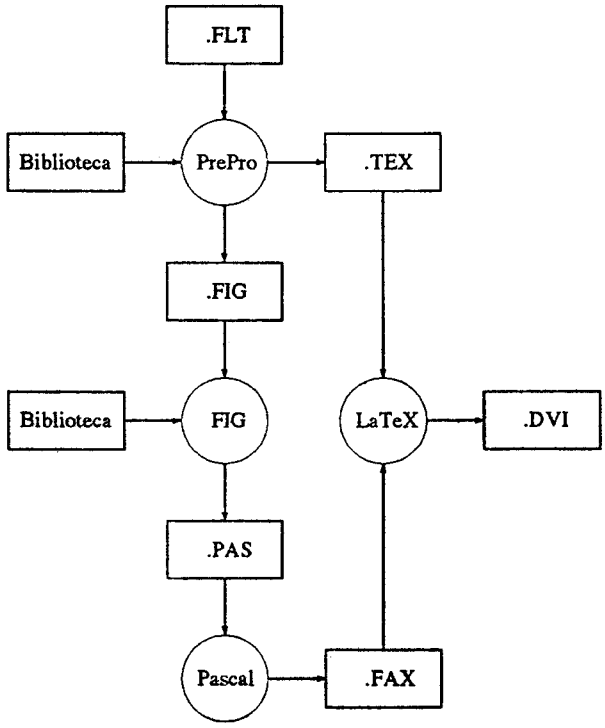


Figura 3.2: Esquema de implementação utilizando recursos gráficos de LaTeX.

é submetido a um pré-processamento (PrePro) que separa o texto e os comandos LaTeX das especificações das figuras, armazenando-os em dois novos arquivos: .TEX e .FIG, respectivamente. As figuras especificadas em FIG são, então, compiladas e traduzidas para um programa Pascal que, por sua vez, ao ser compilado e executado, gera um novo arquivo, .FAX, contendo as descrições em LaTeX das figuras inicialmente descritas em FIG. Finalmente, submetem-se os arquivos .TEX e .FAX a LaTeX obtendo-se o arquivo .DVI que contém uma descrição independente de dispositivo de saída das páginas do texto (incluindo-se as figuras).

3.2 Modelo 2: *utilizando recursos gráficos de PostScript.*

3.2.1 O que é PostScript

PostScript [1,2,3,13] é uma poderosa e flexível linguagem desenvolvida pela Adobe Systems Incorporated que tem a capacidade de descrever eficientemente o aspecto de textos, imagens e figuras geométricas de uma página, independentemente do dispositivo no qual ela será impressa. Por tais motivos, vem se tornando um padrão como linguagem para a descrição de páginas, tanto que já foi incorporada em algumas das mais inovadoras impressoras, dentre as quais a LaserWriter da Apple Computer, Inc [4].

PostScript possui uma grande variedade de operadores gráficos que permitem-lhe descrever precisamente uma determinada página. Estes operadores manipulam três tipos de objetos gráficos:

Texto: podem ser definidos com uma vasta variedade de fontes e colocados na página em qualquer posição, orientação e escala.

Figuras Geométricas: podem ser construídas usando-se os operadores gráficos que descrevem a localização de linhas e curvas de qualquer tamanho, orientação e largura, além de preencher espaços de qualquer tamanho, formato e cor.

Imagens: fotografias ou quaisquer outras imagens digitalizadas podem ser colocadas na página em qualquer escala ou orientação.

PostScript contém elementos de diferentes linguagens de programação, assemelhando-se muito a Forth [10]. Os dados que manipula são armazenados em pilha na memória, utiliza a notação pós-fixa para as operações e sua unidade de comprimento é 1 ponto (1/72 de uma polegada).

3.2.2 Recursos gráficos de PostScript

Com a linguagem PostScript é possível especificar figuras através de um conjunto de operadores gráficos. A especificação começa com a definição de um *caminho* (*path*) sobre uma superfície virtual denominada *página corrente* (*current page*). Um caminho é um conjunto de linhas e curvas que definem uma região a ser preenchida ou representa uma trajetória a ser desenhada na página corrente. Depois de construído o caminho sobre a superfície virtual, pode-se efetivamente imprimi-lo num papel.

Segue-se abaixo uma relação de operadores com os quais pode-se definir e marcar os caminhos sobre a página corrente:

- `a b moveto`
  Este operador atribui ao ponto corrente o ponto (a,b).

- `a b lineto`
  Este operador constrói um segmento de reta ligando o ponto corrente ao ponto (a,b).

- `a b rlineto`
  Este operador constrói um segmento de reta ligando o ponto corrente ao ponto definido por $(pc.x + a\ ,\ pc.y + b)$, onde $pc.x$ representa a abscissa e $pc.y$ a ordenada do ponto corrente.

- `x y r ang1 ang2 arc`
  Este operador constrói um arco circular, definido pelos argumentos `x,y,r,ang1,ang2`, onde: `(x,y)` é o ponto que determina o centro da curvatura, `r` o raio da curvatura e os ângulos `ang1` e `ang2` determinam o início e o fim do arco, no sentido anti-horário.

- `stroke`
  Este operador marca na página corrente o caminho que foi definido.

- `fill`
  Este operador marca na página corrente a região definida pelo caminho.

Existem outros operadores, porém, com estes é possível implementar à maioria das primitivas da linguagem FIG e, assim, viabilizar o modelo proposto nesta seção.

Note que, em PostScript, as circunferências e as elipses podem ser obtidas com o operador `arc` e com transformações adequadas de escala. Quanto aos círculos, eles podem ser obtidos trivialmente através dos operadores `arc` e `fill`.

Veja abaixo um programa escrito em PostScript para desenhar uma árvore binária completa de altura 2. Note-se que existem outras formas para se definir tal figura em PostScript, sendo que aquelas que utilizam comandos iterativos ou procedimentos recursivos podem ficar consideravelmente mais simples. No entanto, tais especificações tendem a ser menos *legíveis* se comparadas àquelas feitas em FIG.

```
/cm {28.34645669 mul}def
/seta {
 0 0 moveto
-0.15 cm  0.05 cm rlineto
 0.00 cm -0.10 cm rlineto
 0.15 cm  0.05 cm rlineto fill}def
1 setlinewidth
newpath  gsave
[ 1.00  0.00  0.00  1.00  1.50 cm  1.50 cm] concat
 0.50 0.50 scale
 0 0 1 cm 0 360 arc
stroke  grestore newpath  gsave
[ 1.00  0.00  0.00  1.00  3.50 cm  1.50 cm] concat
 0.50 0.50 scale
 0 0 1 cm 0 360 arc
stroke  grestore newpath gsave
[ 1.00  0.00  0.00  1.00  2.50 cm  3.00 cm] concat
 0.50 0.50 scale
 0 0 1 cm 0 360 arc
stroke  grestore newpath
 2.00 cm  3.00 cm  moveto
 1.50 cm  2.00 cm lineto
stroke  newpath gsave
[ 1.00  0.00  0.00  1.00  1.50 cm  2.00 cm] concat
243.43 rotate
seta  grestore stroke  newpath
 3.00 cm  3.00 cm  moveto
 3.50 cm  2.00 cm lineto
stroke newpath gsave
[ 1.00  0.00  0.00  1.00  3.50 cm  2.00 cm] concat
-63.43 rotate
seta  grestore stroke newpath gsave
[ 1.00  0.00  0.00  1.00  5.50 cm  1.50 cm] concat
 0.50 0.50 scale
 0 0 1 cm 0 360 arc
stroke grestore newpath gsave
[ 1.00  0.00  0.00  1.00  7.50 cm  1.50 cm] concat
 0.50 0.50 scale
 0 0 1 cm 0 360 arc
```

```
stroke grestore newpath  gsave
[ 1.00  0.00  0.00  1.00  6.50 cm  3.00 cm] concat
 0.50 0.50 scale
 0 0 1 cm 0 360 arc
stroke grestore newpath
 6.00 cm  3.00 cm  moveto
 5.50 cm  2.00 cm lineto
stroke  newpath gsave
[ 1.00  0.00  0.00  1.00  5.50 cm  2.00 cm] concat
243.43 rotate
seta grestore stroke newpath
 7.00 cm  3.00 cm  moveto
 7.50 cm  2.00 cm lineto
stroke newpath gsave
[ 1.00  0.00  0.00  1.00  7.50 cm  2.00 cm] concat
-63.43 rotate
seta grestore stroke newpath gsave
[ 1.00  0.00  0.00  1.00  4.50 cm  4.50 cm] concat
 0.50 0.50 scale
 0 0 1 cm 0 360 arc
stroke  grestore newpath
 4.00 cm  4.50 cm  moveto
 2.50 cm  3.50 cm lineto
stroke newpath gsave
[ 1.00  0.00  0.00  1.00  2.50 cm  3.50 cm] concat
213.69 rotate
seta grestore stroke newpath
 5.00 cm  4.50 cm  moveto
 6.50 cm  3.50 cm lineto
stroke newpath gsave
[ 1.00  0.00  0.00  1.00  6.50 cm  3.50 cm] concat
-33.69 rotate
seta grestore stroke showpage
```

3.2.3 Descrição do modelo

Normalmente, os programas PostScript são gerados automaticamente por programas de composição, tais como: processadores de textos, sistemas CAD

e outros.

No modelo de integração aqui proposto isto também ocorre, visto que as figuras especificadas em FIG são convertidas em programas PostScript. Estes, por sua vez, são traduzidos para comandos do dispositivo de saída através de um software auxiliar (interpretador). Os arquivos gerados por esse software são agregados, finalmente, ao resto do texto via comando `\special` [5].

Embora não sendo dependente de dispositivo, este modelo só é viável se o driver do dispositivo de saída a ser utilizado interpretar adequadamente o comando `\special` e se o software que converte PostScript para comandos do dispositivo de saída estiver disponível. Obviamente, se o dispositivo de saída hospedar um interpretador para PostScript, este modelo torna-se bem mais simples e mais eficiente.

A representação gráfica do esquema de implementação para este modelo está apresentada na Figura 3.3; nela os retângulos que aparecem preenchidos denotam conjuntos de arquivos. A semântica associada a tal representação é basicamente a mesma do modelo visto na seção anterior, a diferença é que as figuras inicialmente especificadas em FIG são traduzidas para a linguagem PostScript e o resultado obtido após a tradução de cada figura é armazenado num arquivo próprio com extensão .PS. Se o dispositivo de saída dispõe de um interpretador PostScript, basta atrelar os arquivos .PS ao arquivo .TEX através do comando `\special`. Caso contrário, são necessários dois passos adicionais antes de anexar os arquivos ao arquivo .TEX: submeter os arquivos .PS a um interpretador PostScript que gera código na linguagem aceita pelo dispositivo de saída a ser utilizado e "filtrar" os arquivos resultantes desta interpretação para possibilitar a inserção adequada das figuras no resto do texto. No Capítulo 5 pode-se obter mais detalhes a respeito destes passos adicionais, visto que eles também constam no modelo adotado.

Quanto às primitivas da linguagem FIG, elas não ficam submetidas a nenhuma restrição, já que os recursos gráficos de PostScript satisfazem todos os requisitos para a implementação das primitivas previstas no projeto original da linguagem, inclusive com os seus respectivos atributos.

Neste modelo, FIG pode ser integrada tanto a TEX quanto a LATEX.

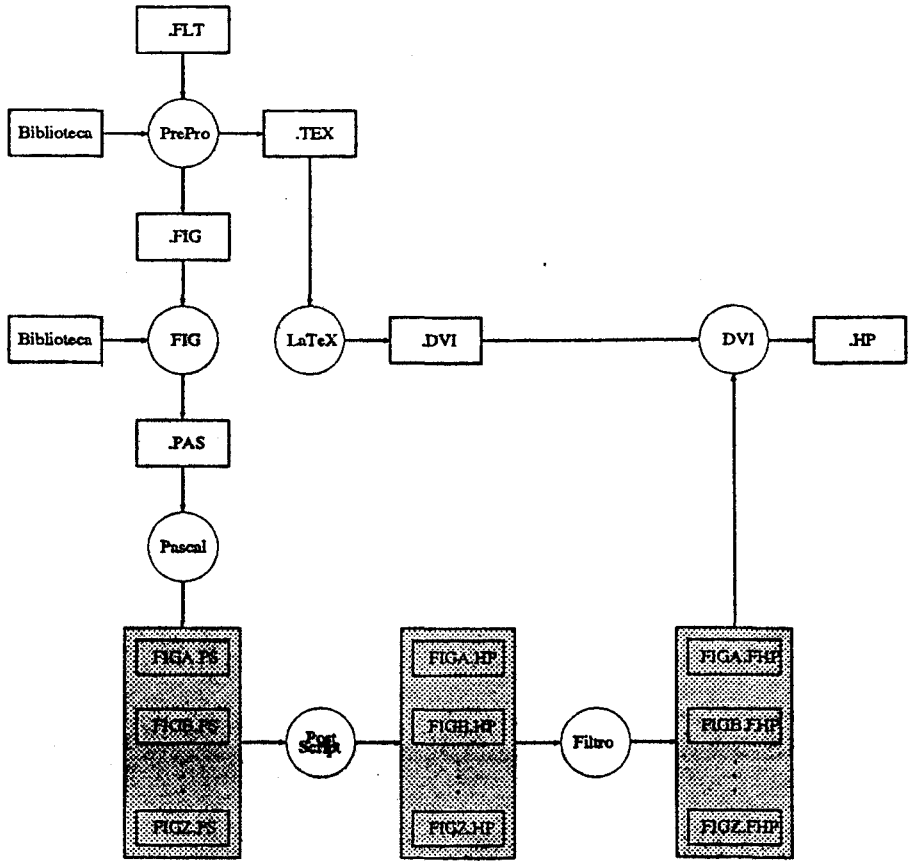


Figura 3.3: Esquema de implementação utilizando recursos gráficos de PostScript.

3.3 Modelo 3: *utilizando recursos gráficos de um dispositivo de saída específico*

Caso as figuras a serem especificadas em FIG requeiram mais das primitivas do que se pode obter com os recursos de LaTeX e o software que converte PostScript para comandos de um específico dispositivo não esteja disponível, pode-se, então, alterar as primitivas da linguagem FIG de modo que elas passem a gerar código diretamente na linguagem deste dispositivo.

O modelo assim implementado torna o usuário dependente de dispositivo e, consequentemente, sempre que ele desejar utilizar outros dispositivos terá que reescrever as primitivas, adaptando-as à nova situação.

Evidentemente, neste modelo faz-se necessária a especificação do equipamento que será utilizado como dispositivo de saída, uma vez que serão utilizados os seus recursos gráficos. Para exemplificar este modelo, supõe-se que o equipamento a ser utilizado como dispositivo de saída seja a impressora com tecnologia laser: *LaserJet series II* da *Hewlett Packard* [12].

3.3.1 Recursos gráficos da impressora LaserJet series II

A linguagem utilizada pela LaserJet series II denomina-se PCL (Printer Command Language) e, assim como a impressora, também foi desenvolvida pela Hewlett Packard.

A linguagem PCL não é muito poderosa no tocante a recursos gráficos. Não existe nenhuma primitiva gráfica para desenhar linhas, curvas, polígonos, círculos, etc. Os únicos recursos disponíveis para especificação de figuras são:

- retângulos de tamanhos arbitrários que podem ser utilizados para fazer linhas horizontais e verticais;
- mapas de bits (*bitmaps*) que devem ser definidos pelo usuário.

Também é possível o preenchimento desses retângulos com vários padrões.

Através dos mapas de bits, é possível a construção de qualquer figura. porém, esta construção é extremamente complexa e trabalhosa.

3.3.2 Descrição do modelo

Neste modelo, a integração de FIG ao ambiente TeX se dá de forma análoga ao Modelo 2, isto é, os arquivos contendo as especificações em linguagem do dispositivo são anexados ao texto original através do comando `\special`.

Portanto, este modelo também depende da capacidade do driver do dispositivo de saída interpretar adequadamente o comando `\special`.

Este modelo permite a integração de FIG tanto a TEX quanto a LATEX e a representação gráfica do seu esquema de implementação pode ser visto na Figura 3.4. A semântica associada a tal representação é basicamente a mesma dos modelos apresentados nas seções anteriores, a diferença é que ao ser compilado e executado o programa Pascal são gerados vários arquivos com extensões .HP, um para cada figura inicialmente especificada em FIG. Estes arquivos contêm as descrições das figuras na linguagem aceita pelo dispositivo de saída e são anexados, via comando `\special`, ao arquivo .TEX.

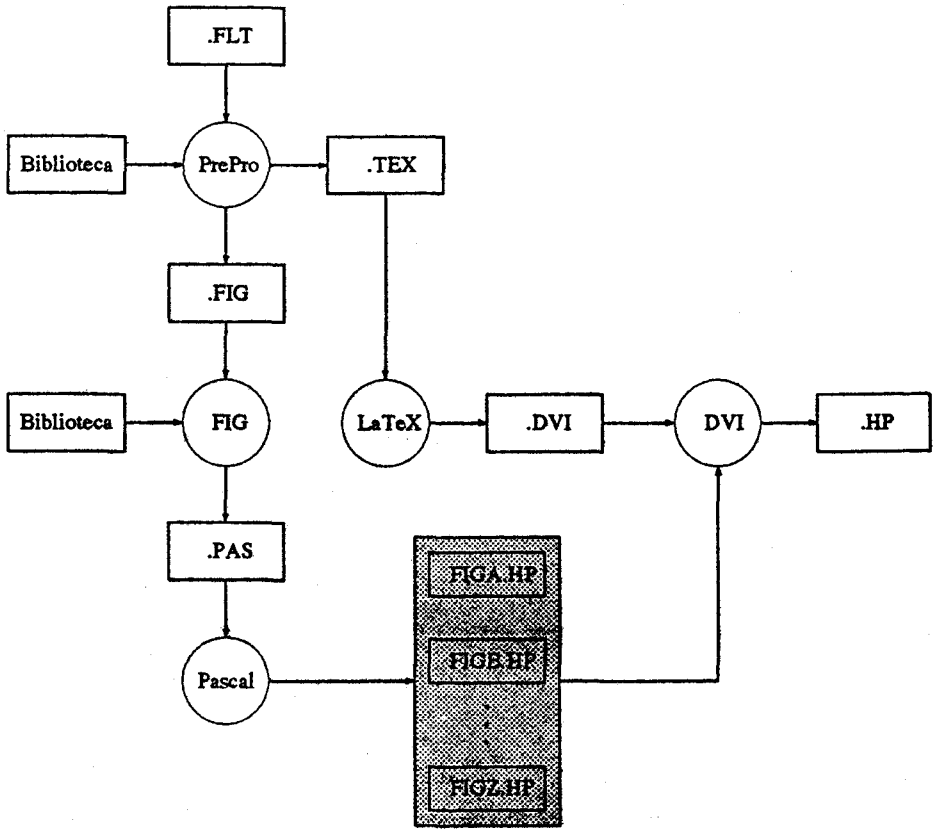


Figura 3.4: Esquema de implementação utilizando recursos gráficos do dispositivo de saída.

Capítulo 4

Aspectos da Implementação da Linguagem FIG

Para a viabilização deste projeto de integração da linguagem FIG ao ambiente TEX, foi necessária a implementação completa da linguagem FIG, uma vez que não havia nenhuma versão disponível, exceto algumas diretrizes indicando meios para se obtê-la [22]. Neste capítulo são detalhados os aspectos da implementação da linguagem FIG.

Na Figura 4.1 pode-se visualizar os passos da tradução de um programa-fonte escrito em FIG para rotinas/comandos de um determinado pacote gráfico.[1] Com o intuito de deixar a linguagem mais flexível, independente de um pacote gráfico específico, a tradução é feita em dois níveis. No primeiro, o programa FIG é traduzido para um programa escrito em uma linguagem de alto nível – no caso, Pascal. No segundo nível, este programa é convertido em chamadas a procedimentos do particular pacote gráfico. Desse modo, quando desejar-se utilizar outro pacote, basta alterar este segundo nível de tradução, o que é relativamente simples, já que isto implica apenas em redefinir as figuras primitivas da linguagem FIG que se encontram numa biblioteca. Todas as dependências a um determinado pacote gráfico estão encapsuladas no conjunto das primitivas.

4.1 Ambiente de Implementação

Utilizou-se a linguagem Pascal, precisamente o Turbo Pascal – versão 5.0 [9], na implementação do compilador para a linguagem FIG. Com isso, o com-

[1] A especificação em FIG desta figura está exibida no Apêndice E.

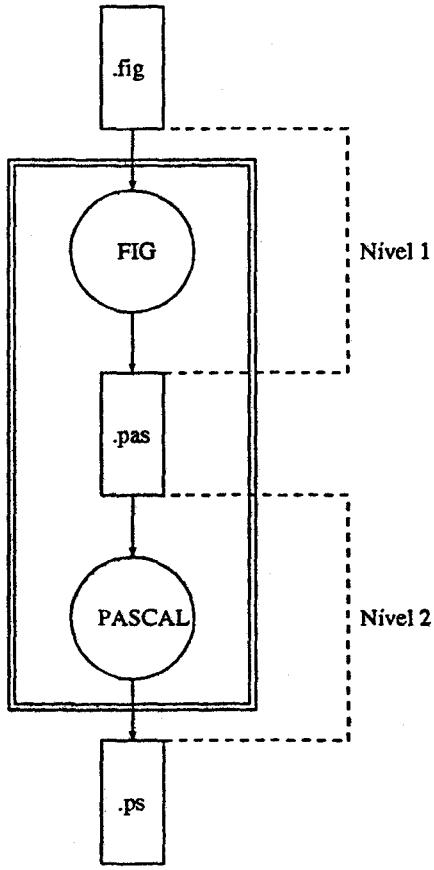


Figura 4.1: Níveis de tradução de um programa FIG.

pilador FIG pode ser executado em qualquer equipamento compatível com a linha IBM-PC.

A linguagem Pascal foi escolhida, pois, além de estar bastante difundida, possui todas as características desejáveis para a realização deste trabalho.

4.2 Passos de Compilação

O compilador FIG gera, como código objeto, um programa escrito na linguagem de alto nível, Pascal. Em Pascal, as constantes e variáveis são declaradas explicitamente no início de cada bloco. Já em FIG, as constantes podem ser declaradas em qualquer parte do programa (através da utilização do símbolo de atribuição a constantes :==) e os tipos das variáveis são determinados analisando-se o contexto no qual aparecem pela primeira vez. Logo, não é possível entremear as fases de análise e de geração de código durante a compilação de um programa FIG, fato este que caracteriza compiladores de um único passo. Portanto, foi necessário desenvolver um compilador de dois passos.

No primeiro passo da compilação é feita a análise do programa FIG e gerada uma representação interna deste programa. Por tratar-se de uma técnica simples e eficiente, optou-se pela análise sintática descendente recursiva para a implementação do compilador [18]. Nesta técnica, lança-se mão de um conjunto de procedimentos mutuamente recursivos, sendo que cada um deles corresponde a um símbolo não terminal da gramática que gera a linguagem.

Quanto à representação interna, ela constitui-se numa "árvore do programa". Desse modo, ao analisar o comando condicional
IF ***expressão*** **THEN** { *comandos1* } **ELSE** { *comandos2* }
são armazenados: uma marca associada ao símbolo terminal **IF**, a *expressão*, os *comandos1* e os *comandos2*, enquanto que os símbolos terminais: {, }, **THEN** e **ELSE** são desprezados.

Veja na Figura 4.2 como este comando é armazenado internamente na árvore do programa.

Evidentemente, ao serem armazenados os comandos e as expressões envolvidas, as mesmas regras são aplicadas.

No segundo passo é gerado, a partir da árvore, o programa Pascal. Para tanto, todas as variáveis devem, necessariamente, estar associadas a algum tipo suportado pela linguagem FIG. Se isto não ocorrer, a rotina de tratamento de erros é chamada.

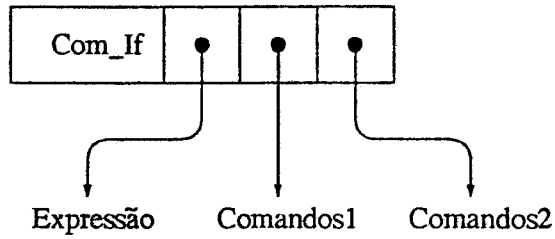


Figura 4.2: Representação interna do comando condicional.

4.3 A Determinação dos Tipos

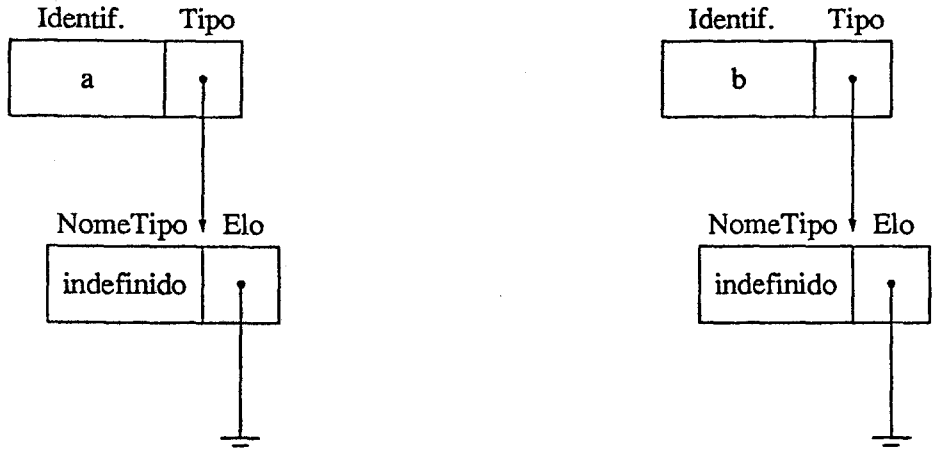
Se o tipo de um objeto não puder ser definido imediatamente após sua primeira aparição no programa, então ele é inserido numa lista ligada cujos elementos devem ser associados ao mesmo tipo que ele. Desse modo, quando o tipo de um dos objetos dessa lista puder ser definido, este tipo é propagado e associado a todos os demais objetos da lista.

Considere a especificação da figura abaixo:

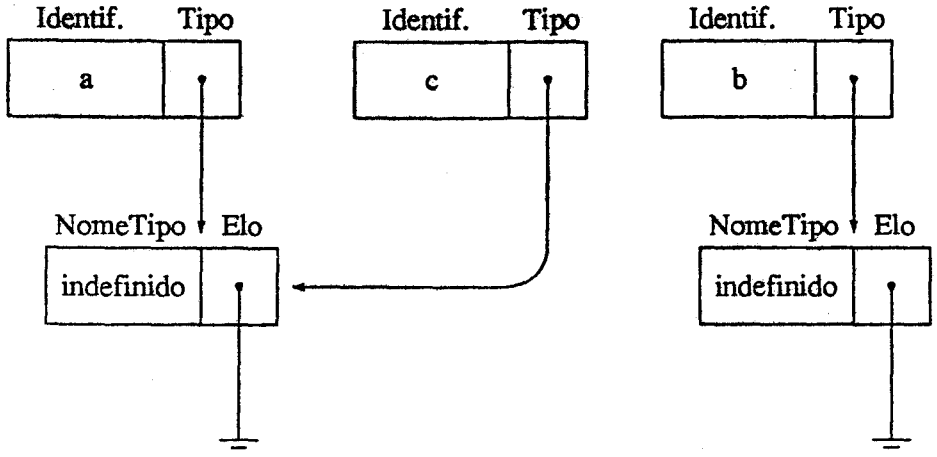
```
Figure exemplo ;
   Results{a,b} ;
   Begin
      c := a ;
      a := b ;
      b := [2,3]
   End ;
```

A determinação dos tipos das variáveis envolvidas na especificação acima é feita através dos seguintes passos:

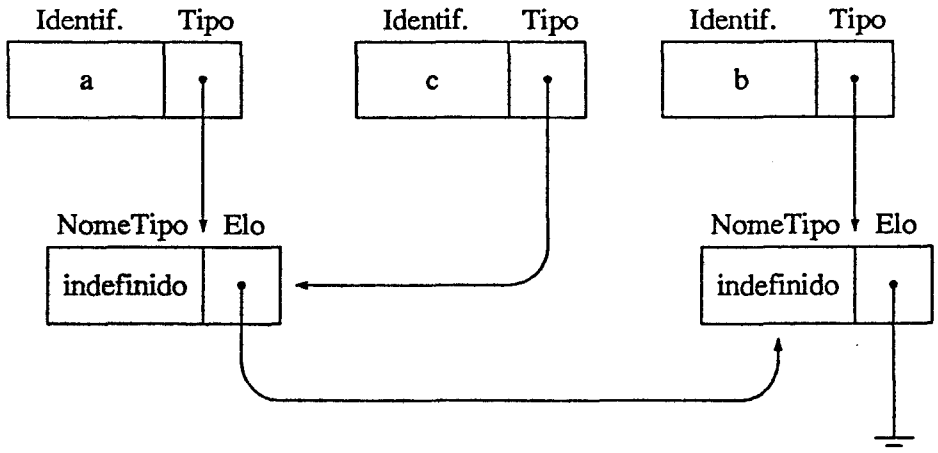
1. `results{a,b} ;`



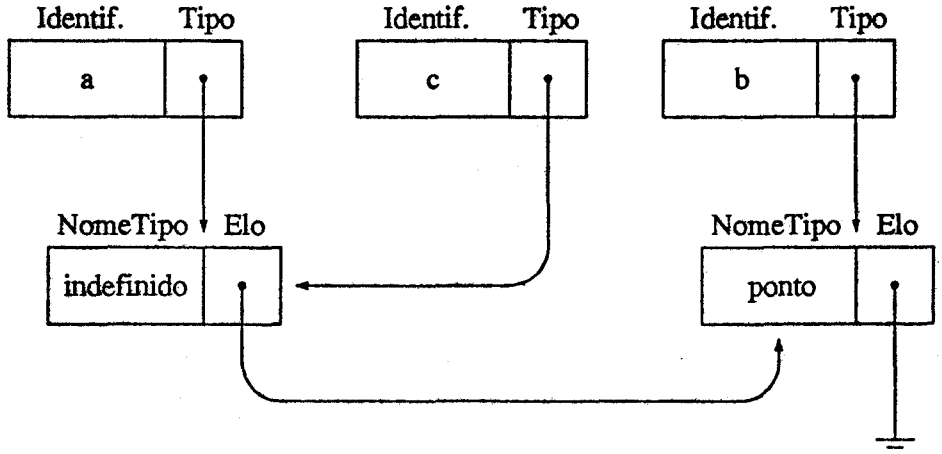
2. `c := a ;`



3. `a := b ;`



4. `b := [2,3]`[2]



A construção desta estrutura é feita na fase de análise. Na fase de síntese,

[2] A especificação em FIG desta figura está exibida no Apêndice E.

para gerar código da declaração do tipo de uma variável, o compilador percorre a lista por ela encabeçada até encontrar o ponteiro *nil* no campo ELO de alguma célula da lista. O valor do campo TIPO desta célula é, então, atribuído à tal variável. Se o tipo encontrado for indefinido, então estar-se-á diante de um erro.

4.4 A Tabela de Símbolos

A tabela de símbolos é tratada como uma pilha que contém símbolos de diferentes categorias: variáveis, constantes, figuras, resultados e parâmetros. Inicialmente, ela é composta apenas pelas variáveis, constantes e figuras pré-definidas. Com o decorrer da análise, os identificadores definidos localmente nas figuras são colocados temporariamente na tabela, enquanto estas estiverem sendo analisadas. Ao ser analisado o corpo do programa principal, a tabela de símbolos conterá, além dos símbolos inicialmente inseridos, apenas os identificadores das figuras definidas pelo usuário e as variáveis e constantes "declaradas" nesta região, visto que apenas estes identificadores são visíveis no corpo do programa principal.

4.5 O Tratamento de Erros

O tratamento de erros é bastante simples: ao detectar um erro léxico, sintático ou semântico, a análise é interrompida e é enviada uma mensagem para o usuário dizendo em que linha foi encontrado o erro. Além disso, explicita-se, através da mensagem, o erro cometido. Veja no Apêndice C uma lista de erros possíveis de serem detectados pelo compilador FIG.

Um tratamento de erros mais adequado poderia ser obtido utilizando-se um dos métodos existentes na literatura (o método de Wirth [27], por exemplo). Uma discussão completa sobre recuperação de erros em analisadores sintáticos descendentes é encontrada em [23].

4.6 O Código Objeto

O código objeto é gerado pelo compilador FIG na linguagem Pascal. Cada figura definida em FIG é traduzida para esta linguagem em um procedimento com o mesmo nome. Além dos procedimentos referentes às figuras, o programa Pascal gerado é constituído, em geral, de chamadas a procedimentos pré-definidos que se encontram compilados numa unidade a parte

(biblioteca). Tais procedimentos são responsáveis, entre outras coisas, pelas transformações geométricas que podem ser aplicadas às figuras e pela atualização dos seus respectivos retângulos envolventes.

O projeto da linguagem FIG baseou-se em alguns aspectos no projeto da linguagem Pascal, principalmente no que tange aos comandos: condicional, de atribuição e de repetição. Dessa forma, o código gerado para tais comandos é quase que uma cópia dos mesmos.

4.7 A Passagem de Parâmetros

As figuras definidas em FIG são traduzidas para a linguagem Pascal em procedimentos que possuem alguns parâmetros fixos e outros variantes. Os parâmetros variantes são aqueles declarados pelo programador FIG quando este define a figura. Os parâmetros fixos, comuns a todos os procedimentos, são:

origem anterior e origem atual: indicam a origem do sistema de coordenadas da figura chamada (a qual foi traduzida para o procedimento em questão) e da figura que a chamou, respectivamente. Servem para atualização do retângulo envolvente.

resultado: é passado por referência e retorna informações sobre o retângulo envolvente da figura e sobre os demais resultados. Na seção 4.9 existem informações adicionais sobre este parâmetro.

indicador de visibilidade: indica se a "instanciação" é real ou virtual, ou seja, indica se devem ser feitas chamadas ao pacote gráfico a fim de gerar efetivamente a figura ou se a "instanciação" serve apenas para obter-se informações sobre as dimensões da figura e, assim, tornar possível o seu posicionamento relativo (seção 4.10).

matriz de transformação: é uma matriz quadrada de ordem três que contém as transformações geométricas que devem ser aplicadas à figura. Veja a seção 4.11 para obter mais informações sobre matrizes de transformações.

direção corrente: leva ao procedimento informações sobre a direção corrente que se está utilizando na especificação do desenho. Essa informação é necessária para o cálculo dos pontos de concatenação e,

conseqüentemente, para o posicionamento automático da figura no desenho. Veja o item *Posicionamento* na seção 2.2.3 onde esta questão também é tratada.

número da combinação: diz qual combinação está sendo passada durante a instanciação da figura e, em decorrência, quais são os comandos que devem ser executados para que ela seja obtida. Se a figura for definida sem combinações, então o valor-padrão 0 (zero) é passado. Por isso, ao serem definidas as combinações, não pode-se utilizar este valor como número de uma combinação.

opção pelo pacote gráfico: como conseqüência das peculiaridades do modelo de integração implementado (veja o Capítulo 5), o produto final obtido após alguns passos constitui-se de especificações em LaTeX e/ou em PostScript das figuras inicialmente definidas em FIG. Assim, este parâmetro indica qual pacote gráfico deve ser utilizado para construir determinada figura. Veja a discussão sobre as diretivas na seção 5.3, onde esta questão é tratada.

Além destes parâmetros fixos, que são transparentes ao usuário, são passados também os parâmetros por ele definidos durante a especificação da figura. O tratamento dado pelo compilador a estes parâmetros é o seguinte: ao fazer a análise de uma figura, o compilador insere os parâmetros formais (desde que existam) na tabela de símbolos e numa lista ligada que constitui-se em um campo do registro da tabela de símbolos no qual está o identificador da figura ora analisada. Cada célula desta lista contém os seguintes campos: identificador, tipo, valor-padrão e um indicador para dizer se o parâmetro formal possui valor-padrão. Esta lista também é "marcada" por um apontador da árvore do programa que, como foi dito, é construída na fase de análise do compilador.

O compilador ao analisar um comando de "instanciação" de figura, "pendura" na árvore do programa uma lista ligada contendo os parâmetros efetivos. Cada célula dessa lista de parâmetros efetivos possui os seguintes campos: expressão, tipo e um indicador que diz se há algum parâmetro efetivo sendo associado explicitamente ao parâmetro formal correspondente.

A passagem dos parâmetros em FIG pode ser feita através de sintaxe posicional ou não-posicional. No primeiro caso, a expressão encontrada na posição do i-ésimo parâmetro é colocada na i-ésima célula da lista de parâmetros efetivos, no campo destinado à expressão, enquanto que o indicador de passagem de parâmetro efetivo desta célula assume o valor *verda-*

deiro. Caso o i-ésimo parâmetro efetivo seja omitido, é consultada a i-ésima célula da lista de parâmetros formais para verificar se aquele parâmetro possui valor-padrão. Se possuir, este valor é, então, utilizado como parâmetro efetivo, porém o indicador de passagem de parâmetro efetivo recebe o valor *falso.*

No segundo caso, sintaxe não-posicional, deve-se fazer uma associação explícita dos parâmetros efetivos e formais. Assim, por exemplo, para associar ao parâmetro formal `radius` da primitiva `circle` o valor 2, basta fazer `circle radius = 2`. Neste tipo de análise, o compilador verifica se o identificador foi realmente declarado como sendo parâmetro formal e qual é a j-ésima posição ocupada por ele ao ser declarada a figura. Se esta verificação for positiva, então, o parâmetro efetivo é colocado na j-ésima célula da lista de parâmetros efetivos, no campo expressão e o indicador de passagem de parâmetro efetivo assume o valor *verdadeiro.* Finalmente, percorre-se a lista de parâmetros efetivos e, para as células cujo campo expressão estejam vazios, preenche-se com o valor-padrão do parâmetro formal correspondente, caso exista, e atribui-se aos respectivos indicadores de passagem de parâmetros efetivos o valor *falso.*

Se um parâmetro formal não possuir valor-padrão e também não for associado a nenhum parâmetro efetivo, então um valor indefinido é a ele atribuído, desde que haja a seção de declaração de combinações na figura; caso contrário, estar-se-á diante de um erro.

4.8 As Combinações

Quando o compilador analisa uma seção de declaração de combinações, ele constrói uma lista ligada L que constitui-se em um campo do registro da tabela de símbolos alocado para o identificador da figura à qual pertence tal seção. Em cada célula desta lista é armazenado o número da combinação e a lista l composta pelos identificadores dos parâmetros formais que a definem. Na lista L existem tantas células quantas forem as combinações.

Ao ser "instanciada" uma figura, o compilador procede da seguinte forma para tornar explícita a combinação implicitamente passada:

- cria uma lista l' com os identificadores dos parâmetros formais que foram associados a algum parâmetro efetivo e verifica se existe uma lista de parâmetros l em L, tal que $l = l'$;
- se existir, então toma o número da combinação que se encontra na mesma célula de l como sendo o número da combinação passada;

- se não existir l em L, tal que $l = l'$, então acrescenta a l' todos os parâmetros formais que possuem valor-padrão, obtendo, assim, uma lista l'';
- finalmente, verifica se existe l em L, tal que l está contida em l''. Se esta verificação der resultado positivo, então é tomado o número associado a l como sendo o número da combinação passada. Caso contrário, estar-se-á diante de um erro: *combinação inválida.*

Se mais de uma combinação for satisfeita, então será tomada a que foi declarada em primeiro lugar.

Se uma figura for definida sem a seção de declaração de combinações, então o número zero é tomado como o número da combinação. Além disso, todos os parâmetros formais que não possuem valor-padrão têm que ser, necessariamente, associados a algum parâmetro efetivo.

4.9 Os Resultados

Em FIG, a comunicação entre os módulos do programa – as figuras – é realizada através de variáveis declaradas como parâmetros ou como resultados. Os parâmetros (seção 4.7) levam dados ao módulo chamado, enquanto que, inversamente, os resultados retornam dados deste módulo.

A linguagem FIG é dotada de um mecanismo para retornar, automaticamente, alguns resultados que dizem respeito ao menor retângulo que envolve uma figura "instanciada", ao qual denomina-se, aqui, apenas *retângulo envolvente.* Além desses resultados, comuns a todas as figuras, a linguagem permite que o programador declare outras variáveis para exercerem a função de retornar resultados.

Na fase de análise, quando o compilador encontra uma seção de declaração de resultados, ele insere temporariamente seus identificadores na tabela de símbolos e cria uma lista ligada para armazená-los. Esta lista é "pendurada" na árvore do programa para ser utilizada na fase de síntese – geração de código, mais precisamente. O tipo de cada resultado deve ser determinado na fase de análise e é um componente de cada célula da lista ligada.

Como já foi dito, uma figura é traduzida em um procedimento cujos parâmetros são aqueles declarados na figura, acrescidos de alguns outros transparentes ao usuário. Entre estes últimos, existe um parâmetro do tipo T, abaixo definido, passado por referência, cuja finalidade é armazenar dados resultantes da "instanciação" da figura.

```
Type  Retenv = Record
                  lb,rt : Ponto ;
                End ;

      T = Record
            fixa : Retenv ;
            r1   : Tipo1 ;
            r2   : Tipo2 ;
            .
            .
            .
          End ;
```

O campo denominado `fixa` contém informações sobre o retângulo envolvente (`lb` – left bottom – canto esquerdo inferior do retângulo envolvente e `rt` – right top – canto direito superior do retângulo envolvente). Os demais campos variam de figura para figura, pois são os resultados declarados pelo programador. Portanto, para cada figura (incluindo-se as primitivas), corresponde uma definição do tipo T acima descrito. Estas definições são codificadas em Pascal no início da geração de código, quando, então, o compilador percorre, na árvore do programa, a parte que contém a lista de figuras, analisa os resultados de cada figura e gera o tipo T a ela correspondente.

Se uma figura "instanciada" for rotulada, então o identificador do rótulo é, no código gerado, o parâmetro efetivo do tipo T passado para o procedimento correspondente à figura em questão. Caso contrário, se não for rotulada, é gerado um outro identificador (oculto ao usuário) para ser tal parâmetro efetivo e, neste caso, os resultados da figura não podem ser utilizados.

4.10 O Posicionamento das Figuras

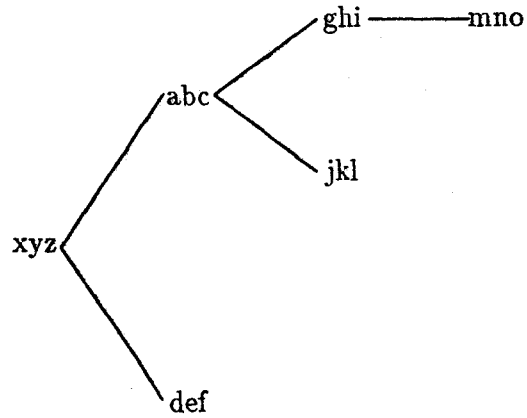
O aspecto mais problemático desta implementação diz respeito ao posicionamento das figuras. Suponha o seguinte comando de "instanciação": `xyz with .center at cc`. Para posicionar a figura `xyz` com o centro do seu retângulo envolvente no ponto `cc`, é preciso antes conhecer as dimensões de `xyz`. Como não é utilizada nenhuma estrutura de dados, a única forma de determinar as dimensões da figura `xyz` é "instanciá-la". Isto acarreta em dupla "instanciação" da figura. Na primeira "instância" (virtual) não há

chamadas a rotinas do pacote gráfico, ou seja, a figura não é efetivamente construída; a "instância" virtual serve apenas para determinar as dimensões de uma figura.

Com estas informações, torna-se possível o cálculo dos fatores de translação (Δx,Δy) necessários para posicionar a figura xyz com o centro de seu retângulo envolvente no ponto cc, basta para isso "concatenar" tais fatores na matriz de transformação com a qual ela será novamente "instanciada". Nesta segunda "instância" (real) pode haver ou não chamadas às rotinas do pacote gráfico, dependendo do valor do indicador de visibilidade.

Portanto, se uma subfigura está no nível n da hierarquia, então ela será "instanciada" 2^n vezes.

Suponha que a figura **xyz** seja definida através da seguinte estrutura hierárquica:



então, a subfigura mno, que está no nível 4 na hierarquia, será "instanciada" 16 vezes ao ser "instanciada" a figura xyz.

Este aspecto (dupla "instanciação") merece ser investigado para melhorar o desempenho do compilador. Qualquer alternativa deve envolver algum tipo de estrutura de dados que deve ser transparente ao usuário para não contrariar a filosofia do projeto da linguagem. Por outro lado, essa dupla "instanciação" permite que certos resultados (por exemplo, distâncias, possam ser utilizados na definição da própria figura.

4.11 As Transformações Geométricas

Considerando que os desenhos têm uma estrutura hierárquica e que, portanto, podem ser decompostos em partes, W.M.Newman concebeu a idéia de *procedimentos de display* [20] como descritores das partes de um desenho. Tais partes, por sua vez, podem ser decompostas em sub-partes, logo, os procedimentos podem ser definidos através de chamadas a outros procedimentos. Newman propôs que nestas chamadas pudessem ser especificadas algumas transformações às quais estaria submetida a parte do desenho descrita pelo procedimento chamado. São essas transformações referentes ao tamanho, posição e orientação das partes de um desenho que tornam viáveis os procedimentos de display, já que cada figura é definida num sistema de coordenadas próprio e tem que se "enquadrar" no sistema de coordenadas da figura chamadora que está num nível superior na hierarquia do desenho.

Em decorrência da possibilidade de se chamar procedimentos com transformações, numa implementação dos procedimentos de display tem que se criar formas de gerenciar e aplicar corretamente tais transformações. A dificuldade surge à medida em que uma parte do desenho transformada, pode exigir, por sua vez, a transformação de suas sub-partes, e assim por diante.

A linguagem FIG foi concebida sobre o conceito de procedimentos de display e, nesta implementação da linguagem, eles são materializados da seguinte forma: ao chamar um procedimento que consiste na especificação de uma parte do desenho, salva-se o "status" corrente, concatena-se as transformações impostas ao procedimento ora chamado com as transformações vindas das hierarquias superiores e, então, transfere-se o controle ao procedimento chamado. Ao receber de volta o controle deste procedimento, recupera-se o "status" antigo e prossegue-se a execução normalmente.

Utiliza-se matrizes denominadas *matrizes de transformações* [11] para armazenar as transformações geométricas. Deste modo, concatenar transformações significa multiplicar essas matrizes. Visto que a ordem de aplicação das transformações é relevante, a ordem de multiplicação dessas matrizes também o é (a operação de multiplicação de matrizes não é comutativa).

As transformações geométricas suportadas pela linguagem FIG são:

Translação Para aplicar esta transformação, deve-se utilizar a seguinte sintaxe: `xyz with .res at p`, que significa transladar a figura `xyz` de tal forma que o seu resultado `res` (que deve ser do tipo *ponto*) coincida com o ponto `p`.

A matriz de tranformação para translações é da forma:

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D_x & D_y & 1 \end{pmatrix}$$

onde D_x é o deslocamento nas abscissas e D_y é o deslocamento nas ordenadas.

Para aplicar a transformação representada por T a cada ponto, expressa-se cada ponto em *coordenadas homogêneas* [11] numa matriz-linha

$$P = \begin{pmatrix} x_p & y_p & 1 \end{pmatrix}$$

e multiplica-se ambas as matrizes, $P.T$, obtendo-se uma nova matriz-linha que nada mais é do que o ponto transformado e também expresso em coordenadas homogêneas.

A expressão dos pontos através de coordenadas homogêneas permite que as transformações possam ser tratadas de forma homogênea e combinadas entre si.

Rotação Para aplicar esta transformação, deve-se utilizar a seguinte sintaxe: **`Rotate ang , p xyz`**, que significa rotacionar **ang graus**, no sentido anti-horário, a figura **xyz** sobre o ponto **p**.

A matriz de transformação para rotações sobre a origem é da forma:

$$R = \begin{pmatrix} \cos\alpha & \text{sen}\alpha & 0 \\ -\text{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

onde α é o ângulo de rotação.

Portanto, o resultado da rotação de um ponto P (expresso em coordenadas homogêneas) em α graus no sentido anti-horário sobre a origem consiste no resultado da multiplicação $P.R$.

A rotação sobre um ponto genérico $P_1(x, y)$ é obtida através da concatenação de três transformações: uma translação com fatores $D_x = -x$ e $D_y = -y$ (translação de P_1 à origem), uma rotação sobre a origem em α graus e outra translação com fatores $D_x = x$ e $D_y = y$ (translação ao ponto P_1 original).

Escala Para aplicar esta transformação deve-se utilizar a seguinte sintaxe: `Scale sx , sy , p xyz`, que significa ampliar/reduzir a figura `xyz` utilizando-se `sx` como fator de escala para as abscissas e `sy` como fator de escala para as ordenadas. O ponto `p` é utilizado como ponto de referência para a efetuação da transformação de escala.

A transformação de escala pode produzir um deslocamento indesejado se não for definido corretamente o ponto `p`.

A matriz de transformação para ampliação/redução, tendo a origem como ponto de referência, é da forma:

$$S = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Portanto, o resultado da aplicação de uma transformação de escala em um ponto P (expresso em coordenadas homogêneas) tendo como referência o ponto $(0,0)$ e utilizando os fatores de escala S_x e S_y, consiste no resultado da multiplicação $P.S$.

A ampliação/redução tendo como referência um ponto genérico $P_1(x,y)$, é obtida através da concatenação de três transformações: uma translação com fatores $D_x = -x$ e $D_y = -y$ (translação de P_1 à origem), uma ampliação/redução utilizando os fatores S_x e S_y e a origem como ponto de referência e em seguida outra translação com fatores $D_x = x$ e $D_y = y$ (translação ao ponto P_1 original).

Reflexão Para aplicar esta transformação deve-se utilizar a seguinte sintaxe: `Reflect p1 , p2 xyz`, que significa refletir a figura `xyz` tendo como referência a linha definida pelos pontos `p1` e `p2`.

Utilizando-se a matriz

$$E = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

obtém-se a reflexão das figuras sobre o eixo das ordenadas.

Multiplicando-se adequadamente a matriz E por outras matrizes que armazenam transformações determinadas a partir dos pontos `p1` e `p2`, é possível obter-se o efeito desejado.

Nesta implementação da linguagem FIG, ao se chamar um procedimento que traduz uma figura qualquer, passa-se, como sendo um de seus parâmetros, uma matriz quadrada de ordem três resultante da concatenação das transformações recebidas dos níveis superiores da hierarquia com as transformações a serem aplicadas exclusivamente à figura associada ao procedimento em questão. Dentro deste procedimento, esta matriz pode ser concatenada a novas transformações ao serem chamados outros procedimentos que também traduzem subpartes da figura, e assim por diante.

4.12 As Primitivas da Linguagem FIG

As primitivas da linguagem FIG são figuras pré-definidas e pré-compiladas que ficam armazenadas na biblioteca do sistema. Na definição de tais figuras encontram-se as chamadas às rotinas do pacote gráfico responsáveis pela efetiva construção dos desenhos. As primitivas da linguagem constituem-se num conjunto básico sobre o qual toda figura é definida.

Nesta versão do compilador FIG, implementou-se a maioria das primitivas previstas no projeto original da linguagem, no entanto, a forma como isto foi feito, principalmente no que tange aos atributos, impôs-lhes pequenas alterações. Algumas dessas alterações decorreram do fato dos recursos gráficos utilizados apresentarem certas limitações. Cada primitiva possui, na verdade, duas versões: uma utilizando recursos gráficos do LaTeX e outra recursos gráficos do PostScript.

É interessante observar que é perfeitamente possível a inserção de novas primitivas na biblioteca, aliás, recomenda-se fortemente esta prática, já que ela pode melhorar sobremaneira o tempo de compilação/execução de um programa, principalmente se a referida figura é freqüentemente utilizada no programa.

4.12.1 Descrição das Primitivas

As primitivas implementadas são:

Circle Esta primitiva está definida com o seguinte cabeçalho:

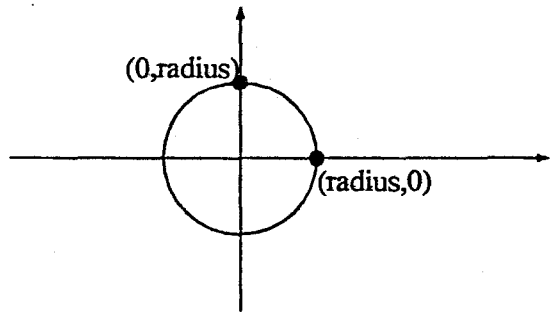


Figura 4.3: Desenho produzido pela primitiva Circle (exceto os eixos e os pontos destacados).

```
Figure Circle ;
   Parameters{ radius = 1 , fill = false } ;
   Combinations{ 1 : radius ;
                 2 : radius , fill
               } ;
   Results{ n , s , e , w } ;
```

Ao chamá-la, obtém-se o desenho de uma circunferência. Caso não seja passado como parâmetro nenhum valor para o raio, é utilizado o valor-padrão do raio. Veja na Figura 4.3 um desenho produzido pela primitiva Circle.

Se for detectada a passagem da combinação de número 2, ou seja, se forem atribuídos valores para ambos os parâmetros durante a "instanciação", então a região delimitada pela circunferência é preenchida, caso `fill` tenha valor *true*.

Além dos pontos do retângulo envolvente, os resultados retornados após uma "instanciação" de Circle são: *n (north)*, *s (south)*, *e (east)* e *w (west)*.

A versão da primitiva Circle em LaTeX possui certas restrições, a saber:

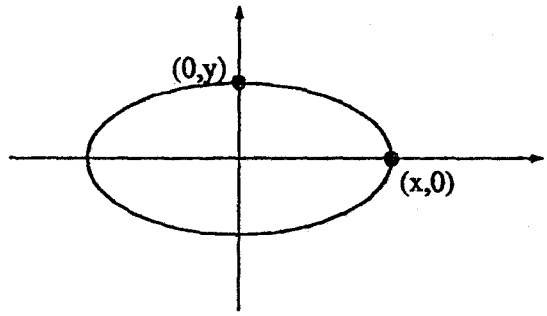


Figura 4.4: Desenho produzido pela primitiva Ellipse (exceto os eixos e os pontos destacados).

- o diâmetro que uma circunferência pode ter é de, no máximo, 1/2 polegada; se a região delimitada pela circunferência tiver que ser preenchida, este valor cai para 1/5 de uma polegada.
- a primitiva Circle é imune às transformações de escala.

Ellipse Esta primitiva está definida com o seguinte cabeçalho:

```
Figure Ellipse ;
   Parameters{ x = 2 , y = 1 , fill = false } ;
   Combinations{ 1 : x,y ;
                 2 : x,y,fill } ;
   Results{ n , s , e , w } ;
```

Este cabeçalho é bastante parecido com o da primitiva Circle, o que era de se esperar, visto que toda circunferência é uma elipse cujo valor x coincide com o valor y. Na verdade, na implementação da primitiva Circle, existe uma chamada à primitiva Ellipse passando-se $radius = x = y$ como parâmetro. Veja na Figura 4.4 um desenho produzido pela primitiva Ellipse.

O parâmetro `fill` tem aqui o mesmo significado daquele utilizado em Circle e a Ellipse também fornece os pontos: n, s, e e w como

resultados.

A versão em LaTeX da primitiva Ellipse produz uma circunferência de raio t, onde $t = máximo(x, y)$. Isto, obviamente, descaracteriza o desenho. Portanto, toda figura que utiliza elipses deve ser chamada com a opção de utilização dos recursos gráficos de PostScript.

Line Esta primitiva está definida com o seguinte cabeçalho:

```
Figure Line ;
   Parameters{ lp } ;
   Results{ first , last } ;
```

onde `lp` é uma lista de pontos: $(p_1, p_2, p_3, \ldots, p_n)$.

Esta primitiva, quando chamada, produz uma figura em que os pontos da lista `lp` são unidos por segmentos de reta, na ordem em que aparecem na lista. Assim como as primitivas Arrow e Polygon, a primitiva Line tem uma característica diferente de todas as demais: possui um número variável de parâmetros, uma vez que `lp` pode ser constituída por n pontos. Obviamente, n deve ter valor mínimo igual a 2 para esta primitiva fazer sentido. Caso haja uma chamada a Line sem passagem de parâmetros, então a linguagem providencia automaticamente dois pontos para serem unidos, a saber: $p_1 = cc$ e $p_2 = (cc.x + dist * \cos\alpha, cc.y + dist * \text{sen}\alpha)$, onde: cc é o ponto definido pelas coordenadas correntes, $dist$ é o valor da distância-padrão sendo utilizado e α é o ângulo definido pela direção corrente.

Outra característica de Line (e também de Arrow) é que seus pontos de concatenação são os resultados **`first`** e **`last`**, que, por sua vez, coincidem com os pontos p_1 e p_n da lista `lp`, respectivamente. Veja na figura 4.5 um desenho produzido por esta primitiva.

A versão da primitiva Line em LaTeX apresenta certas restrições, uma vez que ela é traduzida em comandos **`\line`** de LaTeX (veja seção 3.1.1). Este comando (**`\line`**) consegue traçar uma classe limitada de segmentos (apenas 24 inclinações por quadrante). Deste modo, se um segmento de reta com inclinação α for requisitado, verifica-se se α pertence ao conjunto das inclinações possíveis. Se isto não ocorrer, então toma-se em seu lugar a inclinação deste conjunto mais próxima a α. Com isto, obviamente, pode haver uma descaracterização completa do desenho.

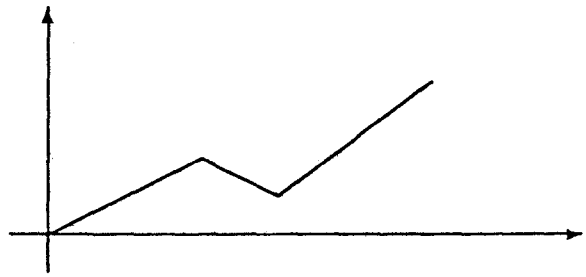


Figura 4.5: Desenho produzido pela primitiva Line (exceto os eixos).

Rectangle Esta primitiva está definida com o seguinte cabeçalho:

```
Figure Rectangle ;
   Parameters{width=1,height=1,fill=false} ;
   Combinations{ 1 : width , height ;
                 2 : width , height , fill } ;
```

Ao ser chamada, esta primitiva fornece um retângulo em que a largura e a altura são dadas pelos parâmetros `width` e `height`, respectivamente. Se nenhum parâmetro efetivo for passado, então os valores padrões `width = 1` e `height = 1` são utilizados. O parâmetro `fill` tem aqui o mesmo significado daquele utilizado pela primitiva Circle. Veja na figura 4.6 um desenho produzido por esta primitiva.

Na implementação de Rectangle existe uma chamada à primitiva Line. Portanto, as limitações impostas à Line são também impostas à Rectangle.

Polygon Esta primitiva está definida com o seguinte cabeçalho:

```
Figure Polygon ;
   Parameters{ lp } ;
```

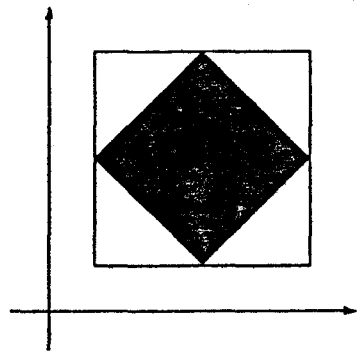


Figura 4.6: Desenho produzido pela primitiva Rectangle (exceto os eixos).

onde lp é uma lista de pontos: $(p_1, p_2, p_3, \ldots, p_n)$.

Ao ser chamada, esta primitiva produz um polígono cujos vértices são os pontos da lista de pontos lp. Veja na figura 4.7 um desenho produzido por esta primitiva.

Na implementação desta primitiva existe uma chamada à Line, sendo que o primeiro ponto da lista lp, p_1, é também passado como último ponto, p_n. Isto faz o que todos os pontos sejam unidos, formando um polígono.

Todas as restrições impostas à Line são impostas também à Polygon.

Arrow Esta primitiva está definida com o seguinte cabeçalho:

```
Figure Arrow ;
   Parameters{ lp } ;
   Results{ first , last } ;
```

onde lp é uma lista de pontos: $(p_1, p_2, p_3, \ldots, p_n)$.

Esta primitiva é semelhante à Line. A única diferença é que na extremidade determinada pelo ponto p_n é desenhada a ponta de uma seta. Veja na figura 4.8 um desenho produzido por esta primitiva.

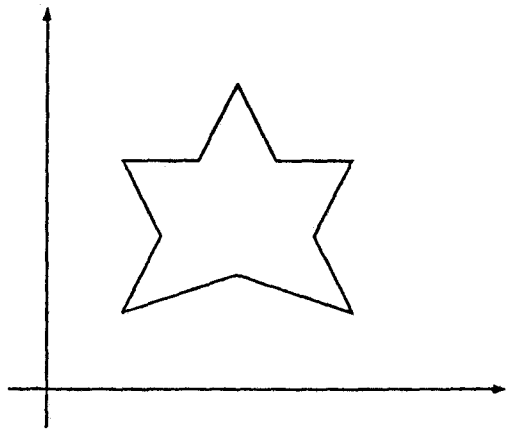


Figura 4.7: Desenho produzido pela primitiva Polygon (exceto os eixos).

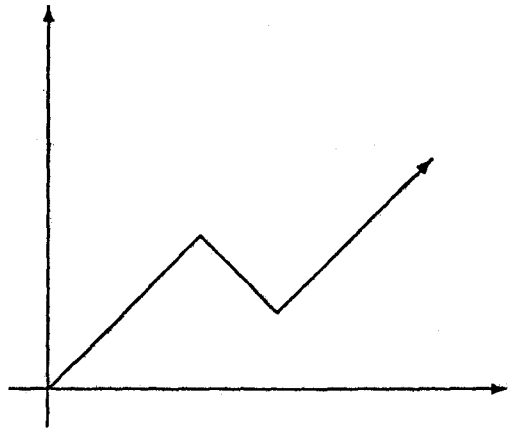


Figura 4.8: Desenho produzido pela primitiva Arrow (inclusive os eixos).

Na versão em LaTeX desta primitiva, o segmento de reta definido pelos pontos p_{n-1} e p_n é implementado através do comando `\vector` do LaTeX para se produzir o efeito de seta desejado, porém, este comando é ainda mais restrito do que o comando `\line` no tocante à inclinações (ele permite apenas 12 inclinações por quadrante), restringindo ainda mais o campo de aplicação desta primitiva.

Os pontos de concatenação da primitiva Arrow são seus resultados `first` e `last`.

Arc Esta primitiva está definida com o seguinte cabeçalho:

```
Figure Arc ;
   Parameters{ang_from = 0,ang_to = 90,radius = 1,
              p1 = [0,0],p2 = [1,1],p3 = [2,0]} ;
   Combinations{ 1 : ang_from , ang_to , radius ;
                 2 : p1 , p2 , p3 } ;
   Results{first , last} ;
```

Ao chamá-la, se a combinação passada for a de número 1, obtém-se o arco de uma circunferência cujo raio mede `radius` centímetros, com os pontos inicial e final sendo determinados pelos ângulos `ang_from` e `ang_to`, respectivamente. Caso contrário, se a combinação for a de número 2 , então os pontos inicial e final do arco correspondem aos pontos p1 e p3, enquanto que o raio e a origem da circunferência são automaticamente calculados. Os pontos de concatenação desta primitiva, assim como as primitivas Arrow e Line, também são os seus resultados `first` e `last` que correspondem aos pontos que determinam o início e o fim do arco. Veja na Figura 4.9 um desenho que inclui a utilização desta primitiva.[3]

Não foi implementada a tradução da primitiva Arc para o LaTeX, portanto, figuras que utilizam esta primitiva devem ser, necessariamente, traduzidas para PostScript.

Text Esta primitiva está definida com o seguinte cabeçalho:

```
Figure Text ;
   Parameters{ txt } ;
```

[3] A especificação em FIG desta figura está exibida no Apêndice E.

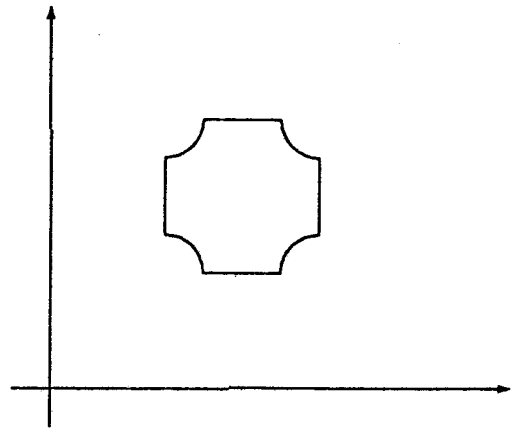


Figura 4.9: Desenho produzido incluindo o uso da primitiva Arc.

Ao ser chamada, esta primitiva produz, como resultado, a impressão da cadeia de caracteres armazenada em `txt`. Veja na figura 4.10 um desenho que inclui a utilização desta primitiva.[4]

A primitiva Text é, sem dúvida, a mais complexa e, portanto, a mais difícil de ser implementada. Esta dificuldade também decorre do fato desta primitiva ter que ser tratada como uma figura qualquer.

A primitiva Text implementada possui algumas restrições em ambas as versões(LaTeX e PostScript). Para amenizar o efeito dessas restrições, foram adicionados dois novos comandos na linguagem FIG: `Latext` e `Postscriptext`. Ambos os comandos permitem que o programador FIG, ao especificar uma figura, utilize *linguagem de baixo nível* em certos pontos do programa. Tudo o que não for possível de ser especificado em FIG pode sê-lo, no caso, diretamente em LaTeX ou em PostScript. A sintaxe destes comandos é a seguinte: `Latext "....."` e `Postscriptext "....."`. Tudo o que se encontra entre aspas é ignorado pelo compilador FIG e interpretado adiante pelo LaTeX ou pelo PostScript. Cada linha do código escrito em baixo nível deve ser inserida num comando próprio.

[4] A especificação em FIG desta figura está exibida no Apêndice E.

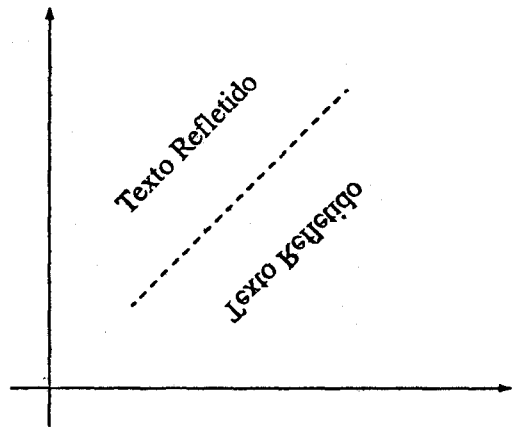


Figura 4.10: Desenho produzido incluindo o uso da primitiva Text.

Embora tenham sido decorrentes das restrições da primitiva Text, os comandos `Postscriptext` e `Latext` podem, obviamente, ser utilizados em outros contextos.

Em LaTeX, a primitiva Text é traduzida para um comando da forma `\makebox(0,0){txt}`, onde `txt` é a cadeia a ser impressa.

Em PostScript, a primitiva Text é traduzida para a seguinte seqüência de comandos:

```
/Times-Roman findfont
11 scalefont
setfont
gsave
[ m11 m12 m21 m22 m31 m32 ] concat
0 0 moveto
(txt) show
grestore
```

Esta seqüência faz com que a cadeia `txt` seja impressa na posição, orientação e escala indicados pela matriz de transformação M.

$$M = \begin{pmatrix} m11 & m12 & 0 \\ m21 & m22 & 0 \\ m31 & m32 & 1 \end{pmatrix}$$

A família de caracteres utilizada é a `Times-Roman`, tendo como fator de escala 11 pontos. A acentuação é possível desde que o usuário forneça os comandos de acentuação necessários juntamente como o texto a ser impresso. Deve-se tomar o cuidado de não se optar pela conversão de figuras para o PostScript, se estas possuirem primitivas Text cujas cadeias apresentam comandos de acentuação válidos em LaTeX, e vice-versa.

4.12.2 Determinação dos Retângulos Envolventes

Uma das características da linguagem FIG diz respeito ao cálculo automático do menor retângulo que envolve completamente uma figura "instanciada", retângulo este denominado simplesmente *retângulo envolvente.*

Como toda figura na linguagem FIG é construída sobre uma base formada por figuras primitivas, atacar o problema da determinação do retângulo envolvente de uma figura genérica, restringe-se em atacar o problema da determinação dos retângulos envolventes das primitivas.

A determinação dos retângulos envolventes das primitivas: Arrow, Polygon e Rectangle depende da determinação do retângulo envolvente da primitiva Line, já que em suas definições existem chamadas a tal primitiva. Pela mesma razão, a determinação do retângulo envolvente de Circle depende do retângulo envolvente de Ellipse. Portanto, a determinação do retângulo envolvente de qualquer figura nesta implementação recai sobre a determinação dos retângulos envolventes das primitivas: Line, Ellipse, Text e Arc.

Para se determinar um retângulo envolvente, basta determinar os seus pontos: *lb (left bottom)* e *rt (rigth top)* que, por sua vez, exige a determinação dos valores: *x-mínimo*, *x-máximo*, *y-mínimo* e *y-máximo* utilizados na construção efetiva da figura. Esta determinação é, de modo geral, dificultada pelas transformações de rotação e ampliação/redução, visto que elas alteram a orientação e a dimensão das figuras.

Para se determinar o retângulo envolvente da primitiva Line, aplica-se a matriz de transformação em cada ponto da lista `lp` e detecta-se os valores: *y-mínimo*, *y-máximo*, *x-mínimo* e *x-máximo*, dentre os novos pontos obtidos.

Para se determinar o retângulo envolvente da primitiva Ellipse, age-se da seguinte forma: aplica-se a matriz de transformações aos seus focos (F_1 e F_2) e ao ponto $(0, y)$. Com os novos pontos assim obtidos é possível determinar a equação da elipse transformada. Derivando-se esta equação em relação a x e a y, obtém-se os valores: *y-mínimo*, *y-máximo*, *x-mínimo* e *x-máximo*, desejados.

O retângulo envolvente da primitiva Arc é determinado da seguinte forma: toma-se como base o retângulo envolvente do arco definido com o mesmo raio e cujos pontos inicial e final coincidam. Devido às transformações geométricas, este arco é, de modo geral, uma elipse rotacionada em α graus. Este retângulo base é, então, recortado convenientemente utilizando-se os pontos críticos (em relação a x e em relação a y) da elipse e os pontos de origem e término do arco em questão.

Quanto ao retângulo envolvente da primitiva Text, ele não é calculado de forma precisa nesta versão da linguagem FIG; o valor de sua largura é obtido através da multiplicação do número de caracteres da cadeia a ser impressa por uma constante, enquanto que para a altura é atribuído uma outra constante. Estas constantes foram definidas de tal forma que o resultado final obtido não ficasse muito distante do resultado exato. Para se obter o resultado exato, é preciso que seja feito um estudo abrangente sobre a família de caracteres a ser utilizada na impressão das cadeias descritas através da primitiva Text.

Capítulo 5

Implementação do Integrador FIG-LaTeX

Neste capítulo são detalhados os aspectos do modelo adotado para a realização da integração da linguagem FIG ao sistema LaTeX. A este modelo denominou-se Integrador FIG-LaTeX.

Embora servindo apenas para integração a LaTeX, pode-se notar que, com poucas alterações, o modelo adotado também presta-se à integração da linguagem FIG ao sistema TeX, desde que sejam utilizados apenas os recursos gráficos de PostScript.

O modelo adotado é, na verdade, uma fusão do Modelo 1 com o Modelo 2 apresentados no Capítulo 4, ou seja, o produto final obtido pode ser uma descrição em LaTeX e/ou em PostScript das figuras especificadas inicialmente em FIG.

O Modelo 1 foi escolhido, a princípio, para levar a cabo esta integração, já que se estava interessado em manter o projeto independente de dispositivo de saída, o que descartava de pronto o Modelo 3. O Modelo 1 enquadra-se perfeitamente nas condições disponíveis para a realização deste projeto, porém, as limitações impostas pelos parcos recursos gráficos de LaTeX sempre foram um fator de preocupação e levaram à idealização do Modelo 2. Se por um lado o Modelo 2 impõe como premissa a disponibilidade de um interpretador para a linguagem PostScript que gera código em PCL (linguagem aceita pelo dispositivo de saída a ser utilizado), por outro, todas as potencialidades da linguagem FIG podem ser desenvolvidas utilizando-se os recursos gráficos da linguagem PostScript. Logicamente, se o dispositivo de saída hospedasse o interpretador para a linguagem PostScript (como é

o caso da impressora Apple LaserWriter [4]), o referido interpretador não seria necessário.

Ao traduzir uma figura especificada em PostScript para comandos PCL, o interpretador acaba gerando arquivos que excedem facilmente uma centena de kbytes, visto que em PCL as figuras têm que ser descritas em forma de mapas de bits. Em conseqüência disso, a execução do driver e a transmissão do arquivo gerado por esta execução para a impressora acabam tornando-se lentas. Optou-se, então, pela fusão do Modelo 1 com o Modelo 2, visando melhorar o desempenho do integrador, já que algumas figuras podem, sem prejuízos, ser traduzidas para LaTeX, tornando menos extensos os aquivos de saída.[1]

No modelo adotado, então, apenas as figuras que não são possíveis de serem construídas em LaTeX devem sê-lo em PostScript. Cabe ao usuário determinar se uma dada figura FIG deve ser traduzida para PostScript ou para LaTeX. O esquema de implementação do modelo adotado está exibido na Figura 5.1.[2]

Para a execução desta integração são necessários os seguintes recursos: o compilador Turbo Pascal - versão 5.0, o compilador FIG, o sistema LaTeX, o pré-processador (PrePro) para LaTeX, a impressora LaserJet series II da Hewlett Packard, o driver DVILASER/HP para a referida impressora, um interpretador para a linguagem PostScript (capaz de gerar código em PCL), o pré-processador (Filtro) para o referido driver e um microcomputador compatível com a linha IBM - PC com memória em disco rígido suficiente para instalação destes sistemas.

Dentre estes recursos, foram implementados neste trabalho: o compilador FIG, o pré-processador - PrePro - para LaTeX e o pré-processador - Filtro - para o driver. PrePro e Filtro são detalhados nas seções 5.2 e 5.4, respectivamente. Veja nas demais seções deste capítulo os detalhes do modelo adotado e no Apêndice D um exemplo completo de integração da linguagem FIG ao sistema LaTeX.

5.1 Os Arquivos

No modelo escolhido existem duas categorias de arquivos: arquivos fixos e arquivos variantes. Todos os arquivos fixos possuem o mesmo nome, por-

[1] Deve-se notar, no entanto, que os resultados finais obtidos via LaTeX e via PostScript não são totalmente uniformes.

[2] A especificação em FIG desta figura está exibida no Apêndice E.

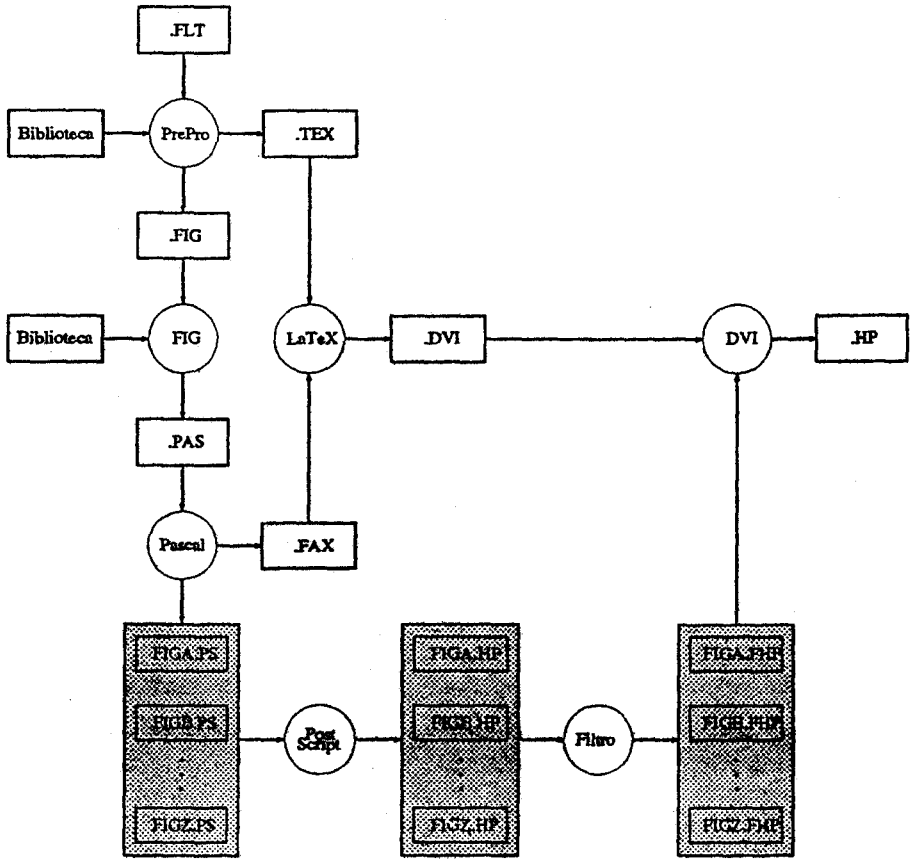


Figura 5.1: Esquema de implementação do Integrador FIG-LaTeX.

tanto, eles são referidos neste texto apenas pelas suas extensões. Os nomes destes arquivos são iguais ao nome do arquivo de entrada que é fornecido pelo usuário. Os demais arquivos (os arquivos variantes) mudam de execução para execução. A quantidade desses arquivos é dada pelo número de especificações de figuras existentes no arquivo de entrada que devem ser traduzidas para PostScript.

Os aquivos fixos são:

`.FLT`: (abreviação para FIG-LaTeX) é o arquivo de entrada do integrador. Ele é criado pelo usuário e constitui-se tanto de texto LaTeX quanto de texto FIG. Para estruturar o arquivo de entrada, foi criado o *ambiente FIG* (no sentido de "environment" de LaTeX) no qual devem ser inseridos única e exclusivamente textos escritos na linguagem FIG. Eliminando-se os ambientes FIG do arquivo `.FLT`, ele torna-se um típico arquivo `.TEX`, podendo ser submetido a LaTeX sem que isto implique em erros. Com isso, quer se dizer que a estrutura básica de um arquivo `.FLT` é a mesma de um arquivo a ser processado pelo sistema LaTeX. Na Figura 5.2 pode-se visualizar o aspecto geral de um arquivo `.FLT`.

Cabe observar que uma "instância" do ambiente FIG pode conter definições de novas figuras ou, exclusivamente, comandos da linguagem que não se constituem em definições de figuras, mas sim em comandos destinados à construção efetiva de desenhos. Neste último caso, o usuário pode, opcionalmente, indicar, através dos argumentos `p` ou `l`, se os recursos gráficos a serem utilizados na tradução do desenho são os de PostScript ou os de LaTeX, respectivamente. Se estes argumentos forem omitidos, então utiliza-se o argumento padrão especificado pelo usuário quando este executa o Integrador FIG-LaTeX. Por exemplo, `FiG-LaTeX Arq.FLT P` indica que a opção-padrão é PostScript.

`.FIG`: Este arquivo é gerado pelo pré-processador PrePro. Nele são convenientemente colocados todos os comandos e/ou declarações encontradas nos ambientes FIG do arquivo de entrada. Na Figura 5.3 é apresentado o aspecto geral de um arquivo `.FIG`.

As diretivas indicadas pelos símbolos `$B` e `$E` são discutidas na seção 5.3.

`.TEX`: Assim como `.FIG`, este arquivo é gerado pelo pré-processador PrePro. Ele constitui-se do texto propriamente dito e de todos os comandos e/ou declarações de LaTeX oriundas do arquivo de entrada.

```
\documentstyle[...]{...}
⋮
preâmbulo
⋮
\begin{document}
⋮
texto LaTeX
⋮
\begin{FIG}
⋮
texto FIG
⋮
\end{FIG}
⋮
texto LaTeX
⋮
\begin{FIG}[p]
⋮
texto FIG
⋮
\end{FIG}
⋮
texto LaTeX
⋮
\begin{FIG}[l]
⋮
texto FIG
⋮
\end{FIG}
⋮
texto LaTeX
⋮
\end{document}
```

Figura 5.2: Aspecto geral de um arquivo de entrada .FLT.

```
Picture figuras ;
!$B UnitA
Figure a1 ;
:
!$E
!$B UnitB
Figure am ;
:
!$E
:
Begin    ! programa principal
!$B FIGA L
:
!$E
:
!$B FIGN P
:
!$E
End Picture
```

Figura 5.3: Aspecto geral de um arquivo .FIG

```
\documentstyle[...]{...}
⋮
preâmbulo
⋮
\begin{document}
\input{.FAX}
⋮
\figa
⋮
\fign
⋮
\end{document}
```

Figura 5.4: Aspecto geral de um arquivo .TEX.

Além destes, existe também um comando para incluir o arquivo com extensão .FAX, que está abaixo descrito. Existem ainda no arquivo .TEX referências aos novos comandos que são definidos em .FAX. Estes novos comandos são inseridos no arquivo .TEX nos lugares correspondentes àqueles onde existem ambientes FIG no arquivo de entrada, que destinam-se a construções de desenhos. As posições no arquivo .TEX correspondentes àquelas ocupadas pelos ambientes FIG destinados a definições de figuras ficam vazias. Na Figura 5.4 é apresentado o aspecto geral de um arquivo .TEX.

.PAS: Este arquivo contém o código objeto em Pascal gerado pelo compilador FIG.

.FAX: A extensão FAX é uma abreviação para FIG AuXiliar. Este arquivo é gerado ao ser compilado e executado o programa que se encontra no arquivo .PAS pelo compilador Pascal. Nele estão as definições em LaTeX dos comandos \figa, figb, etc, que equivalem aos desenhos

especificados em FIG. Ele é incluído no arquivo .TEX onde são referenciados esses comandos. Um desenho descrito no arquivo .FLT dentro de um ambiente FIG seguido do argumento 1 é traduzido para LaTeX e definido como sendo um novo comando \figx, por exemplo. Caso contrário, se o ambiente FIG for seguido pelo argumento p, então o novo comando \figx existente no arquivo .FAX é definido como sendo a seguinte macro de TeX:[3]
\insertplot{arquivo}{altura}{margem esquerda},
onde \insertplot é definida como:

```
\def\insertplot #1 #2 #3 {\par
        \hbox{%
           \hskip #3
           \vbox to #2 {
               \special{hp:plotfile #1}
               \vfill
             }%
           }
         }
```

Os argumentos desta macro são: nome do arquivo que contém a descrição em PCL da figura especificada inicialmente em FIG, a altura total da figura e o seu deslocamento em relação à margem esquerda do texto, nesta ordem. É importante frisar que esta macro produz o efeito descrito somente se o driver utilizado interpretar o comando \special de forma adequada (veja o Capítulo 1).

.DVI: Este arquivo é gerado pelo LaTeX e possui a descrição independente de dispositivo de saída das páginas do texto que está sendo processado.

.HP: Este arquivo é o produto final obtido no processo de integração. Ele é gerado pelo driver DVILASER/HP em linguagem PCL e, por isso, está em formato apropriado para ser transmitido à impressora LaserJet series II.

Como já foi dito, a quantidade de arquivos variantes deste modelo é determinada pelo número de ambientes FIG existentes no arquivo .FLT, cujos

[3] Esta macro está definida em [5].

conteúdos são descrições de desenhos a serem traduzidas para PostScript. Existem três classes de arquivos variantes, a saber:

.PS: Estes arquivos têm extensões .PS por serem constituídos de programas PostScript. Quanto aos seus nomes, eles são do tipo FIGA, FIGB, FIGAAB, etc. Tais arquivos são gerados ao ser compilado e executado pelo Pascal o programa contido no arquivo .PAS. Cada arquivo .PS contém a definição em PostScript de uma figura especificada em FIG num ambiente \begin{FIG}[p]...\end{FIG} no arquivo de entrada.

.HP: A interpretação de cada arquivo .PS pelo interpretador PostScript gera um arquivo .HP correspondente, com mesmo nome. Tais arquivos são constituídos de descrições na linguagem PCL das figuras especificadas nos arquivos .PS.

.FHP: Ao gerar os arquivos .HP, o interpretador PostScript acrescenta-lhes alguns comandos de controle indesejáveis do ponto de vista deste projeto de integração. Portanto, os arquivos .HP são submentidos a um processo de filtragem, a fim de eliminar tais comandos. Os arquivos obtidos a partir deste processo possuem extensão .FHP e são inseridos, finalmente, no restante do texto através do comando \special da macro \insertplot existente no arquivo .FAX.

5.2 O Pré-Processador PrePro

O pré-processador PrePro[4] tem um trabalho relativamente simples: ele recebe o arquivo de entrada .FLT e agrupa adequadamente no arquivo .FIG os trechos que estão escritos nos ambientes FIG. O resto do texto ele insere no arquivo .TEX, juntamente com os comandos \input{.FAX}, \figa, \figb, etc.

Os arquivos de entrada podem ser compostos por vários ambientes FIG, sendo que alguns deles destinam-se à definições de novas figuras, enquanto que outros à construção de desenhos que, eventualmente, utilizam tais figuras. Ao detectar que um ambiente FIG é destinado à definição de novas figuras, então tudo o que estiver escrito naquele ambiente é agrupado por um par de diretivas ($B,$E) e transferido para o arquivo .FIG. Caso contrário, tudo o que estiver no referido ambiente é agrupado por um par de diretivas ($B,$E)

[4] A princípio, poder-se-ia empregar, alternativamente, o arquivo .TEX com *environment* auto-processante, tornando dispensável PrePro.

e copiado em um arquivo auxiliar .AUX. No final do pré-processamento, o arquivo .AUX contém, na verdade, o corpo principal de um programa FIG e é anexado ao arquivo .FIG que, por sua vez, é composto pela seção de declarações de figuras do referido programa. Obviamente, no arquivo .AUX são convenientemente inseridas as seqüencias: "Begin" e "End Picture", enquanto que no .FIG é inserida a seqüência "Picture Figuras;". Como conseqüência deste tipo de análise efetuada pelo pré-processador, no arquivo de entrada pode-se inserir ambientes FIG destinados à construção de desenhos antes mesmo dos ambientes FIG nos quais estão definidas as figuras utilizadas por estes desenhos. Por outro lado, se na definição de uma figura X "instancia-se" uma figura Y, então o ambiente FIG no qual Y é definida deve, obrigatoriamente, preceder o ambiente no qual define-se X para não ocorrer um erro durante a compilação do programa.[5]

Se o primeiro símbolo escrito num ambiente FIG for Figure, então o pré-processador trata-o como sendo um ambiente destinado a conter apenas definições de figuras; caso contrário, trata-o como destinado a conter construções de desenhos.

5.3 O Compilador FIG

O compilador FIG recebe do pré-processador PrePro o arquivo .FIG que deve conter um programa fonte escrito na linguagem FIG. Se não ocorrer nenhum erro, o compilador gera, como código objeto, um programa escrito na linguagem Pascal e o armazena no arquivo .PAS. Como no Capítulo 4 trata-se especificamente do compilador FIG, esclarece-se nesta seção apenas as funções das diretivas $B e $E. Tais diretivas têm duas funções distintas no processo de compilação. Quando inseridas no corpo do programa principal, elas servem para delimitar os comandos FIG que fazem parte de um mesmo desenho. Estes comandos são traduzidos numa macro de LaTeX (através do comando \newcommand) que é referenciada no texto no local onde o desenho que ela representa deve ser construído ou, então, eles são traduzidos para PostScript e armazenados num arquivo .PS (veja a seção 5.1). Ao encontrar uma diretiva $B no corpo do programa principal, o compilador atribui à variável denominada *opção* o argumento p ou l, argumento este que sucede tal diretiva (!$B figa p, por exemplo). Como todos os procedimentos referentes às figuras FIG são chamados com a variável *opção* sendo passada

[5] Poder-se-ia resolver este problema através da geração de *declarações forward*, tornando possível definições de figuras mutuamente recursivas.

como parâmetro (veja a seção 4.7), ao serem executados, eles geram código em LaTeX ou em PostScript, dependendo do valor desta variável. Se o valor for `l`, gera-se código em LaTeX no arquivo `.FAX`, senão, se o valor for `p`, gera-se código em PostScript nos arquivos `.PS`.

Quando inseridas na seção de definição de novas figuras, elas servem para indicar quantas unidades (*Units*) serão utilizadas para armazenar o programa Pascal resultante da tradução e determinar quais são as figuras que compõem cada unidade. Inicialmente, todas as figuras eram traduzidas em um único módulo, porém, como o tamanho de tais programas excediam facilmente os 64 kbytes, decidiu-se subdividir tais programas em várias unidades (que são compiladas separadamente). Desse modo, o limite máximo que os programas gerados a partir da compilação podem ter para poderem ser executados passou a ser dado pela quantidade de memória do microcomputador.

As diretivas `$B` e `$E` são geradas pelo pré-processador PrePro a partir das disposições dos ambientes FIG nos arquivos de entrada (`.FLT`). Isto quer dizer que cabe ao usuário definir quais figuras farão parte da mesma unidade e quais comandos especificam determinado desenho.

Cada "instância" do ambiente FIG do arquivo `.FLT` corresponde a uma unidade do programa Pascal gerado, caso este ambiente seja destinado simplesmente à definição de figuras.

5.4 O Processo de Filtragem

O programa responsável pelo processo de filtragem dos arquivos `.HP` denomina-se Filtro. Este programa recebe do interpretador PostScript[6] arquivos cujos conteúdos estão descritos em linguagem PCL e retira-lhes todos os comandos cujos efeitos impossibilitam a inserção adequada das figuras no resto do texto. Um arquivo `.HP` compõe-se, em geral, dos seguintes comandos:

- `^&l#X`: designa o número de cópias que devem ser impressas para cada página.
- `^E`: este comando serve para "inicializar" a impressora.
- `^*t#R`: designa a resolução a ser utilizada na impressão dos *dados de varredura* transmitidos à impressora.

[6]Devido ao interpretador utilizado nesta implementação, os desenhos cujas especificações em FIG serão traduzidas para PostScript devem ser posicionados completamente no primeiro quadrante.

- `^*p#Y`: designa a posição vertical do cursor na página.
- `^*r#A`: indica o início da transferência de um *gráfico de varredura.*
- `^*rB`: indica o final da transferência de um *gráfico de varredura.*
- `^*b#W[.....]`: é utilizado para transferir uma linha de *dados de varredura* para a impressora. Esta linha está representada por `[.....]`.

O processo de filtragem elimina os comandos: `^E` e `^*p#Y` e mantém os demais: `^&l1X`, `^*t300R`, `^*r0A` e todos os comandos de transferência de linhas de *dados de varredura.* O resultado da filtragem é armazenado no arquivo `.FHP`.

Utiliza-se o símbolo ^ para representar o caractere Escape.

5.5 A Biblioteca

As primitivas da linguagem FIG fazem parte de um módulo pré-compilado que, por sua vez, faz parte da biblioteca do sistema. São as primitivas que fazem chamadas às rotinas do pacote gráfico que efetivamente traçam a figura. O fato das primitivas ficarem num módulo à parte dá ao compilador uma certa flexibilidade, pois basta alterá-las convenientemente sempre que outro pacote gráfico for utilizado. Na biblioteca encontram-se também rotinas responsáveis pelas transformações geométricas, pela atualização dos retângulos envolventes das figuras, pelo gerenciamento de arquivos, etc.

Capítulo 6

Considerações Finais

Ao definir a linguagem FIG [22], J. C. Setubal propôs sua integração a sistemas de composição gráfica, onde suas características seriam particularmente úteis. O projeto da linguagem FIG, bem como esta proposição de integração a sistemas de composição gráfica foram baseados, em parte, nas linguagens PIC [14,15] e IDEAL [24,25] e em suas respectivas integrações ao sistema TROFF[1] [21]. Nestas integrações, ambas as linguagens funcionam como pré-processadores para TROFF.

O projeto de integração da linguagem FIG ao sistema LaTeX [19] apresentado neste texto é uma materialização desta proposição e tem como objetivo principal colocar à disposição dos usuários de LaTeX uma opção a mais para que estes possam inserir figuras nos textos que desejam compor.

Esta opção vem acompanhada de algumas vantagens, tais como: utilização de equipamentos simples para processamento dos textos e das figuras, facilidade na especificação de figuras com estruturas hierárquicas complexas e, por fim, integração harmoniosa de texto e figuras, visto que em ambos os casos a abordagem no tratamento não é interativa. Esta última "vantagem" deixa de sê-lo e passa a ser uma desvantagem caso o usuário não tenha experiência em programação. Neste caso, principalmente, as ferramentas para construção de figuras em que a abordagem é interativa, como é o caso do AutoCad [8], são muito mais *amigáveis*[2].

O fato da linguagem FIG ser muito parecida sintaticamente com a linguagem Pascal contribui para sua assimilação, visto que Pascal é uma linguagem bastante difundida.

[1] O sistema TROFF é um compositor de textos existente em ambiente UNIX.

[2] *user-friendly*.

Como foi visto, no modelo de integração adotado para implementação do Integrador FIG-LaTeX, as figuras podem ser traduzidas para LaTeX ou para PostScript [1,2,3,13], dependendo da opção do usuário. Com a atual tendência de PostScript vir a se tornar um padrão para descrição de páginas e, como conseqüência, vir a ficar residente nos dispositivos de saída (como é o caso da impressora com tecnologia laser, LaserWriter, da Apple Computer, Inc. [4]), os esforços futuros a serem despendidos neste projeto devem ser canalizados nesta direção, o que significa incrementar o conjunto de primitivas da linguagem FIG de modo a aproveitar ao máximo os recursos de PostScript e implementar conceitos que não foram tratados nesta versão da linguagem, tais como: janelas ("windows"), atributos, etc. A primitiva Text, em particular, pode vir a oferecer muito mais recursos, principalmente no que diz respeito à *família de caracteres.*

Após o término da implementação deste integrador, foram realizados alguns testes para verificar seu desempenho. O maior destes testes resultou na composição deste texto. Nele, todas as figuras foram especificadas em FIG e inseridas automaticamente no texto. As especificações em FIG de várias destas figuras podem ser encontradas no Apêndice E.

Concluiu-se, após esta experimentação, que seria valiosa a inclusão na linguagem FIG do conceito de *variáveis indexadas.* Concluiu-se, também, que haveria um certo ganho se a linguagem FIG tivesse sido implementada através de um interpretador. Tal conclusão derivou-se do fato do código objeto gerado pelo compilador ficar, de modo geral, bastante extenso, o que exige a sua decomposição em módulos. Este fato, aparentemente não chega a comprometer o processo de integração, porém, contribui negativamente nas medidas de desempenho quanto ao aspecto *espaço utilizado.* A continuidade deste projeto deve envolver um estudo completo do código objeto gerado pelo compilador, visando sua melhoria.

Outro fator complicador, como já foi dito, deriva das duplas "instanciações" das figuras.

O que, sem dúvidas, traria grandes benefícios para este integrador seria a utilização de um dispositivo de saída que hospedasse um interpretador para a linguagem PostScript. Isto evitaria a geração dos arquivos auxiliares (`.HP` e `.FHP`) levando a uma economia considerável de tempo e memória.

Contudo, diante dos recursos disponíveis, acredita-se que os resultados das experimentações realizadas foram bastante satisfatórios, principalmente se for levado em conta o fato de que algumas outras formas de inserções de figuras no ambiente TeX sendo utilizadas atualmente parecem, de uma forma geral, apresentar problemas análogos.

Outros projetos têm sido realizados visando possibilitar a fácil inserção de figuras em documentos a serem compostos em ambiente TEX. Recentemente, Micah Beck criou um pacote, ao qual denominou TransFig [6,7], que torna possível tal inserção. O usuário utiliza um editor gráfico para especificar suas figuras que são convertidas em código Fig[3]. Este código é traduzido, segundo opção do usuário, em comandos PostScript, LATEX, PICTEX, PIC ou (E)EPIC, que, finalmente, podem ser convenientemente agregados ao resto do documento.

TransFig é, nitidamente, mais abrangente e poderoso do que o projeto apresentado neste texto. Deve-se ressaltar, no entanto, que as filosofias dos projetos são distintas e que existem aplicações em que a linguagem FIG apresenta melhores resultados. A especificação de uma árvore binária completa é um exemplo de aplicação em que FIG leva vantagem sobre Transfig. A especificação de tal figura, via FIG, pode ser feita para uma árvore genérica (com altura n), fazendo uso de sua característica recursiva. Posteriormente, esta especificação pode ser facilmente alterada. Em Transfig, a construção desta mesma figura deve ser feita passo a passo, através de um editor gráfico, duplicando subpartes, quando possível; futuras alterações poderiam implicar na reconstrução completa da figura.

[3]Tanto o editor, quanto o código Fig, foram originalmente desenvolvidos por Supoj Sutanthavibul, na Universidade do Texas, tendo em comum com a linguagem FIG, desenvolvida por J. C. Setubal, apenas o nome.

Apêndice A

Operações e Funções disponíveis em FIG

Operação	Tipo dos Operandos	Operador
Divisão	Numérico	/
Multiplicação	Numérico	*
Adição	Numérico	+
Subtração	Numérico	−
Potenciação	Numérico	**
Resto da Divisão	Numérico	*Mod*
Concatenação	Cadeia	+
Disjunção	Booleano	*Or*
Conjunção	Booleano	*And*
Negação	Booleano	*Not* (unário)
Mudança de Sinal	Numérico	− (unário)
Comparação (igual)	Numérico	=
Comparação (diferente)	Numérico	<>
Comparação (maior)	Numérico	>
Comparação (menor)	Numérico	<

Nome da Função	Tipo da Função	Tipo do Argumento
Sin	Numérico	Numérico
Cos	Numérico	Numérico
ArcTan	Numérico	Numérico
Ln	Numérico	Numérico
Abs	Numérico	Numérico
Frac	Numérico	Numérico
Int	Numérico	Numérico
Round	Numérico	Numérico
Trunc	Numérico	Numérico
Exp	Numérico	Numérico
Dist	Numérico	Ponto,Ponto
StrToNum	Numérico	Cadeia
NumToStr	Cadeia	Numérico,Numérico,Numérico

Exceto Dist, NumToStr e StrToNum, as demais funções são comumente encontradas em várias outras linguagens e têm o mesmo significado usualmente empregado. Dist calcula a distância entre os pontos que lhe são passados como parâmetros, NumToStr converte um número em uma cadeia e StrToNum faz a conversão inversa de NumToStr.

Apêndice B

Alterações feitas na linguagem FIG

Segue-se abaixo a relação de pequenas alterações efetuadas na linguagem FIG apresentada em [22]. Tais alterações visaram simplificá-la e adaptá-la ao contexto da integração ao ambiente TEX. As modificações são as seguintes:

1. O conceito de janela não foi implementado. Portanto, durante a especificação de uma figura não se declara *altura*, *largura* e nem *origem*.

2. As palavras reservadas da linguagem foram traduzidas para o inglês para não destoar dos comandos e declarações do ambiente TEX, que estão neste idioma.

3. Os comandos devem ser seguidos pelo caractere ";". Permite-se, no entanto, a omissão dos ponto-e-vírgulas nos casos em que os comandos vierem seguidos dos átomos `END` ou `}`.

4. Foram introduzidas duas diretivas : **$B** e **$E**, que dizem ao compilador quais são os comandos FIG que fazem parte da definição de uma mesma figura. Assim, para cada par de diretivas **$B** e **$E**, corresponde uma definição de novo comando em LATEX ou em PostScript no arquivo .FAX. As diretivas também especificam quais figuras devem ser traduzidas numa mesma unidade ("Unit") do programa Pascal gerado. Exceto as diretivas, todo texto numa determinada linha precedido pelo símbolo "!" é considerado comentário e, portanto, ignorado pelo compilador.

5. Foram acrescentados os tipos *booleano* e *direção*. O tipo *enumerado* não foi implementado. No final da compilação os tipos das variáveis são verificados, se houver alguma com tipo indefinido, então o usuário será alertado sobre a existência de uma variável que certamente não tem utilidade no programa.

6. Não existem variáveis globais, exceto aquelas pré-definidas na linguagem, tais como : `dist_df`, `height_df`, `uni_ang`, `radius_df`, etc.

7. Não se permite que um mesmo identificador seja declarado como resultado e parâmetro de uma mesma figura.

8. Não permite-se a utilização do rótulo numérico 0 (*zero*), na declaração das combinações.

9. As figuras primitivas disponíveis são: *Line*, *Rectangle*, *Circle*, *Text*, *Ellipse*, *Arrow*, *Polygon* e *Arc*. O conceito de atributos não foi implementado da forma apresentada no projeto original da linguagem.

10. Foram criados dois novos comandos: `Latext` e `Postscriptext`, cujas sintaxes são da forma:`Latext"....."` e `Postscriptext"....."`. O texto encontrado entre aspas é simplesmente ignorado pelo compilador FIG e "passado" adiante para ser tratado pelo LaTeX ou pelo PostScript, respectivamente. Obviamente, tal texto deve ser escrito respeitando-se a sintaxe de LaTeX (ou melhor, a sintaxe do ambiente *picture* de LaTeX) ou de PostScript. Se por ventura desejar-se aspas dentro do texto, deve-se demonstrar isso colocando-se duas aspas seguidas. Estes novos comandos permitem ao programador FIG especificar alguns trechos da figura, ou mesmo toda a figura, em *baixo nível*. Isto pode ser muito útil quando for necessária a descrição de textos numa forma mais complexa do que aquela permitida pela primitiva *Text*, por exemplo. Cada "instância" destes comandos deve conter uma linha do código escrito em *baixo nível*.

11. Nas operações de rotação e escala, o ponto padrão utilizado não é mais *cc*, mas sim [0,0].

Apêndice C

Erros Detectáveis pelo Compilador FIG

Número do Erro	Mensagem
101	número mal definido.
102	(esperado.
103	) esperado.
104	{ esperado.
105	} esperado.
106	[esperado.
107	] esperado.
108	, esperado.
109	; esperado.
110	| esperado.
111	= esperado.
112	" esperado.
113	THEN esperado.
114	TIMES esperado.
115	FROM esperado.
116	TO esperado.
117	DO esperado.
118	Número esperado.
119	BEGIN esperado.

Número do Erro	Mensagem
120	END esperado.
121	PICTURE esperado.
122	X ou Y esperado.
123	. esperado.
124	Identificador esperado.
125	Rótulo numérico esperado.
126	Identificador de comando esperado.
127	Símbolo errado em expressão.
128	Função não definida.
129	!$B esperado.
130	!$E esperado.
131	Arquivo com extensão .FIG esperado.
132	P ou L esperado.
201	A categoria do identificador é incompatível com o contexto.
202	Tipos incompatíveis.
203	Constante de tipo numérico esperada.
204	Constante de tipo booleano esperada.
205	Constante de tipo cadeia esperada.
206	Constante de tipo ponto esperada.
207	Tipo numérico esperado.
208	Variável com tipo indefinido.
209	Identificador não declarado.
210	Identificador redeclarado.
211	Identificador não declarado como resultado desta figura .
212	Parâmetro não declarado .

Número do Erro	Mensagem
213	Este parâmetro já foi passado .
214	Este identificador deve ser parâmetro formal .
215	Parâmetro sem valor-padrão.
216	Combinação inválida.
217	Rótulo numérico não pode ser 0.
218	Este identificador não pode ser rótulo.
219	Rótulo não associado a figura.
220	Resultados mal declarados.
221	Constante redeclarada.
222	Identificador de tipo ponto esperado.

Apêndice D

Exemplo de Integração FIG-LaTeX

Neste apêndice é apresentado um exemplo de integração da linguagem FIG ao sistema LaTeX. Tal apresentação é feita através da exibição dos vários arquivos envolvidos no processo. A integração aqui exemplificada resultou no primeiro capítulo deste texto.

Arquivo de entrada CAPI.FLT: Este arquivo contém, além do texto propriamente dito, comandos LaTeX intercalados com especificações de figuras descritas em FIG.

```
\chapter{Introdu\c{c}\~{a}o}
\label{introducao}
\begin{FIG}
  Figure Sub_Arvore ;
   Parameters{ h = 0 } ;
   Results{ p1 , p2 , p3 } ;
     Begin
       If h = 0
          Then{r : Circle 0.75 ;
                p1 := r.lb ;
                p2 := r.tc ;
                p3 := r.rb }
          Else{t1 : Sub_Arvore h-1 ;
                t2 : Sub_Arvore h-1 With .p1 At t1.p3 + [1,0] ;
                rz : Circle 0.75 At [t1.p3.x + 0.5,t1.p2.y + 1] ;
                Arrow rz.lc , t1.p2 ;
                Arrow rz.rc , t2.p2 ;
                p1 := t1.p1 ;
                p2 := rz.tc  ;
```

```
                p3 := t2.p3  }
    End ;
  \end{FIG}

Com a dissemina\c{c}\~{a}o da inform\'{a}tica nos mais
diversos setores da sociedade,  ...
... a que apresenta-se na Figura~\ref{fig1}.

\begin{figure}
    \begin{FIG}[p]
       scale 0.6 Sub_Arvore 3 At [10.75,10] ;
    \end{FIG}
    \caption{\'{A}rvore bin\'{a}ria completa de altura 3.}
    \label{fig1}
\end{figure}

A linguagem utilizada para a especifica\c{c}\~{a}o de
figuras em \LaTeX\ \'{e} ...
... texto visa, ent\~{a}o, apresentar aspectos da
implementa\c{c}\~{a}o de uma  vers\~{a}o da linguagem
FIG e mostrar alguns modelos que permitem a inclus\~{a}o
autom\'{a}tica de figuras  especificadas nesta linguagem,
no ambiente \TeX.
```

Arquivo CAPI.TEX: Este arquivo deriva do arquivo de entrada e contém apenas comandos LaTeX. Os locais onde haviam especificações de figuras estão vazios ou possuem instâncias de novos comandos que foram definidos e armazenados no arquivo CAPI.FAX.

```
\input{CAPI.FAX}
\chapter{Introdu\c{c}\~{a}o}
\label{introducao}

Com a dissemina\c{c}\~{a}o da inform\'{a}tica nos mais
diversos setores da sociedade, ...
... a que apresenta-se na Figura~\ref{fig1}.

\begin{figure}
    \vspace{1cm}
       \FIGa
    \caption{\'{A}rvore bin\'{a}ria completa de altura 3.}
    \label{fig1}
\end{figure}

A linguagem utilizada para a especifica\c{c}\~{a}o de
figuras em \LaTeX\ \'{e} ...
... texto visa, ent\~{a}o, apresentar aspectos da
implementa\c{c}\~{a}o de uma vers\~{a}o da linguagem
```

```
FIG e mostrar alguns modelos que permitem a inclus\~{a}o
autom\'{a}tica de figuras especificadas nesta linguagem,
no ambiente \TeX.
```

Arquivo CAPI.FIG: Este arquivo possui o programa FIG que foi extraído do arquivo de entrada CAPI.FLT.

```
Picture figuras ;
 !$B UnidA
 Figure Sub_Arvore ;
  Parameters{ h = 0 } ;
  Results{ p1 , p2 , p3 } ;
  Begin
   If h = 0
    Then{r : Circle 0.75 ;
         p1 := r.lb ;
         p2 := r.tc ;
         p3 := r.rb }
    Else{t1 : Sub_Arvore h-1 ;
         t2 : Sub_Arvore h-1 With .p1 At t1.p3 + [1,0] ;
         rz : Circle 0.75 At [t1.p3.x + 0.5,t1.p2.y + 1] ;
         Arrow rz.lc , t1.p2 ;
         Arrow rz.rc , t2.p2 ;
         p1 := t1.p1 ;
         p2 := rz.tc  ;
         p3 := t2.p3  }
  End ;
!$E
Begin
!$B FIGa P
  scale 0.6 Sub_Arvore 3 At [10.75,10] ;
!$E
End Picture
```

Arquivo CAPI.PAS: Este arquivo contém o código gerado pelo compilador FIG, quando este processa o programa armazenado no arquivo CAPI.FIG.

```
USES procpdef , declarac , UnidA ;

BEGIN
  uni_ang := pi/180 ;
  dist_df := 1 ;
  orig_atu := cc ;
  visib := true ;
  opcao := 'L' ;
  Open_Arq('CAPI.fax') ;
```

```
T_Set_Ident(mtc) ;
Fig_T_Scale_Zero( 1.0, 1.0, mtc) ;
FOR i := 1 TO 2 DO
  BEGIN
     cd := east ;
     opcao := 'P' ;
     cc.x := 0 ;
     cc.y := 0 ;
     visib := not(visib) ;
     IF visib THEN Init_New_Command(opcao,'CAPI');
     Fig_R_Init(rtv_corr) ;
     mtc_aux := mtc ;
     T_Set_Ident(matriz) ;
     Fig_T_Scale( 0.600, 0.600,0,0,matriz) ;
     ponto_aux.x := 0 ; ponto_aux.y := 0 ;
     SUB_ARVORE(ponto_aux,ponto_aux,sub_arvore1_aux,false,matriz,
                cd,0,opcao,3.000) ;
    pto1.x := 10.750 ;
    pto2.x := sub_arvore1_aux.fixo.lb.x + (sub_arvore1_aux.fixo.rt.x
              - sub_arvore1_aux.fixo.lb.x)/2;
    pto1.y := 10.000 ;
    pto2.y := sub_arvore1_aux.fixo.lb.y + (sub_arvore1_aux.fixo.rt.y
              - sub_arvore1_aux.fixo.lb.y)/2 ;
    d_x := pto1.x - pto2.x ; d_y := pto1.y - pto2.y ;
    T_Transform(pto1.x,pto1.y,pto1.x,pto1.y,mtc_aux) ;
    T_Transform(pto2.x,pto2.y,pto2.x,pto2.y,mtc_aux) ;
    desl_x := pto1.x - pto2.x ; desl_y := pto1.y - pto2.y ;
    Fig_T_Scale( 0.600,0.600,0,0,mtc_aux) ;
    Fig_T_Translate(desl_x,desl_y,mtc_aux) ;
    ponto_aux.x := orig_atu.x + desl_x ;
    ponto_aux.y := orig_atu.y + desl_y ;
    SUB_ARVORE(orig_atu,ponto_aux,sub_arvore1,visib,mtc_aux,
               cd,0,opcao,3.000) ;
    Fig_R_Atu_R(sub_arvore1.fixo,rtv_corr) ;
    T_Set_Ident(matriz) ;
    Fig_T_Translate(d_x,d_y,matriz) ;
    AtualizRetenv(sub_arvore1_aux.fixo,matriz) ;
    WITH sub_arvore1_aux  DO
       BEGIN
          T_Transform(P1.x,P1.y,P1.x,P1.y,matriz) ;
          T_Transform(P2.x,P2.y,P2.x,P2.y,matriz) ;
          T_Transform(P3.x,P3.y,P3.x,P3.y,matriz) ;
       END ;
    cc.x := sub_arvore1_aux.fixo.post_conc.x ;
    cc.y := sub_arvore1_aux.fixo.post_conc.y ;
  END ;
Fin_New_Command(opcao);
Close_Arq ;
```

```
END.
```

No arquivo UNIDA.PAS encontra-se a unidade UnidA cujo código Pascal correspondente à tradução da sub-figura SUB_ARVORE.

No arquivo PROCPDEF.PAS (Procedimentos Pré-definidos) encontra-se a unidade procpdef constituída de rotinas de biblioteca da linguagem FIG.

Ao ser compilado e executado pelo Pascal, o programa armazenado no arquivo CAPI.PAS gera o arquivo CAPI.PS.

Arquivo CAPI.FAX: Este arquivo contém as definições dos novos comandos que são inseridos no arquivo CAPI.TEX.

```
\input{PLOT.STY}
\newcommand{\FIGa}{\insertplot{FIGA.FHP}{ 5.05cm}{0cm}}
```

O conteúdo do arquivo FIGA.FHP é resultado do processo de filtragem aplicado sobre o arquivo FIGA.HP, que por sua vez é gerado pelo interpretador PostScript quando este processa o programa contido no arquivo FIGA.PS. No arquivo PLOT.STY está armazenada a macro **\insertplot**.

Arquivo FIGA.PS: Como o usuário fez opção para traduzir o desenho para PostScript (\begin{FIG}[p]), este arquivo foi gerado contendo tal tradução. O código resultante é quase idêntico àquele exibido na seção 3.2.2. A diferença é que aqui o código corresponde à tradução de uma árvore binária completa de altura 3.

Apêndice E

Exemplos de Especificações de Figuras Feitas em FIG

Neste apêndice são apresentadas especificações de algumas figuras que foram inseridas automaticamente neste texto. São apresentadas apenas as subfiguras; os desenhos completos incluem as declarações dos corpos dos programas principais que são constituídos, simplesmente, de chamadas a tais subfiguras.

1. Especificação da Figura 2.2 – "Retângulo Envolvente":

```
Figure Retenv ;
  Begin
     r : Rectangle 5,3 At [5,5] ;
     p := r.lt ;
     For i From 1 To 3 Do
        { For j From 1 To 3 Do
             { Circle 0.1 , true At p ;
               p := p + [2.5,0] } ;
               p := [r.lt.x,p.y - 1.5] } ;
     Text "LT" At r.lt + [-0.4,0.4] ;
     Text "TC" At r.tc + [0,0.4]    ;
     Text "RT" At r.rt + [0.4,0.4]  ;
     Text "LC" At r.lc + [-0.4,0]   ;
     Text "CENTER" At r.center + [0,0.4] ;
     Text "RC" At r.rc + [0.4,0] ;
     Text "LB" At r.lb + [-0.4,-0.4] ;
     Text "BC" At r.bc + [0,-0.4]   ;
     Text "RB" At r.rb + [0.4,-0.4] ;
  End ;
```

2. Especificação da Figura 4.1 – "Níveis de tradução de um programa FIG":

```
Figure Conjunto ;
  Parameters{larg = 2,alt = 1} ;
  Results{p1,p2,p3,p4} ;
  Begin
     cd := south ;
     r1 : Rectangle larg , alt ;
     Arrow ;
     c1 : Circle alt div 2 ;
     Arrow ;
     r2 : Rectangle larg , alt ;
     Arrow ;
     c2 : Circle alt div 2 ;
     Arrow ;
     r3 : Rectangle larg , alt ;
     Text ".fig" At r1.center ;
     Text "FIG"  At c1.center ;
     Text ".pas" At r2.center ;
     Text "PASCAL" At c2.center ;
     Text ".ps" At r3.center ;
     p1 := r1.rb ;
     p2 := r2.rt ;
     p3 := r2.rb ;
     p4 := r3.rt
  End ;
Figure Niveis ;
  Begin
     vh := 1 ; vl := 2 ;
     conj : Conjunto vh,vl At [10.75,10] ;
     l := dist(conj.lt,conj.rt) + vl ;
     h := dist(conj.lb,conj.lt) - (4 * vh + 1) ;
     Rectangle l , h At conj.center ;
     Rectangle l - 0.2 , h - 0.2 At conj.center ;
     Postscriptext "[0.1 cm] 0.1  cm setdash" ;
     Line conj.p1 , conj.p1 + [3,0] , conj.p2 + [3,0] , conj.p2 ;
     Line conj.p3 , conj.p3 + [3,0] , conj.p4 + [3,0] , conj.p4 ;
     Text "N\202vel 1" At  conj.p2 + [4,dist(conj.p2,conj.p1) div 2] ;
     Text "N\202vel 2" At  conj.p4 + [4,dist(conj.p3,conj.p4) div 2] ;
     Postscriptext "[ ] 0.1 setdash" ;
  End ;
```

3. Especificação da última figura apresentada na seção 4.3:[1]

```
Figure Desenha_Niu ;  ! Desenha o apontador nil
   Results{pc} ;
   Begin
      cd := south ;
      r : Line ; Line ;
      Line [0,0] , [0.5,0] At r.last ;
      Line [0.1,0] , [0.3,0] At r.last + [0,-0.1] ;
      pc := r.first
   End ;

Figure Base ;
   Parameters{t1,t2,niu} ;
   Results{p1,p2,p3,pc} ;
   Begin
      r1 : Rectangle 2,1 ;
      r2 : Rectangle 1,1 ;
      Circle 0.05 , true  At r2.center ;
      cc := r2.center ;
      cd := south ;
      Arrow , + [0,-2] ;
      cc := + [0,-0.2] ;
      r3 : Rectangle 2,1 With .rt At cc ;
      r4 : Rectangle 1,1 With .lc At r3.rc ;
      Circle 0.05 , true At r4.center ;
      Text "Identif." At r1.tc + [0,0.3] ;
      Text "Tipo" At r2.tc + [0,0.3] ;
      Text "NomeTipo" At r3.tc + [0,0.3] ;
      Text "Elo" At r4.tc + [0,0.3] ;
      Text t1 At r1.center ;
      Text t2 At r3.center ;
      If niu
         Then { Desenha_Niu With .pc At r4.center } ;
      p1 := r4.center ;
      p2 := r4.rc ;
      p3 := r3.bc ;
      pc := r1.tc ;
   End ;

Figure Figura1 ;
   Parameters{t1,t2,t3,t4,niu} ;
   Results{p1,p2,p3,p4,pc} ;
   Begin
      r1 : base t1,t2,niu ;
```

[1] As demais figuras desta seção são obtidas através de chamadas convenientes às figuras: Figura1, Figura2 e Figura3.

```
      r2 : base t3,t4,true With .lt At r1.rt + [5,0] ;
      p1 := r1.p1 ;
      p2 := r1.p2 ;
      p3 := r2.p3 ;
      p4 := r2.p1 ;
      pc := r1.pc
   End ;

Figure Figura2 ;
   Parameters{t,niu} ;
   Results{p1,p2} ;
   Begin
      r1 : Figura1 "a","indefinido","b",t,niu ;
      r2 : Rectangle 2,1 With .tc At [r1.center.x - 0.5,r1.pc.y] ;
      r3 : Rectangle 1,1 ;
      Circle 0.05 , true At r3.center ;
      Text "Identif." At r2.tc + [0,0.3] ;
      Text "Tipo" At r3.tc + [0,0.3] ;
      Text "c" At r2.center ;
      cd := south ;
      cc := r3.center ;
      Line , [cc.x,r1.p2.y + 0.5 ] ;
      Arc 0 , -90 , 0.5 ;
      Arrow , r1.p2 + [0.2,0] ;
      p1 := r1.p1 ;
      p2 := r1.p3
   End ;

Figure Figura3 ;
   Parameters{t,niu} ;
   Begin
      r : Figura2 t , niu;
      cd :=  south ;
      cc := r.p1 ;
      Line ; Line ;
      Arc 180,270,0.5 ;
      cd := east ;
      Line cc , [r.p2.x - 0.5,cc.y] ;
      Arc 270,360,0.5 ;
      Arrow , r.p2 + [0,-0.2]
   End ;

Figure Figura4 ;
   Begin
      Figura3 "ponto",false ;
   End ;
```

4. Especificação da Figura 4.9 – "Desenho produzido incluindo o uso da primitiva Arc":

```
Figure Arco_Linha;
   Begin
      Arc 90 , 0 , 0.5 ;
      Line ;
      Arc 180 , 90 , 0.5 ;
      cd := north ;
      Line ;
      Arc 270 , 180 , 0.5 ;
      cd := west ;
      Line ;
      Arc 360 , 270 , 0.5 ;
      cd := south ;
      Line
   End ;

Figure Arco ;
   Begin
      Arrow [1,0.5] , [1,6] ;
      Arrow [0.5,1] , [7,1] ;
      Arco_Linha at [3.5,3.5] ;
   End ;
```

5. Especificação da Figura 4.10 – "Desenho produzido incluindo o uso da primitiva Text":

```
Figure Reflete ;
   Begin
      Text "Texto Refletido" At [5,5] ;
      Postscriptext "[0.1 cm] 1 setdash" ;
      Line [3,4] , [7,4] ;
      Postscriptext "[ ] 0 setdash" ;
      reflect [3,4] , [7,4] Text "Texto Refletido" At [5,3]
   End ;

Figure Texto ;
   Begin
      Arrow [1,0.5] , [1,6] ;
      Arrow [0.5,1] , [7,1] ;
      rotate 45 Reflete At [3.5,3.5] ;
   End ;
```

6. Especificação da Figura 5.1 – "Esquema de implementação do Integrador FIG-LaTeX":

```
Figure Conjunto ;
   Results{p1,p2,p1c,p2c} ;
   Begin
      cd := south ;
      r1 : Rectangle 2,1 ;
      Arrow ;
      r2 : Circle 0.75 ;
      p1 := r2.rc ;
      p2 := r2.lc ;
      p1c := r1.center ;
      p2c := r2.center ;
   End ;

Figure Corpo ;
   Results {p1,p2,p3,p4,p5,p6,p7,p8,p9,
            p10,p11,conc1,conc2,conc3,conc4} ;
   Begin
      cd := south ;
      r1 : Conjunto ;
      Arrow ;
      r2 : Conjunto ;
      Arrow ;
      r3 : Conjunto ;
      r4 : Rectangle 2,1 At r1.p2 + [-2,0] ;
      Arrow r4.rc , r1.p2 ;
      r5 : Rectangle 2,1 At r2.p2 + [-2,0] ;
      Arrow r5.rc , r2.p2 ;
      r6 : Rectangle 2,1 At r1.p1 + [2,0] ;
      Arrow r1.p1 , r6.lc ;
      r7 : Circle 0.75 At r2.p1 + [2,0] ;
      Arrow r6.bc , r7.tc ;
      r8 : Rectangle 2,1 At r7.rc + [2,0] ;
      Arrow r7.rc , r8.lc ;
      p1 := r1.p1c ;
      p2 := r1.p2c ;
      p3 := r2.p1c ;
      p4 := r2.p2c ;
      p5 := r3.p1c ;
      p6 := r3.p2c ;
      p7 := r4.center ;
      p8 := r5.center ;
      p9 := r6.center ;
      p10 := r7.center ;
      p11 := r8.center ;
      conc1 := r3.p1 ;
```

```
      conc2 := r3.bc ;
      conc3 := r8.rc ;
      conc4 := r7.bc ;
   End ;

Figure CorpoAlma ;
   Results {conc1,conc2,conc3,conc4} ;
   Begin
      r : Corpo ;
      Text ".FLT" At r.p1 ;
      Text "PrePro" At r.p2 ;
      Text ".FIG" At r.p3 ;
      Text "FIG" At r.p4 ;
      Text ".PAS" At r.p5 ;
      Text "Pascal" At r.p6 ;
      p := r.p7 ;
      For i From 1 To 2 Do
         { Text "Biblioteca" At p ;
           p := r.p8 } ;
      Text ".TEX" At r.p9 ;
      Text "LaTeX " At r.p10 ;
      Text ".DVI" At r.p11 ;
      conc1 := r.conc1 ;
      conc2 := r.conc2 ;
      conc3 := r.conc3 ;
      conc4 := r.conc4
   End ;

Figure Capsula ;
   Parameters{mens1,mens2,mens3} ;
   Begin
      r1 : Rectangle 2,0.75 ;
      r2 : Rectangle 2,0.75 At r1.center + [0,1.5] ;
      x := r1.center + [0,-0.8] ;
      For i From 1 To 3 Do
         { Circle 0.03 , true At x ;
           x := x + [0,-0.3] } ;
      r3 : Rectangle 2,0.75 At x + [0,-0.5] ;
      Text mens1 At r2.center ;
      Text mens2 At r1.center ;
      Text mens3 At r3.center ;
   End ;

Figure Completa_EsqImp;
!  Figura do esquema de implementacao adotado
   Parameters{p,q} ; ! p e q sao pontos de concatenacao desta
   Results{conc} ;   ! figura ao resto do desenho
   Begin
```

```
      cd := south ;
      cc := p ;
      Arrow ;
      Postscriptext "0.9 setgray" ;
      r1 : Rectangle 2.5 , 5 , true ;
      Postscriptext "0 setgray" ;
      r2 : Capsula "FIGA.PS","FIGB.PS","FIGZ.PS" At r1.center ;
      Rectangle 2.5 , 5 At r1.center ;
      cc :=  r1.rc ;
      cd := east ;
      Arrow ;
      r3 : Circle 0.75 ;
      Arrow ;
      Postscriptext "0.9 setgray" ;
      r4 : Rectangle 2.5 , 5 , true ;
      Postscriptext "0 setgray" ;
      r5 : Capsula "FIGA.HP","FIGB.HP","FIGZ.HP" At r4.center ;
      Rectangle 2.5 , 5 At r4.center ;
      Arrow ;
      r6 : Circle 0.75 ;
      Arrow ;
      Postscriptext "0.9 setgray" ;
      r7 : Rectangle 2.5 , 5 , true ;
      Postscriptext "0 setgray" ;
      r8 : Capsula "FIGA.FHP","FIGB.FHP","FIGZ.FHP" At r7.center ;
      Rectangle 2.5 , 5 at r7.center ;
      r9 : Circle 0.75 At [r7.center.x,q.y] ;
      Arrow r7.tc , r9.bc ;
      Arrow q , r9.lc ;
      cc := r9.rc ;
      Arrow ;
      r11 : Rectangle 2,1 ;
      Text "DVI" At  r9.center ;
      Text ".HP" At r11.center ;
      Text "Post" At r3.center + [0,0.1] ;
      Text "Script" At r3.center + [0,-0.1] ;
      Text "Filtro" At r6.center ;
      conc := p ;
   End ;

Figure EsquemaImp ;
   Begin
      r1 : CorpoAlma ;
      Completa_EsqImp r1.conc2 , r1.conc3 With .conc At r1.conc2 ;
      r2 : Rectangle 2,1 At r1.conc1 + [2,0] ;
      Arrow r1.conc1 , r2.lc ;
      Arrow r2.tc , r1.conc4 ;
      Text ".FAX" At r2.center ;
```

```
End ;
```

Obs: esta figura poderia ter sido especificada de forma mais simples, porém, como ela assemelha-se muito às figuras 3.2, 3.3 e 3.4, decidiu-se reaproveitar código, evidenciando-se as partes comuns a todas elas em subfiguras. As diferenças são tratadas em subfiguras específicas de cada desenho (neste caso, pelas subfiguras `Completa_EsqImp` e `EsquemaImp`).

Bibliografia

[1] Adobe Systems, *PostScript Language Reference Manual*, Addison-Wesley, 1986.

[2] Adobe Systems, *PostScript Language Tutorial and Cookbook*, Addison-Wesley, 1986.

[3] Adobe Systems, Glenn C. Reid, *PostScript Language Program Design*, Addison-Wesley, 1988.

[4] Apple Computer, *LaserWriter Reference Manual*, Apple Computer, 1987.

[5] ArborText, Inc., *DVILASER/HP User Manual* , ArborTex, Inc., 1987.

[6] Beck, M. *TransFig: Portable Figures for TEX*, Cornell University, Dept. of Computer Science, Technical Report, #89 - 967, 1989.

[7] Beck, M. e A. Siegel, *TransFig: Portable Graphics for TEX*, *TUGboat* The Communications of the TEX Users Group, vol. 11, no. 3 – Preceedings of the 1990 Annual Meeting, 1990.

[8] Berghauser, T.W. e P.L. Schlieve, *Illustrated AutoCAD*, Wordware Publishing, Inc., 1989.

[9] Borland, Inc., *Turbo Pascal Owner's Handbook*, Borland Inc., 1987.

[10] Brodie, L., *Starting Forth: an introduction to the Forth language and operating systems for beginners and professionals*, Englewood Cliffs, Prentice-Hall, 1981.

[11] Foley, J.D. e A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

[12] Hewlett Packard, *LaserJet series II Printer Technical Reference Manual*, Hewlett-Packard Company, 1987.

[13] Holzgang, D. A., *Descobrindo a Linguagem PostScript*, Livros Técnicos e Científicos, 1990.

[14] Kernighan, B. W., *PIC - A Language for Typesetting Graphics*, Software - Practice and Experience, vol. 12, no. 1, pp. 1-21, 1982.

[15] Kernighan, B. W., *A Graphics Language for Typesetting. Revised User Manual*, Computing Science Technical Report no.116, AT&T Bell Laboratories, 1984.

[16] Knuth, D. E., *The TEXbook*, Addison-Wesley, 1984.

[17] Kowaltowski, T. e J. C. Setubal, *FIG: Uma Linguagem para Especificação de Figuras*, ANAIS do VIII Congresso da Sociedade Brasileira de Computação - XV Semish, 1988.

[18] Kowaltowski, T., *Implementação de Linguagens de Programação*, Guanabara Dois, 1983.

[19] Lamport, L., *LATEX: A Document Preparation System*, Addison-Wesley, 1986.

[20] Newman, W. M., *Display Procedures* Communications of the ACM, vol.14, no.10, pp. 651-660, 1971.

[21] Ossanna, J. F., *NROFF/TROFF User's Manual*, Computing Science Technical Report no.54, AT&T Bell Laboratories, 1979.

[22] Setubal, J. C., *FIG: Uma Linguagem para Especificação de Figuras* (tese de mestrado), DCC – Unicamp, 1987.

[23] Silva, H. V. R. C., *Recuperação de Erros em Analisadores Sintáticos Descendentes* (tese de mestrado), DCC – Unicamp, 1981.

[24] Van Wyk, C. J., *A Language for Typesetting Graphics*, Standford Department of Computer Science, Report no.STAN-CS-80-803, 1980.

[25] Van Wyk, C. J., *A High-level Languages for Specifying Pictures*, ACM Transactions on Graphics, vol.1, no.2, pp. 163-182, 1982.

[26] Wichura, M., *"The PiCTeX Manual". Software Documentation*, The University of Chicago (November, 1986). Second printing, by The TeX Users Group, as *TeXniques*, Number 6, 1987.

[27] Wirth, N., *Algorithms + Data Structures = Programs*, Englewood Cliffs, Prentice-Hall, Inc., 1976.