FIG
Uma Linguagem para Especificação de Figuras

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. **João Carlos Setubal** e aprovada pela Comissão Julgadora.

Campinas, 21 de agosto de 1987

Prof. Dr. _____
(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**.

A Teca e Claudia,
companheiras de aventura.

Agradecimentos

Ao prof. dr. Tomasz Kowaltowski, pela inestimável orientação e constante estímulo;

ao prof. dr. Cláudio L. Lucchesi, pelo decisivo impulso inicial, sem o qual a esta altura este trabalho talvez nem tivesse sido começado;

ao prof. dr. Hans K. E. Liesenberg, pelo auxílio junto aos trâmites da pós-graduação;

ao prof. dr. Léo Pini Magalhães, por uma conversa esclarecedora;

ao Fernando P. Barreiros, pela valiosa leitura crítica de uma primeira versão;

ao Cau e ao Silvio, pelo fornecimento de material bibliográfico;

ao meu tio Antonio Rezende, pelo estímulo e apoio logístico;

e aos colegas deste Departamento de Ciência da Computação, cujo apoio foi crucial para o bom termo desta dissertação.

Sumário

FIG é uma linguagem de programação cujo objetivo é possibilitar a especificação de figuras bidimensionais de forma não-interativa. Embora a linguagem tenha sido projetada para uso geral, ela poderia ser usada particularmente em conjunto com um programa de composição gráfica de modo a permitir inserção automática de figuras em textos. A característica principal de FIG é o recurso de definição e uso de *figuras* de maneira análoga a procedimentos, modelando assim o desenho sem utilizar estruturas 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 chamada ou 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 à instanciação consecutiva de diferentes *figuras*; por posicionamento relativo a outras *figuras* já instanciadas; e por coordenadas absolutas. FIG tem um conjunto de primitivas gráficas, cada qual com seus atributos. O projeto de FIG fundamentou-se na linguagem PIC de Kernighan, nos procedimentos de display de Newman e em certos aspectos da norma GKS. Foi construído um compilador experimental que produz Pascal como linguagem objeto. A saída gráfica é realizada através de um pacote gráfico conveniente, que pode ser facilmente trocado.

Abstract

FIG is a programming language for non-interactive specification of bidimensional pictures. Although the language was designed for general purpose picture drawing, it is aimed at integration with typesetting programs so as to allow automatic insertion of pictures into text. Its main features are the definition and use of *figures* in the same way as procedures, thus modeling pictures in a data-structure-free manner. FIG makes it easy to build drawings with a large hierarchical structure. *Figures* have a flexible parameterization mechanism and return values in a special way. *Figure* call (instantiation) can be modified by built-in geometric transformations. There are three ways to place a *figure* in the plane: implicit movement associated with the consecutive instantiation of *figures*; placement relative to already instantiated *figures*; and absolute placement. FIG has a set of primitive *figures*, each with its own set of attributes. FIG's design was based primarily on Kernighan's PIC language, on Newman's display procedures and on certain features of the GKS international standard. An experimental compiler producing machine-independent Pascal object code was implemented. Graphical output is achieved through routines of a convenient graphics package.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Tem alcançado bastante impulso nestes últimos anos o uso de programas de computador para composição gráfica de textos. Dois dos mais conhecidos programas desse tipo são TROFF [28], existente em ambientes UNIX[1], e TEX[2], de Knuth [14], este independente de ambientes, e com o qual, aliás, este próprio texto foi composto.

Uma das motivações principais que conduziu à feitura desses programas foi a de dar ao autor do texto controle completo sobre a produção da sua obra. Como se sabe, a existência de editores de texto já há certo tempo vem facilitando o trabalho datilográfico, mas a composição gráfica ainda era tarefa dos tipógrafos. Com o advento das impressoras a laser e compositoras digitais, e posteriormente dos programas acima citados, os autores passaram também a ser tipógrafos de seu próprio trabalho. Porém, para que esses programas sejam realmente versáteis, é preciso que o autor tenha condições de compor inclusive partes de seu trabalho que não são propriamente *texto*, tais como equações matemáticas e figuras, o que, em princípio, é totalmente factível com as novas impressoras e compositoras. E, de fato, tanto TROFF quanto TEX permitem a composição de quase qualquer tipo de equação matemática. Já quanto a figuras, a situação é um pouco diversa, pois elas são uma forma de comunicação essencialmente diferente de texto, requerendo um tratamento especial.

A integração de figuras em programas compositores de texto depende muito da forma de entrada do texto utilizada pelo programa. Nos sistemas acima citados, essa forma é não-interativa: neles, o usuário deve preparar o texto que quer compor utilizando um editor de textos comum[3] num terminal alfanumérico, in-

[1] UNIX é uma marca registrada da AT&T

[2] TEX é uma marca registrada da American Mathematical Society

[3] mas que não insira automaticamente caracteres de controle não convencionais.

cluindo no seu texto, em locais apropriados, seqüências especiais de caracteres. Essas seqüências, que são específicas do programa que está sendo utilizado, são comandos que indicam como o texto deve ser composto, tanto nos detalhes como no seu todo. A seguir, o arquivo assim preparado é submetido a um processamento em uma ou mais etapas que produz um outro arquivo. Finalmente, aciona-se um segundo programa que, tomando este último arquivo como entrada, transforma o seu conteúdo em instruções próprias ao dispositivo de saída (a impressora ou a compositora) produzindo o texto composto. Para incluir desenhos ao longo do trabalho uma possibilidade é permitir que figuras sejam especificadas também por seqüências especiais de caracteres, tal como no restante do sistema. Porém, se a variedade de figuras que se deseja é grande e se a flexibilidade em especificá-las é razoável, essas seqüências acabam por se tornar uma linguagem de programação própria.

Uma outra forma de funcionamento dos programas compositores de texto é a interativa: o usuário dispõe de uma estação de trabalho com capacidade gráfica relativamente sofisticada (pelo menos 500 x 500 pixels) e, acionando o programa compositor, digita seu texto, que vai aparecendo na tela praticamente com a mesma forma que aparecerá impresso em papel. É a chamada abordagem WYSIWYG ("what you see is what you get"). Nestes sistemas, a inserção de figuras pode ser feita por meio de um programa gráfico interativo auxiliar, fazendo uso por exemplo dos modernos ambientes integrados, com múltiplas janelas, disponíveis em estações de trabalho mais recentes.

Por enquanto estes dois tipos de abordagem têm sido concorrentes, com partidários de um e de outro. Cremos, porém, que a abordagem não-interativa apresenta uma série de vantagens no momento que explicam a sua predominância entre a comunidade de usuários. A principal delas talvez seja a utilização de equipamento simples e de grande disponibilidade para a entrada do texto (os terminais alfanuméricos), o que é mais verdade ainda no caso do Brasil, onde as estações de trabalho sofisticadas vão custar ainda a se difundir. Outra vantagem é a própria existência dos programas TEX e TROFF, ambos com recursos bastante poderosos e com uma disseminação muito grande nos Estados Unidos e em outros países. Finalmente, outra vantagem é a facilidade e o estímulo à estruturação dos textos ("logical design", segundo [17]) que uma tal abordagem oferece; a numeração dos capítulos, seções e parágrafos, das equações matemáticas e das citações bibliográficas, por exemplo, fica por conta do programa.

Tomando então como premissa que vale a pena um esforço no sentido de tornar ainda mais versáteis compositores de texto não-interativos, coloca-se o problema da integração de um mecanismo que permita especificação de figuras com esses programas. Conforme mencionado acima, um possível mecanismo é

uma linguagem de programação voltada para figuras. Esta abordagem parece ser a melhor pois se integra mais harmoniosamente com o restante do sistema. Além disso, as linguagens de programação, embora mais difíceis de usar do que os sistemas gráficos interativos, têm uma ou outra vantagem em relação a estes últimos, tal como facilidades na especificação de figuras muito estruturadas (por exemplo, uma estrela de 15 pontas) ou na parametrização de figuras que se repetem com variações.[4] E, de fato, já foram projetadas linguagens de programação para figuras nos sistemas compositores de texto citados anteriormente. No caso de TROFF existem as linguagens PIC [10] e IDEAL [36], ambas para figuras no plano. No caso de TEX parece que existem diversos trabalhos, dentre os quais, por exemplo, a capacidade de figuras do sistema LATEX [17].

Porém, esses trabalhos, no nosso entender, apresentam algumas limitações: PIC, por exemplo, embora relativamente fácil de usar, não oferece todos os recursos normalmente disponíveis em linguagens de programação, como procedimentos; IDEAL, por outro lado, embora bastante flexível e abrangente, exige do usuário conhecimentos de sistemas de equações e de aplicação de números complexos em geometria. Já LATEX é muito limitado quanto ao tipo de figuras que se consegue especificar. Parece haver, portanto, espaço para uma linguagem que não seja tão sofisticada quanto IDEAL, mas ao mesmo tempo apresente mais recursos do que PIC. O objetivo deste trabalho é justamente o de apresentar uma solução intermediária para uma tal linguagem, mantendo a aplicação para figuras bidimensionais.

Cumpre entretanto notar que a preocupação maior ao longo do trabalho foi de definir as características dessa linguagem através de estudo de trabalhos anteriores e realizar uma implementação experimental, sem levar em conta quaisquer sistemas compositores de texto. A integração com um tal sistema, seja ele qual for, fica como uma possível continuidade deste trabalho.

Este texto encontra-se dividido da seguinte forma: no Capítulo 2 apresenta-se uma revisão de trabalhos existentes na literatura sobre linguagens de programação para figuras, salientando aqueles que serviram de subsídio para a elaboração da linguagem contida nesta dissertação. No Capítulo 3 apresenta-se a linguagem propriamente dita. No Capítulo 4 são mostrados aspectos da implementação experimental realizada. Finalmente no Capítulo 5 são apresentadas as sugestões de aperfeiçoamentos e de prosseguimento do trabalho.

[4] Os sistemas gráficos interativos, por outro lado, são muito mais vantajosos em outros ambientes, como por exemplo em CAD ("computer-aided design").

Capítulo 2

Revisão de trabalhos anteriores

A área de linguagens para especificação de figuras já é bastante tradicional na computação, havendo trabalhos desde a década de 60. Como resultado, existe na literatura uma quantidade bastante grande de descrições das chamadas *linguagens gráficas*, cada qual apresentando sua solução para o problema. A intenção deste capítulo não é de fazer uma revisão exaustiva desses trabalhos, mas apenas de apresentar um panorama geral do assunto e descrever certas linguagens específicas às quais tivemos acesso e que tiveram maior influência no projeto da linguagem que é tema desta dissertação.

2.1 Breve histórico

A computação gráfica, como se sabe, teve, nas pesquisas de Ivan Sutherland [32] no início da década de 60, o seu grande trabalho acadêmico pioneiro. Tratava-se de um sistema interativo, o que era uma enorme inovação para a época, e serviu de modelo para muitos trabalhos posteriores.

Após a disseminação inicial das linguagens de alto nível surgiram inúmeros esforços objetivando dotar as linguagens existentes de recursos gráficos. Assim é que apareceram versões de Fortran, PL/1, Algol e outras que incorporavam bibliotecas de subrotinas para produção de figuras. Também apareceram linguagens especiais que se propunham tratar de figuras em aplicações específicas, como desenho animado.

O primeiro trabalho significativo em linguagem especial para figuras foi o de Kulsrud [16], em 1968. Já nesse trabalho alguns dos pontos fundamentais do

problema foram salientados, tais como a necessidade de procedimentos e de um esquema que permitisse tratar figuras com estruturas hierárquicas. A linguagem proposta incluía comandos para desenhar figuras simples tais como pontos, linhas e arcos, comandos rudimentares para manipular as figuras, comandos para controle do programa, e até mesmo comandos para tratamento de regiões. Era, no entanto, uma linguagem sintaticamente muito primitiva, aproximando-se de uma linguagem de montagem. Ressalte-se ainda a preocupação em criar a linguagem através de um "metacompilador", análogo aos compiladores cruzados atuais.

A relevância do tópico no correr dos anos tornou-se grande e motivou a realização de um simpósio conjunto entre os grupos SIGPLAN e SIGGRAPH da ACM tendo como tema exclusivo as linguagens gráficas, em 1976 [9]. Dentre os diversos trabalhos apresentados, um dos mais citados passou a ser o de Schrack [29]. Nele aparece o conceito de *tipo de dado gráfico*, realizado através de uma extensão de Fortran para ser usada em CAD.

Desde então, o tema de linguagens gráficas parece ter sido relegado a um segundo plano, pois não tivemos conhecimento de realização de um novo simpósio. Se isto for verdade, poderíamos supor que a disseminação dos sistemas gráficos interativos a partir de meados da década de 70, graças ao barateamento do hardware, foi a responsável por essa queda de interesse. Por outro lado, houve um esforço internacional muito grande com o objetivo de padronizar a terminologia e as interfaces da computação gráfica, o que resultou na norma GKS [8]. Nesta norma, entre outras coisas, foi padronizada a forma de chamada de subrotinas gráficas em linguagens de programação.

Mesmo assim, trabalhos relativos a linguagens gráficas continuaram e continuam a ser produzidos. É importante mencionar a linguagem LOGO que, embora trazendo embutidas rotinas gráficas bastante simples, trouxe como grande contribuição uma nova forma de construção geométrica, a "geometria da tartaruga" [1]. Com o advento dos sistemas compositores de texto, desenvolveram-se, por volta de 1981, as linguagens PIC e IDEAL, já citadas, trazendo inovações para a área e que serão detalhadas adiante.

Mais recentemente surgiram linguagens que fogem aos padrões tradicionais, sendo linguagens gráficas propriamente ditas, no sentido de que a programação em si não é mais unidimensional, mas bidimensional. É o caso da linguagem Mandala [18], mas que se propõe a ser uma linguagem de propósito geral, e não apenas aplicada a figuras.

Tudo indica portanto que é uma área de pesquisa onde há muito por inovar.

2.2 A linguagem PIC

Um dos objetivos deste trabalho é, conforme mencionado acima, o de explorar possíveis variações das linguagens para figuras criadas em ambientes de composição de texto. A base a partir da qual o trabalho foi feito é a linguagem PIC idealizada por B. W. Kernighan e apresentada nesta seção. Tomamo-la como modelo tendo em vista o ambiente na qual foi projetada, que lhe conferiu um alto grau de praticidade e simplicidade, como se verá. O resumo de PIC que se segue está baseado nas publicações [10] e [11].

PIC é antes de mais nada uma ferramenta a mais no conjunto fornecido pelo sistema operacional UNIX para preparação de documentos ("a bancada do escritor" [12]). Sendo assim, um dos seus objetivos de projeto foi a integração perfeita com os demais programas da família, particularmente com TROFF, que é o compositor de textos propriamente dito.

PIC funciona como um pré-processador para TROFF. O usuário digita seu texto entremeado com os comandos TROFF; quando deseja inserir uma figura, inclui a diretiva .PS ("picture start"), em seguida os comandos PIC, e finaliza com .PE ("picture end"), prosseguindo com o restante do texto. Dentro do programa PIC o usuário pode chamar comandos de TROFF, por conseguinte tendo acesso completo a todos os recursos de composição de textos dentro da definição de um desenho.

Além da integração com TROFF, outro objetivo explícito do projeto de PIC foi o de desenvolver uma linguagem fácil de se usar e aprender, inclusive por pessoas não acostumadas com programação, como pessoal administrativo.

Apesar disso, PIC não deixa de ser uma linguagem de programação do estilo tradicional, com variáveis, expressões, comandos de controle como decisão e repetição, e funções de biblioteca como seno, raiz quadrada e logaritmo. Algumas de suas peculiaridades neste aspecto são: não exige declaração de variáveis; a separação de comandos se faz por mudança de linha como em Fortran, ou por ponto-e-vírgula na mesma linha; e, embora permita expressões booleanas, não tem variáveis desse tipo (são todas reais).

Os desenhos são construídos a partir de um conjunto de *primitivas gráficas*: retângulo, linha (segmento de reta), seta, círculo, elipse, arco de círculo e "B-splines". Para incluir qualquer dessas primitivas no desenho basta invocar o comando correspondente. Os mecanismos de controle do tamanho, posição e atributos das primitivas invocadas são descritos a seguir e formam o cerne da linguagem.

O *tamanho*: cada primitiva tem parâmetros específicos que determinam sua geometria; por exemplo, o círculo tem o raio, o retângulo a altura e largura, etc. Porém, esses parâmetros podem ser omitidos no momento de chamada de

uma primitiva. Nesse caso, o sistema toma valores padrão para os parâmetros omitidos. Esses valores padrão estão em variáveis globais pré-definidas, e estas podem ser modificadas a qualquer instante.

Os *atributos*: PIC permite os atributos *estilo* para linhas e "splines" (contínua, pontilhada e tracejada) e *invisibilidade* para todas as primitivas. Este último permite invocar uma primitiva sem que haja saída; a primitiva assim chamada pode entretanto ser referenciada posteriormente (a utilidade disto ficará mais clara adiante).

A *posição*: este é o aspecto mais problemático de uma linguagem unidimensional que se propõe a modelar um mundo bidimensional (problema comum a todas as linguagens gráficas evidentemente). Embora os desenhos em PIC tenham como base o plano cartesiano, procura-se evitar ao máximo o uso de coordenadas absolutas para posicionamento das primitivas (embora isto também seja possível). Assim é que um dos mecanismos fundamentais é o *movimento implícito* associado ao posicionamento das figuras. Isto faz com que a invocação de várias primitivas em sucessão coloque-as uma após a outra, de acordo com a *direção corrente*. As primitivas se juntam através de pontos bem determinados, e que variam de primitiva para primitiva. Esse mecanismo supõe também a existência de uma *posição corrente*, que é o local a partir do qual uma primitiva será desenhada, e que também fica bem determinado após a invocação de qualquer primitiva. O usuário dispõe, como é natural supor, de comandos para alterar a direção e a posição correntes; as direções permitidas são no entanto somente direita, esquerda, abaixo e acima.

Ainda outro recurso para evitar o uso de coordenadas absolutas e ao mesmo tempo permitir uma ligação lógica entre os objetos é o do *posicionamento relativo*: as primitivas podem ser posicionadas umas em relação às outras, fazendo uso de pontos especiais que resultam da invocação de cada primitiva. Esses pontos referem-se ao *retângulo envolvente* que define a primitiva invocada: trata-se do menor retângulo com lados paralelos aos eixos coordenados que contém completamente a figura.[1] Os pontos são os cantos superiores esquerdo e direito, cantos inferiores esquerdo e direito, centro do lado esquerdo, do lado direito, do lado superior e do lado inferior, e o centro propriamente dito. Um problema que surge é o da referência a esses pontos. Isto é resolvido em PIC por um mecanismo de *rotulamento*. Cada primitiva invocada recebe um rótulo, implícito ou explícito, e seus pontos especiais podem ser referenciados com a mesma sintaxe dos campos de registros de Pascal. Por exemplo, `A.nw` é o canto superior esquerdo do objeto rotulado como `A`. O rótulo implícito faz uso da ordem com que as primitivas são

[1] Conhecido por "bounding box" [30] ou "extents" [7] na literatura de língua inglesa.

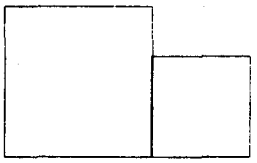

Figura 2.1: Desenho resultante da execução dos comandos de PIC exemplificados no texto.

chamadas ao longo do programa; por exemplo, `last box.sw` ou `2nd circle.c`, onde `box` e `circle` são nomes de primitivas. Certas primitivas ainda possuem pontos específicos, como as linhas (ponto inicial e ponto final). O encadeamento dos objetos entre si se faz com a palavra-chave `with`, combinada com `at`, conforme mostra o exemplo a seguir, retirado de [11]:

```
box ht 0.75 wid 0.75
box ht 0.5 wid 0.5 with .sw at last box.se
```

O resultado da execução desses comandos está ilustrado na figura 2.1. A palavra reservada `at` pode também ser usada para posicionamento absoluto normal.

O atributo *invisibilidade* mencionado acima pode ser usado justamente para referência aos pontos de uma primitiva invocada sem que ela seja na realidade desenhada.

Cadeias de caracteres em PIC têm dupla personalidade: podem ser atributos de outras primitivas ou uma primitiva por si só, o *texto*. Quando funcionando como atributo, basta suceder uma invocação de primitiva com uma cadeia de caracteres entre aspas, e essa cadeia será posicionada sobre a primitiva de acordo com o alinhamento ("justification") ativo. Dentro das aspas pode aparecer qualquer seqüência de caracteres válida em TROFF, e mesmo válida em EQN, o pré-processador de equações matemáticas. Assim sendo, o tamanho do texto é controlado por TROFF e não por PIC. Quando usado como primitiva, o texto pode usar todos os recursos de posicionamento citados acima.

As primitivas podem ainda ser agrupadas em *blocos*. Todo bloco também define um retângulo envolvente, e também pode ser rotulado. Além disso, também é possível fazer referência às primitivas que estão agrupadas em seu interior.

PIC dispõe também de um mecanismo de macros, permitindo assim definir figuras parametrizadas. Não é um mecanismo de procedimentos verdadeiro, embora, juntamente com os blocos, "simule" subrotinas, no dizer de seu autor.

Existem alguns recursos para facilitar a leitura de arquivos contendo dados ou comandos de PIC. Este último recurso possibilita a preparação de programas PIC fora do contexto TROFF e posterior inclusão.

Ressalte-se finalmente que o fato da especificação das figuras se fazer por meio de um texto ASCII permitiu que se pudessem criar pré-processadores especializados para PIC. Um deles é o programa GRAP [3], que é semelhante a PIC mas voltado para a confecção de gráficos cartesianos, histogramas, etc.

Em conclusão, pode-se dizer que PIC é uma linguagem simples mas poderosa, cumprindo plenamente os objetivos a que se propôs. No entanto, ainda há lugar para aperfeiçoamentos. O próprio Kernighan reconheceu em [10] que, embora conte com uma simulação de subrotinas, "procedimentos verdadeiros provavelmente ainda são necessários".

E por quê? Ora, procedimentos são um recurso mais poderoso e versátil do que simples macros. Com procedimentos torna-se mais fácil, como se verá, a construção de desenhos que apresentam uma natureza hierárquica, onde existem figuras que entram na composição de outras figuras, que por sua vez podem também entrar na composição de outras figuras, e assim por diante. É evidente que a presença de um tal mecanismo representa um salto na complexidade tanto da definição quanto da implementação da linguagem, sendo assim compreensível que ele ainda não tenha sido incluído em PIC. (Talvez este aperfeiçoamento esteja em curso no momento; a referência mais recente de que dispusemos [11] é de dezembro de 1984.)

Nas seções seguintes são apresentados outros trabalhos da literatura em que procuramos soluções para o problema dos procedimentos acima citado, tentando também colher idéias para outros tipos de aperfeiçoamentos em PIC.

2.3 Procedimentos de Display

Em 1971, W. M. Newman publicou um artigo apresentando o conceito de procedimentos de display ("display procedures") [24]. A motivação para esse trabalho estava em certas dificuldades dos sistemas gráficos da época, baseados nos "display files" e monitores vetoriais. Os "display files" eram conjuntos de instruções indicando como deveria se movimentar o canhão do monitor para reproduzir uma determinada figura. Uma das dificuldades era a excessiva dependência que essas instruções tinham dos dispositivos de saída a que estavam ligadas; outra era a duplicação das instruções que ocorria quando o monitor em uso não era capaz de fazer uma mudança de escala no desenho.

Newman então propôs o uso de linguagens de alto nível para fazer a modelagem de desenhos. O mecanismo básico dessa modelagem é justamente o procedimento de display. Nesta proposta, admite-se que os desenhos têm uma estrutura hierárquica: podem ser decompostos em partes e sub-partes e assim por diante.

Cada parte ou subparte é um *símbolo* e estes são descritos por procedimentos com um nome que caracteriza o símbolo. A chamada de um tal procedimento provoca a execução das instruções que definem o símbolo e sua conseqüente exibição, correspondendo portanto a uma *instanciação* do símbolo no contexto da rotina chamadora. Os procedimentos podem naturalmente chamar outros procedimentos (símbolos compostos de sub-símbolos), mas também certos procedimentos primitivos. Portanto a árvore de chamadas da execução do programa reproduz a hierarquia da figura como um todo.

Cada símbolo é definido num sistema próprio de coordenadas, o "master coordinate system". Para que os símbolos possam ser realmente úteis é preciso então que haja um mecanismo que permita a sua transformação (em termos de tamanho, orientação e posição) quando de sua chamada para um sistema de coordenadas hierarquicamente superior. Ou seja, se houver um símbolo que representa por exemplo um furo circular, ele pode ser usado para representar furos circulares de qualquer diâmetro, desde que convenientemente transformado. Newman propôs que cada procedimento de display pudesse ser chamado com uma sintaxe especial (que significa também uma alteração semântica) onde seriam especificadas essas transformações na forma de qualificativos do procedimento, e é nisto que eles diferem dos procedimentos comuns. Por exemplo, um determinado símbolo de nome `resistor` poderia ser instanciado assim (segundo exemplo em [24], em que a idéia foi aplicada na linguagem Euler):

```
resistor at [100,100] rot pi/2
```

Os símbolos `at` e `rot` conotam translação e rotação, respectivamente.

Imaginando-se a árvore que descreve a figura, os nós representam procedimentos, ou símbolos, enquanto as arestas representam as transformações que se aplicam a cada nó. Porém, para que as diversas instanciações ocorram corretamente ao longo da árvore é preciso que haja uma concatenação das transformações nos diversos níveis. Assim, por exemplo, um símbolo do nível três tem que aplicar todas as transformações vindas dos níveis um e dois, e qualquer implementação dos procedimentos de display tem que providenciar um mecanismo que realize essa concatenação de forma sistemática.

Com estas idéias, Newman mostrou que o "display file" podia ser eliminado, e a figura assim especificada tornava-se independente dos dispositivos de saída. Ainda mais interessante era a possibilidade de se passar a usar todos os recursos das linguagens de programação na descrição do desenho, o principal dos quais, segundo Newman, seria a execução condicional dos procedimentos. Isto é útil quando se deseja, por exemplo, ter representações alternativas para um mesmo símbolo.

O artigo de Newman se tornou um clássico, e seu método passou a ser considerado como uma alternativa importante em relação aos programas baseados em estruturas gráficas de dados, uma vez que é "data-structure-free". O grande problema do método, contudo, é justamente esse: todos os símbolos têm que ser compilados antes da execução do programa; modificações dinâmicas tornam-se inviáveis, a menos que o sistema todo seja interpretado.. Como hoje em dia a ênfase em modificações interativas é muito grande, os procedimentos de display encontram uso restrito; segundo Foley e van Dam [7], poderiam vir a se tornar uma opção viável quando se desenvolver hardware especial adequado à interpretação em alta velocidade de uma linguagem que os implemente.

Cabe mencionar aqui que o próprio Newman desenvolveu uma extensão de LOGO empregando os seus procedimentos de display [25], mas essa iniciativa não teve maiores repercussões.

2.4 A linguagem MIRA

Um trabalho em linguagens gráficas desvinculado de sistemas compositores de texto é o apresentado por Magnenat-Thalmann e Thalmann [20].

Seu objetivo foi o de desenvolver um extensão gráfica para a linguagem Pascal, batizada de MIRA, que não se restringisse às habituais chamadas de subrotinas. Propuseram então a possibilidade de declaração de tipos especiais: os *tipos figuras*, semelhantes aos registros de Pascal, mas com diferenças marcantes: dentro de um tipo figura pode haver parâmetros, variáveis locais e instruções, como nos monitores de Brinch Hansen [4]. Na verdade, trata-se de uma generalização da idéia de Schrack [29].

Um *tipo figura* tem certa semelhança com procedimentos que descrevem símbolos. Porém, ao invés da saída gráfica do símbolo se realizar por meio de chamada do procedimento, em MIRA é possível apenas declarar variáveis como sendo de um *tipo figura* definido anteriormente. Para realizar a saída é preciso acionar o comando `create` ⟨variável⟩, que cria a figura na memória e depois o comando `draw` ⟨variável⟩, que efetivamente desenha a figura.

O único *tipo figura* primitivo é o `vector` (segmento de reta), e diversos `vectors` podem ser ligados pelo operador especial `connect`. Além disso, apresenta procedimentos pré-definidos para realizar transformações em variáveis de *tipo figura*, tais como simetria, translação e rotação. A existência de uma representação da figura em memória permite ainda a operação de união. Hierarquias de símbolos podem ser montadas com o comando `include`.

O posicionamento das figuras, embora possa ser parametrizado, exige uso de

coordenadas absolutas. Não há menção em [20] sobre eventuais atributos das figuras. Outra restrição é a impossibilidade de referência às variáveis locais de um *tipo figura*.

Trata-se portanto de uma linguagem em que se procura modelar as figuras tomando como base a *variável*, e não os procedimentos, embora o resultado seja muito parecido. Como linguagem para ser usada diretamente por um público amplo não nos parece conveniente, pois a variável como representação de uma figura é um conceito bem menos intuitivo do que um procedimento. Para desenvolvimento de aplicações (sistemas para desenho de fluxogramas, circuitos lógicos, etc) parece ter seu mérito, embora fuja da padronização gráfica sugerida pela norma GKS. Seus autores continuaram o trabalho tendo apresentado uma versão de MIRA para figuras tridimensionais [21].

2.5 A linguagem IDEAL

A linguagem IDEAL[2] foi desenvolvida por Van Wyk [36] para atuar exatamente no mesmo ambiente que PIC, ou seja, pré-processamento de TROFF. Porém, a linguagem resultou radicalmente diferente, como se verá a seguir. O resumo abaixo também contém informações retiradas de [35].

Em IDEAL, as figuras são especificadas tomando por base o plano complexo. Assim, variáveis (coordenadas ou números simples) são expressas como o par (a, b) referente ao complexo $a + bi$. A razão dessa escolha é que números complexos são extremamente úteis para resolver problemas de geometria, inclusive transformações geométricas ([5], [38]); por exemplo, rotações podem ser expressas por multiplicação e divisão de números complexos. Com isso, em IDEAL não há explicitação de nenhuma transformação geométrica; estas se efetivam através da especificação conveniente de "coordenadas complexas".

As figuras propriamente aparecem sob a forma de procedimentos que Van Wyk chama de *boxes*. Esses procedimentos têm variáveis locais e instruções, porém sua característica principal e peculiar é que incluem sistemas de equações relacionando os diversos parâmetros (que são variáveis locais) que descrevem a figura. Não é portanto uma linguagem de programação tradicional; seu autor a classifica como "declarative constraint language", isto é, "o usuário descreve quais relações devem existir entre as variáveis, ao invés da seqüência de comandos de atribuição que realizariam estas relações" [36]. Assim, por exemplo, o *box* retângulo ficaria assim definido:

[2] MIRA, nome da linguagem vista na seção anterior, é **maravilha** em latim; não faltaria um pouco de modéstia a esses autores?

```
retangulo {
        var ne, nw, sw, se, c ,ht, wd;
        ne = se + (0,1)*ht;
        nw = sw + (0,1)*ht;
        ne = nw + wd;
        c = (ne+sw)/2;
        conn ne to nw to sw to se to ne;
        }
```

A primitiva gráfica que aí aparece é conn, que, à semelhança de MIRA, cria segmentos de reta encadeados. Juntamente com "B-splines", são as únicas primitivas da linguagem, embora haja possibilidade de usar *boxes* de biblioteca.

Uma vez especificado, o *box* retângulo pode ser instanciado através do comando put, onde se especificam equações adicionais àquelas que estão contidas no *box*. IDEAL se encarrega de resolver o sistema, encontrando uma solução, se existir, e realizando a saída da figura. Não há problema se houver equações redundantes, mas há erro se elas forem insuficientes. A idéia de parametrizar figuras através de sistemas de equações permite a instanciação com conjuntos diferentes de parâmetros. No exemplo acima, pode-se instanciar o retângulo fornecendo-se um canto, a altura e a largura; ou então três cantos; ou então dois cantos e um dos comprimentos; e assim por diante. Apesar dessa característica ser extremamente conveniente, evidentemente o programador passa a ter que saber como expressar suas figuras por meio de equações simultâneas. Este método foi inspirado no trabalho de Knuth em seu sistema Metafont [13] onde também existem subrotinas com estas características; Knuth, aliás, foi o orientador da tese de Van Wyk que resultou em IDEAL.

Tal como em MIRA, é necessário especificar coordenadas absolutas para posicionar as figuras. Tal como em PIC, instâncias de *boxes* podem ser rotuladas, e isto permite acesso às suas variáveis locais com a mesma sintaxe dos campos de registros em Pascal. Desenhos hierárquicos são facilmente obtidos, uma vez que permitem-se comandos put dentro de outros put.

Não existem mecanismos para controle dos atributos das figuras. Por outro lado são implementados dois outros poderosos recursos. Um deles é a *pen*: trata-se da possibilidade de obter uma carreira de instâncias de um *box* qualquer ao longo de um segmento de reta, bastando especificar o início, o fim, o número de instâncias e o nome do *box* desejado. Este recurso é bastante adequado para traçar desenhos repetitivos ou mesmo estampas. Outro recurso é o de controlar a interferência que figuras superpostas causam entre si. Por um lado pode-se especificar uma figura como *opaca*, isto é, ela cobre as partes das figuras que

coincidem com ela. Isto é análogo ao atributo prioridade encontrado no GKS. Por outro lado, pode-se especificar uma figura como opaca, mas cobrindo tudo que está fora dela (`opaque exterior`). Estes efeitos podem ser conseguidos com tratamento adequado das conhecidas operações de recorte ("clipping").

Como os efeitos de `opaque exterior` são drásticos, existem os comandos `construct` e `draw`. `construct` produz uma saída "virtual", ou seja, considerando um plano auxiliar de desenho, e `draw` junta o plano auxiliar com o plano base.

Em suma, IDEAL é uma linguagem que apresenta idéias novas e bastante poderosas. Com essas ferramentas pode-se especificar um conjunto de figuras bastante diversificado em programas relativamente curtos. O preço, no entanto, é uma certa complexidade nos conceitos envolvidos, que exigem bem mais do programador do que PIC, por exemplo.

2.6 Outras propostas

As seções anteriores mostraram que é possível formular linguagens bastante distintas com praticamente o mesmo objetivo. Nesta seção discutem-se alguns outros trabalhos aos quais tivemos acesso.

Um esforço no sentido de unificar e formalizar os diversos conceitos das linguagens gráficas foi feito por Mallgren [22]. Nesse artigo o autor propõe uma linguagem de programação para especificação de figuras baseada em tipos abstratos de dados, gráficos. Mallgren apresenta cinco tipos de dados básicos: o *ponto*, a *região*, a *função geométrica*, a *transformação gráfica* e a *figura estruturada em árvore*. Para cada um deles utiliza-se a técnica formal de especificação por equações algébricas para se definir um conjunto de operações, seus argumentos e resultados. O objetivo é obter uma linguagem para figuras hierárquicas genéricas e onde programas possam ser verificados formalmente. Na verdade é muito mais uma proposta teórica, pois no artigo não há indicações sobre resultados práticos. Um artigo anterior, onde se apresentam resultados iniciais concernentes a esta proposta é [23].

E realmente os conceitos propostos são bastante abrangentes. As *transformações gráficas*, por exemplo, são funções que mapeiam figuras em figuras. Essas funções podem ser bastante variadas; por exemplo, transformações geométricas, operações de recorte, remoção de linhas escondidas e mudança de cores, todas podem ser englobadas na abstração *transformação gráfica*. O tipo abstrato de dado *função geométrica* é um tipo particular de função que o tipo abstrato de dado *transformação gráfica* se encarrega de aplicar, e corresponde às habituais transformações de translação, mudança de tamanho, rotação, bem

como de operações sobre as matrizes que definem essas transformações.

O tipo abstrato de dado *figura estruturada em árvore* representa a formalização da noção de desenho hierárquico. Tal como na proposta de Newman, vista na seção 2.3, aqui os nós da árvore representam as figuras e as arestas representam as transformações associadas a cada sub-figura. As operações definidas sobre esse tipo de dado são: inserção de uma figura como sub-figura de outra já existente, incluindo uma transformação gráfica associada; remoção de uma sub-figura; obtenção de uma sub-figura dado seu nome; obtenção de uma transformação gráfica associada a uma sub-figura; e alteração de uma transformação gráfica associada a uma sub-figura. É possível ainda juntar duas figuras, formando uma terceira. Para se obter a saída de figuras seria ainda necessário acionar o comando `post`.

As primitivas gráficas fundamentais são *moveto* e *lineto* que criam segmentos de reta. A existência do tipo *region* permite todas as operações de conjuntos (união, interseção, diferença). A *region* é fundamental também para dar base às transformações gráficas, que sempre aplicam funções a regiões.

Em conclusão pode-se dizer que o artigo proporciona uma visão mais abrangente e formal das linguagens gráficas do que os trabalhos vistos anteriormente; de uma certa forma é a continuação dos trabalhos de Schrack [29] e de Magnenat-Thalmann e Thalmann [20]. Porém, como sua ênfase é na verificação formal de programas gráficos, e não na sua praticidade, a aplicação destas idéias foge um pouco ao escopo do presente trabalho.

Outra proposta encontra-se no artigo de Ng e Marsland [27]. Trata-se apenas de um pré-processador gráfico para linguagens de alto nível onde os autores estão mais interessados em uniformizar a abordagem para as diversas linguagens do que criar uma nova. Ainda outro trabalho é o de Yip [39], que entretanto descreve apenas um conjunto de subrotinas para Pascal baseado na proposta Core [31], a qual, por sinal, foi suplantada como padrão internacional pelo GKS.

Finalmente, pode-se citar a proposta de padrão internacional PHIGS; entre outras coisas, ela padroniza a definição, geração e manipulação de hierarquias de objetos geométricos [19]. Sendo uma norma, a preocupação maior não é com idéias novas e sim com a uniformização de conceitos e nomes; além disso, a proposta objetiva principalmente os grandes sistemas interativos em CAD, onde freqüentemente ocorrem estruturas hierárquicas complexas associadas a grandes estruturas de dados. Poder-se-ia salientar apenas que a *estrutura* é a abstração principal que abriga a hierarquia, e em torno dela existe todo um complexo aparato destinado a permitir o seu armazenamento, modificação e saída gráfica.

Capítulo 3

A linguagem Fig

Neste capítulo é apresentada a linguagem projetada com base no estudo dos trabalhos vistos no capítulo 2. Antes, porém, é importante enfatizar as características que nortearam o projeto:

- trata-se de uma linguagem experimental que poderá ser usada em uma variedade de contextos, mas deve servir principalmente para ser inserida em sistemas compositores de texto;
- embora nova, ela se baseia na linguagem PIC descrita na seção 2.2, procurando ser simples como esta, mas também introduzindo novos recursos;
- para este trabalho não houve preocupação com interatividade. Supôs-se desde o início que o desenho especificado pela linguagem seria resultado da execução de um programa previamente compilado, sem interferência do programador durante a execução.

A seguir apresenta-se uma descrição informal da linguagem resultante, que foi denominada FIG. Embora informal, adotou-se a convenção de utilizar *itálico* para não-terminais e `esta fonte` para as palavras reservadas. No apêndice A encontra-se a descrição formal da gramática de FIG.

3.1 O sistema de coordenadas

Os desenhos produzidos por meio de FIG são bidimensionais e toda a linguagem está estruturada com base no tradicional sistema cartesiano de coordenadas. Assim sendo, o *ponto no plano* é um dos principais elementos de FIG. Este é denotado pelo par (x, y), sendo que x e y são valores reais que podem variar na faixa

permitida pelo processador no qual o programa estiver sendo executado. A conversão para os valores permitidos pelo particular dispositivo gráfico de saída que for ser usado é deixada ao encargo das rotinas do pacote gráfico básico associado a FIG (mais detalhes sobre isto no Capítulo 4).

As unidades padrão adotadas foram o centímetro para medidas lineares e o grau para medidas angulares. Essas unidades podem ser facilmente mudadas de acordo com a conveniência do usuário (veja seção 3.5).

O desenho é especificado numa área retangular representando uma janela (no sentido de "window" do GKS) do plano cartesiano. A posição e as dimensões dessa janela têm um valor padrão que pode também ser alterado pelo programador (veja seção 3.5). Toda parte do desenho que estiver fora da janela será automaticamente recortada; também este aspecto deve ser de responsabilidade do pacote gráfico associado a FIG.

3.2 Forma básica da linguagem e aspectos sintáticos

FIG é uma linguagem de programação convencional, com comandos, variáveis, atribuição a variáveis e os mecanismos de controle habituais. Usa palavras reservadas e todas elas estão em português (sem acentos, no entanto).

Os comandos em FIG devem ser colocados um em cada linha, sem terminação de linha, ou então mais de um comando por linha desde que separados por ponto-e-vírgula. Se um comando não couber numa linha ele pode ser continuado na linha seguinte desde que a linha anterior termine com o caractere "\".

Vários comandos incluem a execução de um ou mais outros comandos. Nesse caso o comando ou comandos a serem executados devem estar delimitados pelas chaves, "{" e "}".

Todo texto numa dada linha precedido pelo caractere "!" é ignorado pelo processador FIG. Isto destina-se aos comentários da linguagem.

3.3 Variáve[illegible] e constantes

A linguagem F[illegible] outras linguagens de programação, possui como elementos bás[illegible] nstantes. Embora haja diferentes *tipos* de variáveis, es[illegible] das; isto é, os tipos são implícitos, depreendidos [illegible] ra vez aparecem as variáveis. Existem d[illegible] pré-definidas; ao longo desta apresentaçã[illegible] o apêndice B encontra-se uma lista comp[illegible]

Os tipos implícitos associados às variáveis e constantes são os seguintes:

- valores numéricos;
- pontos no plano;
- valores enumerados;
- cadeias de caracteres.

A seguir descreve-se cada um destes tipos.

Valores numéricos: esses valores são sempre tratados como números reais; onde houver necessidade de números inteiros é feita uma truncagem apropriada. A notação científica é válida; por exemplo, `0.314e-1`.

Pontos no plano: os pontos compõem-se de dois valores reais, a abcissa e a ordenada. Constantes são expressas pela notação `[x,y]`, onde `x` e `y` podem ser expressões quaisquer. Quando uma variável contiver um ponto, denotam-se seus componentes sufixando o nome da variável com os caracteres `.x` e `.y`, como por exemplo `pt.x` e `pt.y`. Uma característica de FIG é o *movimento implícito* associado à saída gráfica. Este aspecto encontra-se detalhado na seção 3.9.1, mas aqui é relevante mencionar a existência de duas variáveis globais pré-definidas relacionadas a esse movimento. Uma delas é um ponto especial, denominado *coordenada corrente*, e outro, um ângulo especial, denominado *direção corrente*. As variáveis correspondentes são `cc` e `dc` respectivamente.

Valores enumerados: são semelhantes aos valores enumerados de Pascal, porém todos os valores são identificadores e não há ordem implícita entre eles. Destinam-se a definir atributos das primitivas. Todos os possíveis valores enumerados dos atributos estão indicados na seção 3.9.3.1.

Cadeias de caracteres: correspondem ao tipo *string* de algumas linguagens de programação. Constantes deste tipo são cadeias de caracteres entre aspas.

3.4 Expressões

As variáveis e constantes podem ser combinadas em expressões. Para essa combinação o programador dispõe de operadores aritméticos e de funções embutidas na linguagem. Operadores relacionais também existem para comparação entre objetos. A seguir apresenta-se uma relação dos operadores e funções válidos para cada tipo de objeto. Não se permite a combinação de tipos diferentes na mesma

expressão.

valores numéricos:

operadores aritméticos: `+` , `-` (unário e binário), `*`, `/`, `**`, `mod`

funções: `sen`, `cos`, `raizq`, `arctg`, `exp` (e^x), `ln`, `abs`, `frac`, `int`

Todas as funções têm como argumento um único parâmetro deste tipo.

pontos do plano:

operadores aritméticos: `+`, `-` (unários e binários)

função: `dist(p1,p2)`, que fornece a distância entre os pontos p1 e p2

valores enumerados:

não há

cadeias de caracteres:

operadores: `+` (concatenação)

funções: `numcar(e)`, onde se converte a expressão `e` numa cadeia de caracteres e `carnum(c)`, onde se converte a cadeia de caracteres `c` num valor numérico.

O significado das operações aritméticas *binárias* em pontos[1] é o seguinte:

```
[x1,y1] + [x2,y2] = [x1 + x2, y1 + y2]
[x1,y1] - [x2,y2] = [x1 - x2, y1 - y2]
```

Já as operações aritméticas *unárias* em pontos servem para especificação relativa dos mesmos. Neste caso distinguem-se duas categorias:

1. Os componentes entre os colchetes representam na verdade Δx e Δy, ou seja:

```
+[x,y] = [cc.x + x, cc.y + y]
-[x,y] = [cc.x - x, cc.y - y]
```

[1] Os pontos neste caso comportam-se como vetores.

onde cc é a coordenada corrente.

2. os componentes entre os colchetes representam *distância* e *ângulo*, ou seja:

```
+[dist,ang^] = [cc.x + dist*cos(ang),cc.y + dist*sen(ang)]

-[dist,ang^] = [cc.x - dist*cos(ang),cc.y - dist*sen(ang)]
```

As duas categorias são diferenciadas através do ângulo, que deve ser seguido pelo símbolo ^ (ou ↑), conforme indicado.

Além disso, a especificação relativa permite omissão de um dos componentes. Para o caso de Δx e Δy, supõe-se que o valor omitido é zero. Para o caso de distância e ângulo, somente a distância pode ser omitida, assumindo, nesse caso, o valor da variável global pré-definida `dist_pd` (distância padrão).

Exemplos:

```
+[30,] = [cc.x + 30,cc.y]
-[,20] = [cc.x,cc.y - 20]
+[,45^] = [cc.x + dist_pd*cos(45),cc.y + dist_pd*sen(45)]
```

Adicionalmente, os próprios ângulos podem ser relativos. Nesse caso, deve-se colocar a expressão que denota o ângulo também entre colchetes e o colchete esquerdo precedido dos sinais + ou -. Assim especificado, o ângulo resultante será a combinação da expressão com o valor da direção corrente.

Portanto, uma especificação válida para pontos é:

```
+[33,-[45]^] = [cc.x + 33*cos(dc-45),cc.y + 33*sen(dc-45)]
```

3.5 O programa em FIG

Um programa em FIG é composto das seguintes partes: *cabeçalho, definições de figuras* e *programa principal*, estruturadas da seguinte forma:

cabeçalho
definições de figuras
`inicio`
programa principal
`fim desenho`

O *cabeçalho* deve obedecer à seguinte sintaxe:

```
desenho título
escala = valor !fator de escala (padrão = 1)
origem = ponto !origem do desenho (padrão = [0,0])
largura = valor !largura total do desenho (padrão = 20)
altura = valor !altura total do desenho (padrão = 30)
```

Notar que `escala`, `origem`, `largura` e `altura` são todos opcionais, pois o sistema provê valores padrão. Através de `escala` pode-se modificar as unidades de trabalho. Como o centímetro é a unidade padrão, se `escala` tiver o valor 10, a unidade passará a ser o milímetro, pois esse valor é usado para *dividir* todas as medidas lineares que aparecerem no programa. Para usar outra unidade qualquer basta verificar quanto vale um centímetro nessa unidade e fornecer esse número. Cabe aqui mencionar que ângulos são medidos em graus; para alterar este padrão deve-se atribuir um valor conveniente à variável global pré-definida `uni_angulo`. Por exemplo, se se desejarem radianos, basta atribuir a `uni_angulo` o valor $\pi/180$.

Através de `origem`, `largura` e `altura` modifica-se a janela em que aparecerá o desenho.

As *definições de figuras* estão detalhadas na seção 3.8. O *programa principal* é constituído de um ou mais comandos do tipo descrito nas seções 3.6 e 3.9.

3.6 Comandos básicos

Nesta seção descrevem-se os comandos da linguagem não diretamente ligados à produção de figuras; em geral são apenas comandos da linguagem Pascal adaptados.

3.6.1 Atribuição de variáveis e constantes

variável `:=` *expressão* `(atribuição normal)`
constante `:==` *expressão* `(atribuição de constante)`

Constantes não podem aparecer do lado esquerdo de atribuições normais.

3.6.2 Repetição simples

`repetir` *expressão* `vezes {` *comandos* `}`

O resultado da avaliação de *expressão* é truncado.

3.6.3 Repetição composta

variando *variável* de *expressão* ate *expressão* passo *expressão*
executar { *comandos* }

Este é análogo ao comando for de Pascal, acrescido de passo, que pode ser omitido.

3.6.4 Decisão

se *expressão_booleana* entao { *comandos1* } senao { *comandos2* }

A *expressão_booleana* resultará num valor verdadeiro ou falso. Se for verdadeiro, *comandos1* são executados. Caso contrário, *comandos2* são executados, desde que constem (senao é opcional).

O encadeamento de vários comandos de decisão uns dentro dos outros segue as mesmas regras da linguagem Pascal (para resolução da ambigüidade do "else pendente").

Expressão_booleana é definida como:

expressão *operador_relacional* *expressão*

onde podem ser usados os seguintes operadores relacionais habituais de Pascal: >, <, >=, <=, =, <>.

As expressões booleanas podem ser combinadas pelos conectivos et (*and*), ou (*or*) e nao (*not*).

3.6.5 Repetição com decisão

enquanto *expressão_booleana* executar { *comandos* }

É equivalente ao while de Pascal.

3.6.6 Comando caso

caso *identificador* { *tratamento_dos_casos* }

O *tratamento_dos_casos* é uma lista com os seguintes itens:

rótulo numérico: { *comandos* }

Também aqui o valor eventual de *identificador* é truncado para se fazer a associação com os rótulos numéricos (que são necessariamente constantes numéricas inteiras); há porém um emprego especial deste comando, que é visto na seção 3.8.5.

3.6.7 Comando de terminação

```
terminar
```

Este comando interrompe a execução de qualquer grupo de comandos no qual esteja inserido.

3.6.8 Entrada e saída de valores

Entrada e saída foram aspectos não abordados na definição da linguagem. Para efeito de facilidade na implementação foi suposta a disponibilidade dos comandos `readln` e `writeln` de Pascal.

3.7 Figuras

Em FIG, o mecanismo fundamental para tratamento dos elementos gráficos recebe o nome de *figura*. Uma figura em FIG é uma primitiva gráfica *ou* um conjunto de variáveis e instruções que indicam como se deve construir uma determinada figura. A base semântica da figura é o mecanismo de procedimentos das linguagens de programação tradicionais, mas modificado para incluir o conceito de procedimento de display, de Newman, visto na seção 2.3.

Assim sendo, toda figura definida em FIG pode ser "chamada" como um procedimento. Essa chamada provocará em geral a saída gráfica da figura quando da execução do programa. Esse tipo de chamada, porém, difere substancialmente do mecanismo habitual de chamadas de procedimentos, e por isso chamamos de *instanciação* esse processo e de *instância* o seu resultado. A diferença reside no fato de que o programador pode, além de passar parâmetros para a figura, modificar a instanciação com certas *operações*, que são transformações geométricas pré-definidas.

Distinguem-se em FIG a *definição* de figuras e sua *utilização*. Nas próximas seções estes itens são detalhados.

3.8 A definição de figuras

A definição de uma figura é análoga à declaração de um procedimento. Porém, em FIG as definições de figuras devem sempre anteceder o programa principal, e não se permitem definições dentro de definições.

Quanto à sua definição, toda figura possui as seguintes características:

- sistema local de coordenadas

- nome
- parâmetros
- resultados

Antes porém de discutir essas características, faz-se necessário um comentário sobre as regras de escopo de variáveis. Toda figura pode definir suas próprias variáveis locais; todavia, estas não são acessíveis às demais figuras. Somente as variáveis globais (aquelas que aparecem no corpo do programa principal) estão disponíveis dentro de qualquer parte do programa, desde que não haja variáveis locais com mesmo nome; nesse caso, prevalece a variável local. O acesso a variáveis locais tem necessariamente que ser feito através do mecanismo de resultados descrito adiante.

3.8.1 O sistema local de coordenadas

Toda figura é definida num sistema próprio de coordenadas, o *sistema local*, que tem eixos coordenados paralelos ao sistema global. Assim, toda coordenada que aparece dentro da definição de uma figura refere-se ao sistema local e não ao global. Além disso, também estão pré-definidas localmente as variáveis `cc` e `dc`; isto significa que as figuras têm um movimento implícito (ver seção 3.9.1) próprio, que pode ou não coincidir com o movimento do contexto onde a figura está sendo instanciada.

O sistema local é utilizado unicamente para definição de figuras. Para que uma figura seja desenhada ela precisa ser instanciada, direta ou indiretamente, no sistema global, quando então sua posição, tamanho e orientação no plano do desenho ficarão bem determinados.

3.8.2 O nome

O nome da figura é equivalente ao nome de um procedimento. É o nome pelo qual ela será instanciada.

3.8.3 Os parâmetros

Os parâmetros das figuras correspondem à noção habitual de parâmetro passado por valor em Pascal. Os parâmetros podem ser de qualquer dos tipos existentes na linguagem.

Ao definir uma figura, o programador deve especificar quais serão seus parâmetros, se houver algum. Além disso, há dois recursos que sofisticam o uso da parametrização: a definição de *valores padrão* e *combinações* de parâmetros.

Os *valores padrão* de parâmetros existem para permitir omissão de parâmetros no momento da instanciação. Se um determinado parâmetro, para o qual foi definido valor padrão, for omitido, a figura é instanciada como se o valor padrão tivesse sido passado.

As *combinações* de parâmetros são um mecanismo que permite definir uma figura através de um conjunto de parâmetros redundantes. No momento de instanciação, o programador especifica apenas os parâmetros que naquele instante lhe são mais convenientes para construir a figura. Por exemplo, pode-se definir um círculo especificando três pontos da borda ou então o centro e o raio ou ainda dois pontos diametralmente opostos. Na definição da figura `circulo`, o programador declarará todos esses parâmetros, mas também especificando em que combinações eles podem aparecer. No corpo da definição de `circulo` o programador deve então colocar instruções que permitam a construção do círculo a partir de qualquer das combinações declaradas, identificando, de acordo com a sintaxe fornecida, qual conjunto de instruções corresponde a qual combinação. Veja exemplo na seção 3.8.6.

3.8.4 Os resultados

Os *resultados* têm uma certa semelhança com o conceito de parâmetro de saída das linguagens de programação convencionais. Os resultados podem ser referenciados, após a instanciação da figura, através de uma sintaxe semelhante à dos campos de registro de Pascal; mais detalhes na seção 3.9.2.1.

O objetivo dos resultados é proporcionar dados geométricos relativos à figura instanciada para que estes possam ser usados na instanciação de outras figuras. Assim sendo, seu uso destina-se muito mais aos tipos valor numérico e ponto no plano do que aos demais.

Por exemplo, na definição de um círculo podemos especificar que sua área é um resultado. Assim, a cada instanciação do círculo, com um determinado raio (este, um parâmetro), estará disponível a área dessa particular instanciação do círculo.

Os resultados dividem-se em duas categorias: os *resultados implícitos* e os *resultados explícitos*. Resultados implícitos são aqueles que a linguagem automaticamente se encarrega de oferecer para todas as figuras, independentemente de especificação por parte do programador. Resultados explícitos são aqueles que requerem uma declaração especial, como se vê adiante.

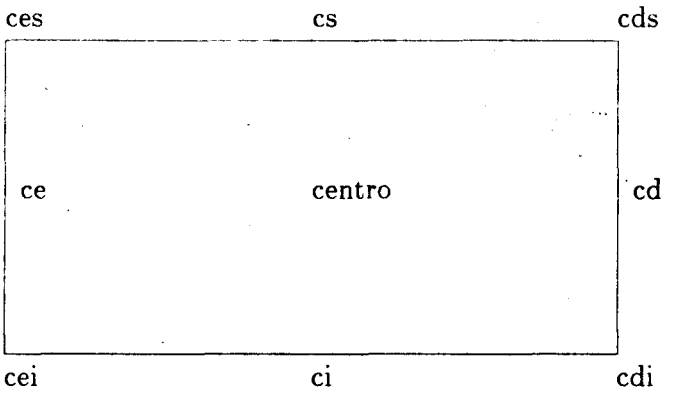

Figura 3.1: Os retângulo envolvente das figuras e seus pontos

Os resultados implícitos são os pontos do retângulo envolvente da figura, definido como sendo o menor retângulo, com lados paralelos aos eixos coordenados, que envolve completamente a figura; automaticamente estão disponíveis nove pontos desse retângulo, conforme a figura 3.1, além de sua altura e largura.

Os nomes desses pontos significam apenas abreviações convenientes:

`ces`: canto esquerdo superior

`cs`: centro superior

`cds`: canto direito superior

`ce`: centro esquerdo

`centro`

`cd`: centro direito

`cei`: canto esquerdo inferior

`ci`: centro inferior

`cdi`: canto direito inferior

Finalmente, os próprios parâmetros também podem ser resultados, desde que declarados explicitamente. Com isto, o usuário pode obter o valor de um parâmetro de uma combinação diferente daquela utilizada num dado instante. Por exemplo, um círculo pode ser instanciado usando a combinação "três pontos não colineares", e o usuário obtém como resultado o raio, este um parâmetro da combinação "raio,centro" (note-se, entretanto, que instruções para o cálculo do raio a partir de três pontos devem constar da definição da figura, neste exemplo).

3.8.5 Sintaxe de definição de figuras

A definição de figuras deve obedecer à seguinte sintaxe:

```
figura nome
parametros {
nome = valor padrão
nome = valor padrão
:
}
combinacoes {
1:  lista de parâmetros
2:  lista de parâmetros
:
}
resultados { lista de resultados explícitos }
inicio
comandos
fim
```

Observações:

- parâmetros, combinações e resultados são opcionais;
- Os parâmetros também podem estar separados por vírgulas;
- Os nomes de parâmetros que aparecem nas combinações têm que constar obrigatoriamente da lista de parâmetros previamente declarada;
- Para associar as combinações aos comandos que definem a figura deve-se usar o comando `caso` (seção 3.6.6) com identificador do caso idêntico ao nome da figura e com rótulos idênticos aos rótulos numéricos usados na declaração das combinações;

- Um resultado explícito só estará disponível se durante a instanciação for feita uma atribuição a ele.

3.8.6 Exemplos de definições de figuras

Abaixo seguem alguns exemplos que procuram ilustrar os diversos aspectos discutidos acima. Esses exemplos supõem que existe a figura primitiva `linha` definida na seção 3.9.3. Esta primitiva admite como parâmetros uma lista de pontos.

```
figura quadrado
parametros { lado = 10 }
resultados { area }
inicio
linha +[lado,], +[,lado], -[lado,], -[,lado]
area := lado * lado
fim

figura quadrilatero
parametros {
larg = 10, alt = 5, p1 = cc, p2, p3, p4 }
combinacoes {
1: p1, larg, alt  ! casos 1,2,3 e 4 produzem um retangulo
2: p2, larg, alt
3: p3, larg, alt
4: p4, larg, alt
5: p1, p2, alt    ! casos 5,6,7 e 8 produzem um paralelogramo
6: p3, p4, alt
7: p2, p3, larg
8: p1, p4, larg
9: p1, p3         ! casos 9 e 10 produzem um retangulo
10: p2, p4
11: p1, p2, p3    ! produz um paralelogramo
12: p1, p2, p3, p4  ! produz um quadrilatero generico
}
inicio
caso quadrilatero {
1: { linha p1, +[larg,], +[,alt], -[larg,], p1 }
2: { linha p2, +[,alt], -[larg,], -[,alt], p2 }
3: { linha p3, -[larg,], -[,alt], +[larg,], p3 }
4: { linha p4, -[,alt], +[larg,], +[,alt], p4 }
```

```
5: { linha p1, p2 + [,alt], p1 + [,alt], p1 }
6: { linha p3, p4, -[,alt], p3 - [,alt], p3 }
7: { linha p2, p3, -[larg,], p2 - [larg,], p2 }
8: { linha p1, +[larg,], p4 + [larg,], p4, p1 }
9: { linha p1, [p3.x,p1.y], p3, [p1.x,p3.y], p1 }
10:{ linha p2, [p2.x,p4.y], p4, [p2.y,p4.x], p2 }
11:{ linha p1, p2, p3, [p3.x - (p2.x-p1.x),p3.y - (p2.y-p1.y)], p1}
12:{ linha p1, p2, p3, p4, p1 }
}
fim
```

3.9 A utilização de figuras

Nesta seção descrevem-se o comando de instanciação de figuras bem como mecanismos associados.

3.9.1 O movimento implícito

FIG, como se verá adiante, possui um mecanismo para realizar o posicionamento das figuras instanciadas em termos de coordenadas absolutas (absolutas dentro do sistema de coordenadas no qual forem expressas). Porém, à maneira de PIC, também dispõe de um *movimento implícito* associado à execução do programa. Isto significa que as figuras podem ser invocadas sem explicitação de suas posições, e isto faz com que elas se posicionem uma após a outra, concatenadas por pontos bem determinados.

Para a consecução deste mecanismo existem duas variáveis globais pré-definidas, já mencionadas: a *coordenada corrente* (`cc`) e a *direção corrente* (`dc`). O valor da primeira indica o ponto a partir do qual a próxima figura a ser instanciada será desenhada, caso se mantenha o movimento implícito. A instanciação provoca atualização automática desta variável. Já a direção corrente indica em que direção se dá o movimento. Esta pode variar entre as seguintes possibilidades:

- da esquerda para a direita: `leste`
- da direita para a esquerda: `oeste`
- de cima para baixo: `sul`
- de baixo para cima: `norte`

Os identificadores `norte`, `sul`, `leste` e `oeste` são constantes pré-definidas com os valores, respectivamente: 90^, 270^, 0^ e 180^, onde o símbolo ^ indica, neste caso, graus.

O usuário pode a qualquer instante fazer alteração nessas variáveis através de atribuição. Note-se entretanto que `dc` só pode receber um dos quatro valores listados acima. Conforme mencionado na seção 3.8.1, o sistema local de coordenadas de uma definição de figura também tem seu próprio movimento implícito, bem como coordenada corrente e direção corrente locais.

Os valores padrão para `cc` e `dc` no início do programa e no início da definição de uma figura são `[0,0]` e `leste`, respectivamente.

As figuras são posicionadas implicitamente levando em conta o seu *ponto de concatenação*; este é sempre o centro do lado do retângulo envolvente relacionado à direção corrente. Assim, por exemplo, se o movimento é para `leste` então o ponto de concatenação é o centro do lado direito da última figura com o centro do lado esquerdo da próxima, independentemente das sub-figuras que compõem as figuras que estão sendo concatenadas (certas primitivas, no entanto, possuem pontos especiais de concatenação; veja seção 3.9.3). Em outras palavras, pode-se dizer que as figuras são posicionadas levando em conta o *espaço* que realmente ocupam no plano, e não sua posição em relação à origem local.

3.9.2 O comando de instanciação

Conforme já mencionado, uma figura é análoga a um procedimento, portanto na sua forma mais simples o comando de instanciação é apenas o nome da figura, seguido eventualmente de seus parâmetros.

A forma geral do comando, no entanto, é a seguinte:

rótulo *operações* *nome_de_figura* *parâmetros* *posicionamento*

nome_de_figura é simplesmente o nome com o qual a figura foi definida. As demais partes do comando acima são a seguir detalhadas.

3.9.2.1 O rótulo

O *rótulo* (opcional), é um nome sucedido pelos dois pontos (":"). Ele nomeia a instância da figura de modo a permitir referência posterior a seus resultados. Isto significa que resultados *só são acessíveis por instâncias de figuras que tenham sido rotuladas*. Nesse caso, basta sufixar o rótulo com o nome do resultado à maneira dos campos de registro em Pascal.

nome	parâmetros
`rodar`	ângulo, ponto (padrão: cc)
`ampliar ou reduzir`	fx, fy (padrão: fx), ponto (padrão: cc)
`espelhar`	$p1$, $p2$ (pontos)

Tabela 3.1: Operações disponíveis e seus parâmetros.

Por exemplo, supondo definida a figura `abc`, e supondo que ela tenha resultados `r1` e `r2`, que seriam pontos, e `vv0` que seria um valor, o seguinte trecho de programa é sintaticamente correto em FIG:

```
xyz:  abc !  abc está aqui sendo instanciada
a := xyz.r1.x - xyz.vv0
b := xyz.r2
```

3.9.2.2 As operações

As *operações* são transformações geométricas de que o usuário dispõe para modificar as figuras definidas. Essas transformações são aquelas habitualmente encontradas em sistemas gráficos: rotação, mudança de escala e espelhamento (para a translação há um recurso próprio). Cada operação tem seu conjunto de parâmetros, como se vê adiante.

Uma figura pode ser transformada através de uma ou mais operações, mas é muito importante se observar em que ordem elas são aplicadas, pois transformações geométricas não são em geral comutativas [7]. Quando houver mais de uma operação, elas (e seus parâmetros) devem vir separadas pelo símbolo "|". A operação que for especificada *por último* será aplicada em primeiro lugar. Por outro lado, pode-se imaginar que as operações são sempre aplicadas após uma primeira instanciação "virtual" da figura. Operações como um todo são opcionais.

As operações disponíveis estão mostradas na tabela 3.1.

Nessa tabela, devem-se observar as seguintes convenções quanto aos parâmetros: para cada operação devem ser fornecidos os parâmetros efetivos separados pela vírgula; os parâmetros que tiverem valor padrão podem ser omitidos, mas os separadores permanecem, a não ser que o parâmetro omitido seja o último de sua respectiva lista.

O ponto que aparece em `rodar` e `ampliar/reduzir` (estes são sinônimos) é usado como referência para fazer a rotação ou a mudança no tamanho. Os

parâmetros fx e fy indicam o fator de escala no eixo x e no eixo y, respectivamente, da mudança de tamanho gerada por `ampliar/reduzir`. Eles se comportam da seguinte forma:

- $f > 1$: provoca ampliação;
- $1 > f > 0$: provoca redução;
- $0 > f > -1$: provoca redução com espelhamento em torno do eixo x ou y, conforme o caso;
- $-1 > f$: provoca ampliação com espelhamento em torno do eixo x ou y, conforme o caso.

Finalmente, $p1$ e $p2$ de `espelhar` são pontos que devem definir uma linha em torno da qual será realizado o espelhamento.

Exemplos:

```
rodar 135^ abc
rodar 270^, [31,41] abc
rodar 315^ | ampliar 2 , 3 abc
reduzir 0.5,, [99,11] | rodar 45^, [0,0] abc
```

3.9.2.3 Os parâmetros

Os *parâmetros* são uma lista de parâmetros efetivos relacionada à lista de parâmetros formais contida na definição da figura que se quer instanciar. Se a figura tiver mais de um parâmetro formal, os parâmetros efetivos devem ser separados por vírgula. Ademais, há duas formas de organizar a lista de parâmetros efetivos: usando sintaxe *não-posicional* e *posicional*.

Na notação não-posicional o nome do parâmetro formal deve anteceder o parâmetro efetivo, ligados pelo símbolo "=" e os parâmetros efetivos podem vir em qualquer ordem. Na notação posicional, não é preciso usar o nome do parâmetro formal, mas deve ser preservada a ordem com que os parâmetros formais foram declarados.

Em ambos os casos é possível a omissão de parâmetros formais, desde que estes tenham valor padrão. Na notação não-posicional simplesmente omite-se o parâmetro formal; na notação posicional, o "lugar vago" do parâmetro omitido deve continuar entre duas ",", exceto no caso do último.

Quando a figura instanciada tiver combinações, devem-se tomar cuidados especiais. Para a primeira combinação declarada são válidas as regras acima; para

as demais, necessariamente deve-se usar a notação não-posicional; eventuais ambigüidades são resolvidas tomando-se a primeira combinação dentre as possíveis que satisfizer a lista de parâmetros efetivos passada. A passagem de uma combinação inexistente provoca erro de compilação.

Parâmetro especial FIG ainda permite que seja especificado um parâmetro especial do tipo enumerado, que não precisa constar da declaração de parâmetros formais de nenhuma figura: é o *atributo genérico*. Trata-se de um atributo (ver seção 3.9.3.1) qualquer que será aplicado a todas as primitivas que forem chamadas na instanciação da figura para as quais fizer sentido esse atributo.

Por exemplo, supondo definida a figura xyz com parâmetros `param1` e `param2`, a seguinte instanciação é válida:

```
xyz param2 = 32, param1 = 64, pontilhado
```

3.9.2.4 O posicionamento

Para realizar o posicionamento explícito da figura deve-se usar a seguinte sintaxe:

`com` *ponto1* `em` *ponto2*

O *ponto1* deve necessariamente ser um resultado (implícito ou explícito) da figura, em que se deve omitir o nome da figura mas preservar o ponto (".") na frente do resultado. Já *ponto2* é um ponto qualquer. Tanto `com` quanto `em` podem ser omitidos. Se `em` for especificado mas não `com`, usa-se o centro do retângulo envolvente para o posicionamento; se apenas `com` for especificado, este ponto coincidirá com `cc`.

Este é um mecanismo que permite não apenas o posicionamento absoluto, como também a realização do encadeamento entre figuras, através da utilização de pontos que são resultados da figura instanciada (no `com`) em conjunção com pontos que são resultado de instâncias de outras figuras (no `em`). Os resultados utilizados, no entanto, precisam ser evidentemente pontos.

Exemplos:

```
xyz:  quadrado
triangulo com .ces em xyz.cdi
triangulo com .centro em [12,34]
```

3.9.3 Figuras primitivas

FIG proporciona uma série de figuras primitivas, que na realidade são procedimentos de biblioteca.

As figuras primitivas comportam-se de forma praticamente igual às figuras definidas pelo próprio usuário, com as seguintes exceções:

1. algumas primitivas permitem como parâmetro um tipo chamado de *lista de pontos*, que é uma lista de número arbitrário de pontos, separados por vírgula;
2. algumas primitivas têm ponto de concatenação não relacionado ao retângulo envolvente;
3. associado às primitivas existe um conjunto de propriedades, os *atributos*, que são descritos na seção 3.9.3.1, adiante.

As primitivas disponíveis são as seguintes:

- linha
- seta
- spline
- marcador
- retângulo
- polígono
- arco de circunferência
- círculo
- arco de elipse
- elipse
- texto

A seguir apresentam-se as características de cada uma delas na forma de um trecho de sua definição em FIG. Estão omitidos os comandos que realizam as primitivas, que dependem das primitivas disponíveis do particular pacote gráfico em uso. Note-se ainda que, exceto quando indicado, as primitivas se concatenam no movimento implícito tal como as figuras de usuário, isto é, tomando como referência o retângulo envolvente.

Linha

```
figura linha !  traça segmentos de reta
parametros { lp } !  uma lista de pontos
!  valor padrao:  p1 = cc, p2 = cc + [dist_pd,dc^]
resultados {
prim,ult !  são o primeiro e último pontos
}
```

Concatena-se pelos pontos prim e ult.

Seta

```
figura seta !  traça segmentos de reta, terminados por uma seta
parametros { lp } !  uma lista de pontos
!  valor padrao:  p1 = cc, p2 = cc + [dist_pd,dc^]
resultados {
prim,ult !  são o primeiro e último pontos
}
```

Concatena-se pelos pontos prim e ult.

Spline

```
figura spline !  traça uma spline cúbica pelos pontos dados
parametros { lp } !  uma lista de pontos
!  valor padrao:  nao tem
resultados {
prim,ult !  são o primeiro e último pontos
}
```

Concatena-se pelos pontos prim e ult.

Marcador

```
figura marcador !  traça símbolos pontuais
parametros { lp } !  uma lista de pontos
!  valor padrao:  1 ponto, em cc
resultados {
prim,ult !  são o primeiro e último pontos
}
```

Concatena-se pelos pontos prim e ult.

Retângulo

```
figura retangulo !  traça um retangulo com lados paralelos
!  aos eixos coordenados
parametros {
largura = larg_pd
altura = alt_pd
p1 = cc, p2 !  cantos do retangulo
}
combinacoes {
1:  p1, largura, altura !  p1 aqui é o canto esquerdo inferior
2:  p1, p2 !  pontos extremos de uma diagonal
}
```

Polígono

```
figura poligono !  traça um polígono regular ou arbitrário
parametros {
lp !  lista de pontos, indicando os vértices (poligono arbitrário)
p1 = cc !  ponto inicial, caso polígono regular
n_lados = 5 !  número de lados, caso polígono regular
t_lado = largura_pd !  tamanho do lado, caso polígono regular
}
combinacoes {
1:  lp !  poligono arbitrario
2:  p1, n_lados, t_lado !  poligono regular
}
```

Arco de circunferência

```
figura arco_c
parametros {
centro = cc + [dist_pd,dc] , raio = dist_pd
ang = 90^ !  ângulo de varredura
ang_ini, ang_fin !  ângulos inicial e final
p1, p2, p3 !  pontos não colineares
}
combinacoes {
1:  centro, raio, ang_ini, ang_fin
2:  centro, p1, p2 !p1 é um extremo, p2 está na radial do outro ex-
tremo
3:  p1, p2, p3 !  não colineares
4:  centro, p1, ang !  p1 define um extremo
}
```

O arco é traçado no sentido anti-horário nos casos 1, 2 e 4.

Círculo

```
figura circulo
parametros {
centro = cc + [dist_pd,dc], raio = dist_pd
p1, p2, p3
}
combinacoes {
1:  centro, raio
2:  p1, p2, p3 !  não colineares
3:  p1, p2 !  linha diametral
}
resultados {
raio, n,s,e,w,nw,sw,ne,se !estes são os pontos cardeais do circulo
}
```

Arco de elipse

```
figura arco_e
parametros {
centro = cc + [dist_pd,dc]
p1, p2, p3
}
combinacoes {
1:  centro, p1, p2 !centro e p1 definem o semi-eixo maior,
                        !p2 está na borda
2:  p1, p2, p3 !p1 e p3 são extremos de um eixo, p2 está na borda
}
resultados {
s_eixo_ma, s_eixo_me, f1, f2 !s_eixo's são as medidas dos semi-eixos
                        !  e f1 e f2 são os focos da elipse (pontos)

}
```

Elipse

```
figura elipse
parametros {
centro = cc + [dist_pd,dc],
s_eixo_ma = dist_pd
s_eixo_me = dist_pd/2
p1, p2, p3
}
combinacoes {
1:  centro, s_eixo_ma, s_eixo_me
2:  p1, p2, p3 !p1 e p3 são extremos do um eixo, p2 está na borda
3:  centro, p1, p2 !  centro e p1 definem o semi-eixo maior e p2
                        !  está em qualquer lugar da borda
}
resultados {
s_eixo_ma, s_eixo_me, f1, f2 !s_eixo's são as medidas dos semi-eixos
                        !  e f1 e f2 são os focos da elipse (pontos)

}
```

Texto

```
figura texto !  é uma cadeia de caracteres
parametros {
cadeia, p1 = cc
}
```

O texto é posicionado em p1 conforme o atributo alinhamento.

3.9.3.1 Os atributos de primitivas

Atributos são propriedades de primitivas que controlam algumas de suas características visuais. São oito tipos de atributos, e cada um se aplica a um certo conjunto de primitivas. Cada tipo de atributo pode assumir diferentes valores dentro dos disponíveis, e estes são pré-definidos em FIG, conforme a lista a seguir:

espessura: `fina, media, grossa`
(aplica-se a `linha, spline, arco_c, arco_e`).

estilo: `continuo, tracejado, traco_ponto, pontilhado`
(aplica-se a `linha, spline, arco_c, arco_e`).

tipo: `ponto, cruz, xis, asterisco, bola`
(aplica-se a `marcador`).

tamanho: `pequeno, medio, grande`
(aplica-se a `marcador e seta`).

conteúdo: `vazio, cheio, hachurado`
(aplica-se a `retangulo, poligono, circulo, elipse`).

hachura: `horiz, vertic, a45, a135`
(aplica-se a `retangulo, poligono, circulo, elipse`).

alinhamento: `ces, cs, cds, ce, centro, cd, cei, ci, cdi`
(aplica-se a `texto`; supõe que toda cadeia pode ser envolvida por um retângulo onde os valores são os mesmos indicados na figura 3.1, porém obedecendo às regras estabelecidas pelo GKS [6]).

invisibilidade: `visivel, invisivel` (aplica-se a todas).

Observações:

1. os atributos de espessura e estilo aplicam-se a `retangulo`, `poligono`, `circulo`, `elipse` se seu conteúdo for `vazio`;

2. o atributo hachura só tem sentido quando conteúdo é `hachurado`;

3. a invisibilidade faz com que não haja saída gráfica efetiva, embora os eventuais resultados sejam calculados;

4. a cada atributo corresponde uma variável global pré-definida, com um certo valor padrão inicial; a lista dessas variáveis acha-se incluída no apêndice B.

3.10 Exemplo de um programa completo em FIG

O programa a seguir desenha a figura 3.2. Supõe-se que os retângulos tenham largura 2 e altura 1 e o círculo tenha raio 0.9.

```
    desenho fluxograma

    figura quadrado
    inicio
    retangulo altura = 2, largura = 2
    fim

    figura losango
    inicio
    rodar 45^ quadrado
    fim

    figura conjunto
    resultados {
    p1, p2 }
    inicio
r1: retangulo
    p1 := r1.cs
    seta
r2: retangulo
    seta
l:  losango
    p2 := l.ci;
    seta l.ce, -[dist_pd,], +[,r2.centro.y - l.ce.y], r2.ce
```

```
    fim

    ! aqui comeca o programa principal
    dist_pd := 0.8;
    dc := sul
    circulo
    seta
    conjunto com .p1 em cc
    seta
l:  losango
    seta 1.cd, +[2,], -[,dist_pd]
c:  reduzir 1,-1 conjunto com .p1 em cc  ! "conjunto" e' espelhado
    linha -[,dist_pd], -[c.p2.x - 1.ci.x,]
    aqui := cc
    seta 1.ce, -[2,], -[,dist_pd]
    retangulo
    seta
r:  retangulo
    linha -[,r.ci.y - aqui.y], aqui
    seta
    circulo
    fim desenho
```

É importante notar que no programa acima não há nenhuma coordenada absoluta.

3.11 Origem das características de FIG

O leitor atento deste capítulo e do precedente certamente terá identificado a origem das diversas características de FIG. Mesmo assim, cremos ser de bom alvitre explicitar as influências que os trabalhos consultados tiveram sobre o projeto da linguagem.

projeto geral da linguagem: PIC, conforme já mencionado, foi a base utilizada para a parte gráfica, e Pascal para os mecanismos de controle;

figuras: são uma adaptação dos procedimentos de display, de Newman;

combinações: são uma tentativa de capturar numa linguagem procedimental as equações simultâneas encontradas em IDEAL;

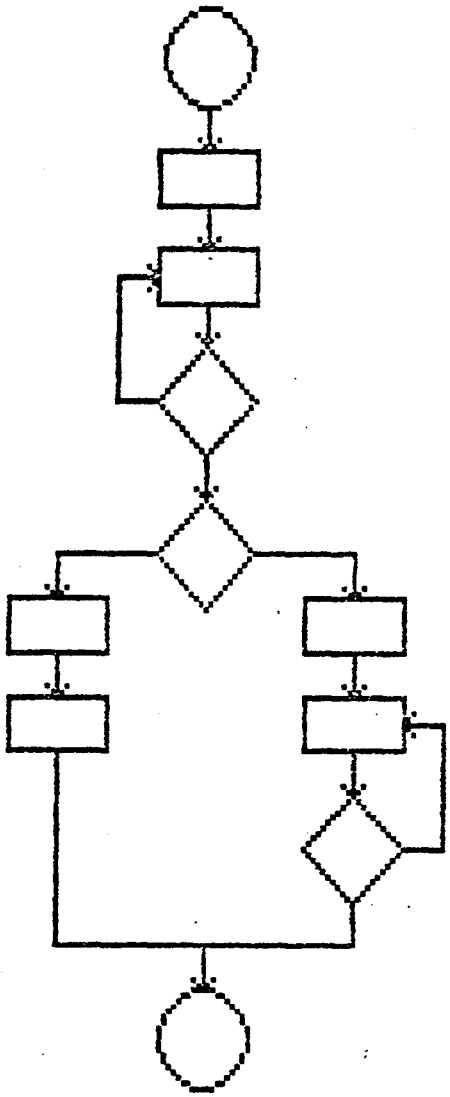

Figura 3.2: O resultado da execução do programa mostrado na seção 3.10.

resultados: são uma adaptação das variáveis locais de IDEAL;

movimento implícito: é uma característica de PIC;

primitivas e seus atributos: são inspirados nas primitivas e em aspectos da noção de atributo encontrados na norma GKS [6];

mecanismos de passagem dos parâmetros: a idéia das notações posicional e não-posicional é originária da extensão de Pascal existente na implementação do sistema VAX/VMS [37], embora ela também exista na linguagem ADA.

Capítulo 4

A implementação de Fig

Embora o objetivo de FIG seja de estar inserida num ambiente de composição gráfica de textos, o presente trabalho apresenta apenas uma implementação da linguagem isolada. Os aspectos relativos à integração com sistemas de composição em geral, e com um sistema em particular, poderiam formar a seqüência do trabalho aqui iniciado, conforme discutido no capítulo 5. Assim sendo, o objetivo principal da implementação foi o de testar a concepção geral da linguagem. Nesse sentido, foram tomadas as seguintes decisões:

- o código-objeto a ser gerado é uma linguagem de alto nível;
- a saída gráfica se realiza por chamadas de subrotinas de um pacote gráfico integrado à linguagem de alto nível, e preferencialmente independente de dispositivos gráficos de saída. Este pacote gráfico se encarregará das tarefas de transformação de vista e do recorte.

A particular linguagem de alto nível escolhida é um aspecto secundário. Já o pacote gráfico idealmente deveria seguir a norma GKS, uma vez que esta representa um padrão internacional. Por outro lado, qualquer implementação de qualquer sistema tem obviamente que levar em conta os recursos de hardware disponíveis. No nosso caso, embora dispuséssemos de um VAX-11/785, a saída gráfica mais fácil poderia ser feita num equipamento do tipo IBM-PC, usando os recursos gráficos do Turbo Pascal [34]. Como estávamos principalmente interessados em explorar aspectos da concepção de FIG, havendo inclusive uma certa escassez de tempo para cumprir esse objetivo, preferimos fazer a opção mais fácil no tocante à saída gráfica, portanto escolhendo Turbo Pascal, com saída dependente de dispositivo (ou seja, o vídeo de um PC). Porém, como se verá a seguir, o projeto do sistema fez com que as rotinas gráficas ficassem encapsuladas, tornando-se simples uma adaptação para outros pacotes gráficos.

4.1 O processamento de um programa em FIG

O processamento pelo qual deve passar um programa em FIG está ilustrado na figura 4.1. Como se vê na figura, o compilador FIG atua como um pré-processador para o compilador Pascal. Já as primitivas fazem parte de um módulo pré-compilado, que é a biblioteca do sistema. É neste módulo que se encontram as definições das primitivas com chamadas para as rotinas do pacote gráfico. Caso seja adotado um pacote gráfico diferente, bastará alterar essas chamadas; e se o pacote seguir a norma GKS, todo o conjunto se torna independente de dispositivo gráfico de saída.

À parte isso, nada impediria que o usuário fosse capaz de inserir suas próprias figuras na biblioteca, tornando-as também primitivas do sistema. Pode-se imaginar ainda que essas figuras poderiam tanto ser definidas em FIG, como também em Pascal, se assim fosse conveniente.

4.2 Alguns aspectos da implementação

4.2.1 O tradutor

O tradutor propriamente dito foi implementado usando o compilador Pascal disponível no ambiente VAX/VMS. Assim, o programa em FIG é editado no VAX e traduzido ainda no VAX pelo compilador FIG. O resultado dessa tradução é então transferido a um PC e compilado e executado com o Turbo Pascal, obtendo-se finalmente a saída.

Foi escolhida a técnica de análise sintática descendente recursiva na implementação do tradutor. Trata-se de uma técnica simples mas razoavelmente efetiva. Foi utilizado o modelo de um analisador sintático apresentado em [15]; nesse modelo, a geração de código é entremeada com a análise sintática. Com exceção do que está mencionado na seção sobre resultados, adiante, toda a tarefa de tradução é executada em apenas um passo.

A tabela de símbolos é tratada como uma pilha que pode conter símbolos (identificadores) de diferentes tipos. Nesse aspecto, a instanciação de figuras comporta-se exatamente como uma chamada de procedimento, com o compilador se encarregando de realizar o empilhamento e desempilhamento das variáveis locais das figuras. Apenas os parâmetros formais merecem um tratamento especial, conforme descrito na seção 4.2.3.

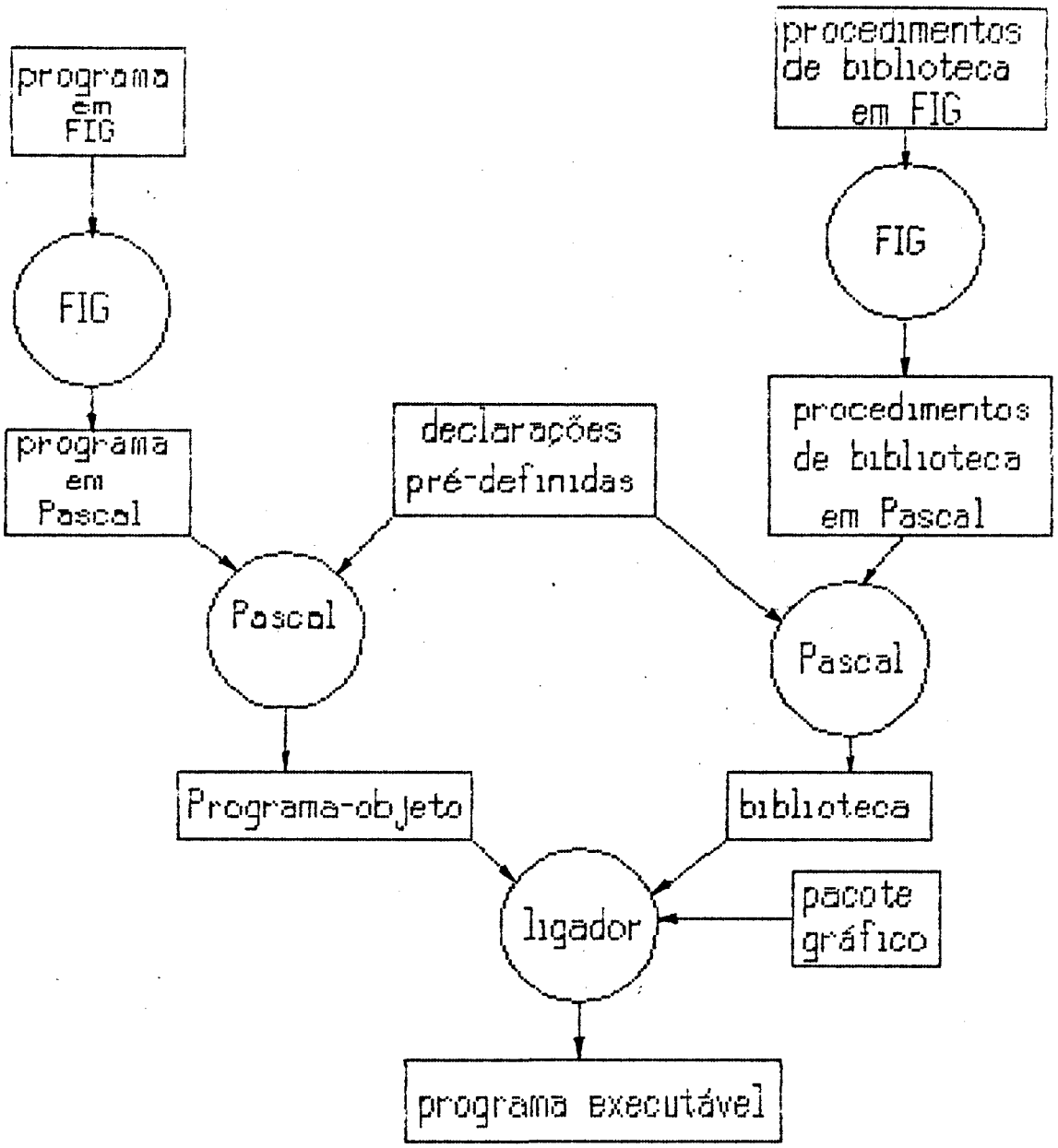

Figura 4.1: O processamento de um programa em FIG

4.2.2 Tradução de figuras

As figuras de FIG foram traduzidas diretamente em procedimentos de Pascal. Os parâmetros formais do procedimento são os mesmos da figura, acrescidos de alguns outros "ocultos" ao usuário de FIG.

Porém, a chamada das figuras obedece ao conceito de procedimento de display, realizado em FIG no comando de instanciação (operações-figura-posicionamento). O problema-chave da implementação dos procedimentos de display é a correta aplicação das transformações geométricas.

Conforme mencionado na seção 2.3 as transformações que ocorrem num dado ponto da hierarquia do desenho devem ser concatenadas com todas as transformações ocorridas em níveis hierarquicamente superiores. Para alcançar esse objetivo, existem técnicas já amplamente discutidas na literatura ([7], [26]). A principal delas consiste em usar uma *matriz de transformação corrente* (MTC). Nessa técnica, as coordenadas não são transformadas até que seja feita a saída de uma primitiva do pacote gráfico que está sendo usado. Nesse momento aplica-se a MTC às coordenadas dessa primitiva.[1] Ao ser aplicada, a MTC contém todas as transformações individuais (translação, rotação, mudança de tamanho) que ocorreram desde a raiz da árvore que representa a hierarquia, e que foram concatenadas por pré-multiplicação na MTC. A pré-multiplicação faz com que a mais recente transformação individual seja aplicada em primeiro lugar, o que é exatamente o desejado. Por outro lado, primitivas podem ser chamadas dentro de uma figura que chama outras sub-figuras. Isto exige a manutenção de um contexto para a MTC, que deve ser restaurado após o retorno da sub-figura. Assim sendo, o processo de instanciação dentro do corpo de uma figura deve ser traduzido na forma

salvamento de contexto
concatenação de eventuais transformações
chamada da sub-figura
restauração do contexto

O projeto geral das rotinas para auxiliar a instanciação das figuras nesta implementação baseou-se em proposta contida em [30]. Os nomes das rotinas são:

```
t_save (* faz o salvamento do contexto de transformações *)
```

[1] Supõe-se que a matriz esteja expressa em coordenadas homogêneas [7, p. 249].

```
t_restore (* restaura o contexto de transformações *)
t translate (* concatena uma translação com a MTC *)
t_rotate (* concatena uma rotação com a MTC *)
t_scale (* concatena uma mudança de tamanho com a MTC *)
```

Finalmente, cumpre notar que é no corpo das primitivas que se encontram as chamadas para as rotinas de saída gráfica do pacote gráfico associado. Este é o trecho do compilador que teria que ser reescrito para cada novo pacote gráfico utilizado, além, é claro, das chamadas para inicialização e término do pacote, que também são específicas. Particularidades como existência ou não de posição corrente e forma de associação de atributos às primitivas, no pacote, também devem ser levadas em conta.

4.2.3 Passagem de parâmetros

Os parâmetros formais, além de serem colocados temporariamente na pilha de variáveis locais, são também armazenados numa lista ligada especial, onde cada nó contém as seguintes informações: nome, tipo, valor padrão, valor efetivo, combinação a que pertence, e dois campos booleanos: se há ou não valor padrão, e se foi ou não passado um valor. Essa lista é montada quando se faz a análise sintática da figura.

Quando surge uma instanciação, o compilador primeiramente verifica se a notação sintática utilizada é posicional ou não-posicional. Se for posicional, basta seguir a lista ligada e preencher o campo *valor efetivo*; quando houver omissão de parâmetro, basta preencher esse campo com o valor padrão, porém verificando antes a existência deste último. Se for não-posicional, a cada parâmetro efetivo percorre-se a lista em busca do formal correspondente; a cada achado, preenchem-se as informações necessárias e posiciona-se o apontador novamente no início da lista. Após o término dos parâmetros efetivos, a lista é novamente percorrida para verificar quais parâmetros foram omitidos, e completando os nós com os valores padrão. Nesse momento gera-se o código com a chamada do procedimento que representa a figura.

As combinações não foram ainda implementadas; planejamos fazer com que o analisador sintático reconheça a combinação que está sendo passada e monte a chamada do procedimento incluindo um parâmetro "oculto" a mais que indicará qual a combinação utilizada. A linguagem exige a notação não-posicional para todas as combinações com exceção da primeira justamente para viabilizar esse reconhecimento. Dentro do corpo do procedimento um simples comando `case`

usando como identificador esse parâmetro permitirá a execução condicional dos diversos trechos que descrevem as combinações.

4.2.4 Resultados

Os resultados das figuras são expressos na tradução como variáveis do tipo registro de Pascal, em que o nome da figura instanciada é usado como nome da variável, prefixando-o ainda com o rótulo, se houver. O tipo registro utilizado é composto de duas partes: uma fixa, onde se encontram os resultados implícitos (e portanto válida para todas as figuras), e uma que varia conforme os resultados eventualmente declarados pelo usuário. Portanto, a definição de uma figura provoca a construção de um tipo registro próprio daquela figura, o qual fará parte dos parâmetros "ocultos" do procedimento correspondente, porém do tipo var (passagem por referência).

O formato da parte fixa dos registros com resultados tem a seguinte declaração:

```
type retenv = record
   cei,cds,centro: ponto;
   larg,alt: real;
   end;
```

Note-se que não há necessidade de incluir todos os pontos da figura 3.1, uma vez que apenas os três acima contêm informações sobre os demais.

Ao encontrar uma chamada de figura, o tradutor deve se encarregar de incluir, na chamada do procedimento correspondente, um parâmetro efetivo "oculto" ao usuário, que é uma variável do tipo registro, específico da figura que está sendo instanciada. Na execução da figura, essa variável é preenchida e os resultados passam a estar disponíveis. Caso a instância não seja rotulada, seus resultados estarão numa variável temporária, muito embora transparente ao usuário. Com uma tabela auxiliar seria relativamente fácil obter um comportamento semelhante a PIC, em que é possível referenciar instâncias não rotuladas pela sua ordem dentro do programa (primeira, segunda, última, etc), mas isto não foi feito.

O problema básico desta abordagem é a geração do código contendo as declarações de tipos e de variáveis. O código referente à declaração do tipo de uma figura deve constar no início do programa Pascal objeto, e no entanto só pode ser gerado após a análise da seção `resultados` das figuras. O mesmo ocorre com as variáveis desses tipos, relativas às instâncias das figuras: estas só são conhecidas em sua totalidade após a análise completa do programa. Uma solução simples para esse problema — que foi adotada nesta implementação — é o armazenamento num arquivo auxiliar das informações de constituição dos tipos e

nomes das variáveis. Após a análise completa de um programa correto, é feito um reprocessamento do arquivo objeto, levando em conta as informações contidas no arquivo auxiliar, inserindo as declarações de tipos e de variáveis nos locais apropriados. Isto evidentemente equivale a ter um compilador de dois passos.

Um problema que foi notado na fase de implementação e deve ser estudado com mais rigor é a eventual alteração dos valores dos resultados devido às operações aplicadas sobre a figura. Por exemplo, se se define como resultado uma *área*, a operação `ampliar/reduzir` certamente produzirá uma mudança nesse valor. Isto não está sendo levado em conta na atual implementação, pois o compilador não tem condições de saber se um valor numérico é um número com dimensões (linear ou de área) ou não. A resolução deste problema talvez implique em algum tipo de declaração por parte do usuário.

4.2.5 O posicionamento relativo

A combinação dos procedimentos de display com posicionamento relativo mostrou-se o aspecto mais problemático da implementação. A razão disto é que o posicionamento relativo exige conhecimento do resultado da instanciação *antes* de realizar a saída gráfica propriamente dita. Para clarificar essa questão, vejamos como se daria a compilação da seguinte frase (supondo definida a figura `abc`):

```
rodar 45^ abc com .ces em p1
```

Esta frase deve resultar na seguinte tradução:

```
t_save;
t_translate(Δ);
t_rotate(45);
abc(fig_r_abc);
t_restore
```

O problema todo está em saber quanto vale o Δ que aparece na translação. Na verdade, ele é apenas o resultado da avaliação da expressão `p1 - fig_r_abc.ces`, onde `fig_r_abc.ces` teria que ser conhecido por uma instanciação prévia de `abc` que não precisaria levar em conta nenhuma translação, mas que também não deveria realizar nenhuma saída gráfica. A solução encontrada foi a seguinte:

```
t_save(false);
t_rotate(45);
abc(fig_r_abc);
t_restore;
```

```
Δ := p1 - fig_r_abc.ces;
t_save(true);
t_translate(Δ);
t_rotate(45);
abc(fig_r_abc);
t_restore
```

Como se vê, toda instanciação com posicionamento relativo no programa fonte resulta na verdade em *duas* instanciações no programa objeto. Como as instanciações estão estruturadas numa hierarquia, isto significa um número exponencial de instanciações. Uma figura instanciada no nível n da hierarquia, será na verdade instanciada 2^n vezes, sendo que somente a última deve resultar em saída gráfica! Para programas que tenham muitas figuras, em vários níveis hierárquicos, isto pode resultar em grande ineficiência.

O problema poderia ser atenuado se fosse feito um pré-processamento do programa gerando num arquivo auxiliar dados geométricos sobre todas as instâncias de figuras que aparecem no programa. No momento da execução esse arquivo seria consultado para que o posicionamento relativo pudesse ser feito sem instanciação prévia. Evidentemente isto quebraria o espírito de independência de estrutura de dados, mas desde que ficasse transparente ao usuário, não parece apresentar maiores inconvenientes.

Finalmente, é preciso explicar o parâmetro de `t_save`. Trata-se de um mecanismo que indica quando a instanciação deve realmente realizar a saída gráfica. Quando `false`, é uma instanciação prévia, e portanto não deve realizar saída; quando `true`, é uma instanciação que pode ocasionar saída, dependendo da situação em que se encontra o próprio procedimento que está gerando esse código. Esta situação é levada em conta da seguinte forma: o parâmetro é armazenado na pilha de contexto, porém somente depois de multiplicado logicamente pelos parâmetros anteriores (assim levando em conta o status das instanciações superiores hierarquicamente). Com isso, no momento de invocação de uma primitiva do pacote gráfico, basta verificar o valor no topo da pilha; a saída só é realizada se este for verdadeiro (portanto significando que toda a cadeia de chamadas está realizando uma instanciação com saída).

4.2.6 O movimento implícito

O movimento implícito seria facilmente implementado através de atualizações convenientes das variáveis *coordenada corrente* e *direção corrente*. Porém, um

problema surge pelo fato de que em FIG se permite que figuras tenham um movimento implícito próprio. Isto pode fazer com que o movimento do programa principal seja, num dado instante, diferente do movimento da figura correntemente instanciada. Para resolver esse problema é indispensável obter informações sobre o retângulo envolvente da figura *antes* de realizar a saída gráfica. Por exemplo, se a direção corrente do programa principal é `oeste` e a da figura instanciada é `leste`, é preciso saber a largura do retângulo envolvente da figura antes de fazer a chamada de `t_translate`. Para resolver isto basta aproveitar o mecanismo de instanciação prévia descrito na seção 4.2.5.

Capítulo 5

Conclusão

Conforme já mencionado, este trabalho faz parte de um projeto mais amplo que visaria a integração de uma ferramenta para confecção de desenhos num sistema de composição gráfica como TEX, por exemplo. Neste capítulo procuramos sugerir algumas das extensões para a linguagem que vislumbramos durante a execução do trabalho, bem como expor as etapas que poderiam dar continuidade a este projeto.

A extensão mais interessante seria a modificação do mecanismo de *operações*, de modo a permitir que o usuário pudesse montar as suas próprias operações sem depender daquelas embutidas na linguagem. O usuário poderia, por exemplo, combinar duas operações primitivas que utiliza com freqüência numa só. Ou então criar uma operação própria, como *cisalhamento*, que não pode ser conseguida pela combinação de rotação, translação e mudança de tamanho. Para realizar isto bastaria que o usuário montasse, com uma sintaxe conveniente, uma matriz de transformação geométrica em coordenadas homogêneas com os valores que desejar, dando-lhe um nome qualquer. O compilador FIG se encarregaria de aplicar essa matriz quando verificasse a presença desse nome no programa. Esta sugestão já aparece no artigo de Newman [24], discutido na seção 2.3.

Outra extensão importante seria o aperfeiçoamento do mecanismo de parametrização das figuras, tanto no aspecto de passagem de parâmetros efetivos quanto das combinações. A passagem de parâmetros idealmente deveria ser o mais flexível possível, podendo o usuário combinar a seu bel-prazer quantos e quais parâmetros quisesse, e com ou sem referência ao nome do parâmetro formal correspondente. Para isto, no entanto, é necessário um mecanismo de análise sintática e semântica bem mais sofisticado do que o proposto nesta implementação. Por outro lado, talvez fosse bastante interessante permitir que figuras pudessem ser parâmetros de outras figuras, o que, à primeira vista, não parece

ser difícil de realizar, uma vez que a definição de Pascal permite que procedimentos sejam parâmetros de outros procedimentos. E a combinação de parâmetros poderia seguir o modelo de IDEAL, que parece ser o mais flexível. Porém, não está claro como fazer para que o usuário não tenha que saber algo sobre resolução de sistemas de equações, como acontece em IDEAL.

Ainda outra extensão poderia ser a inclusão do tipo *array*. É duvidoso, no entanto, que o enriquecimento indiscriminado da linguagem conduza a benefícios reais ao usuário.

Quanto a possíveis etapas posteriores do trabalho, antes de tentar a integração com um sistema compositor de textos seria fundamental uma utilização experimental da linguagem por parte de um grupo de usuários, no sentido de promover um *feed-back* ao projeto e aferir as diversas características da linguagem. Somente depois de recolhidas as críticas advindas de uma tal utilização, e implementadas eventuais modificações, é que se deve iniciar o trabalho de integração com TEX ou outro sistema. Uma vez integrado, novas críticas poderiam surgir, com conseqüente mudança da linguagem até se atingir um estágio considerado razoável pela maioria dos usuários.

A integração com um programa de composição gráfica apresenta logo de início um problema básico: seria ou não necessário fazer uma alteração nesse programa para incluir a capacidade de figuras? Para que não seja preciso, o programa de composição deve conter algum tipo de recurso para que figuras possam ser compostas através de um conjunto de macros que atuariam como uma espécie de "driver" gráfico. Esta solução, no entanto, pode vir a fazer mau uso do recurso de saída final (a impressora ou compositora), caso as figuras sejam simuladas com pequenos caracteres, como acontece no sistema LATEX [17]. Já que a tendência parece ser a incorporação da capacidade de figuras nos programas compositores, cremos que vale a pena a segunda opção, ou seja, fazer ou alterar esses programas para que ofereçam uma capacidade gráfica sofisticada sobre a qual uma linguagem como FIG poderia ser implementada.

Apêndice A

A gramática de Fig

Segue abaixo a gramática da linguagem FIG. Foi utilizada a notação BNF estendida, de acordo com a exposta em [33]. Nela, são usadas as seguintes convenções:

- os caracteres [e] servem para delimitar seqüências de símbolos opcionais;
- os caracteres { e } são usados para denotar repetição de zero ou mais vezes da seqüência de símbolos neles contidos; caso o símbolo deva ocorrer pelo menos uma vez, isto é indicado no fechamento da chave por $\}_1$;
- o caractere | indica alternativas;
- os caracteres (e) servem para agrupar diferentes alternativas de seqüências de símbolos;
- os caracteres da linguagem que coincidirem com os metalinguísticos acima são colocados entre 'aspas simples'.

Um aspecto mais propriamente léxico e não definido nesta gramática é o da separação dos itens de certas listas abaixo (como *comandos*). Em princípio parece ser interessante permitir tanto mudança de linha quanto o tradicional ponto-e-vírgula.

⟨*programa*⟩ : : = ⟨*cabeçalho*⟩
{⟨*definição de figura*⟩}
`inicio`
⟨*programa principal*⟩
`fim desenho`

```
⟨cabeçalho⟩ ::= desenho ⟨identificador⟩
                [escala = ⟨valor⟩]
                [origem = ⟨ponto⟩]
                [largura = ⟨valor⟩]
                [altura = ⟨valor⟩]

⟨definição de figura⟩ ::= figura ⟨identificador⟩
                          [ parametros '{'
                          {⟨identificador⟩[ = ⟨expr⟩] }1
                          '}' ]
                          [ combinacoes '{'
                          {⟨rótulo numérico⟩: ⟨identificador⟩{, ⟨identificador⟩}1}1
                          '}' ]
                          [ resultados '{'
                          ⟨identificador⟩ {,⟨identificador⟩ }
                          '}']
                          inicio
                          {⟨comando⟩}
                          fim

⟨programa principal⟩ ::= {⟨comando⟩ }

⟨comando⟩ ::= ⟨identificador⟩ := ⟨expr⟩ |
              ⟨identificador⟩ :== ⟨expr⟩ |
              repetir ⟨expr⟩ vezes '{' { ⟨comando⟩ } '}' |
              variando ⟨identificador⟩ de ⟨expr⟩ ate ⟨expr⟩ [ passo ⟨expr⟩]
                       executar '{' { ⟨comando⟩ } '}' |
              se ⟨expr⟩ entao '{' { ⟨comando⟩ } '}'
                       [ senao '{' { ⟨comando⟩ } '}' ] |
              enquanto ⟨expr⟩ executar '{' { ⟨comando⟩ } '}' |
              caso ⟨identificador⟩ '{'
                   { ⟨rótulo numérico⟩: '{' { ⟨comando⟩ } '}' }1
                   '}' |
              terminar |
              ⟨instanciação⟩

⟨instanciação⟩ ::=
              [⟨identificador⟩:] [ ⟨operação⟩ { '|' ⟨operação⟩ } ]
              ⟨identificador⟩ { ⟨parâmetro efetivo⟩ } [ com ⟨ponto⟩] [em ⟨ponto⟩]

⟨operação⟩ ::= rodar ⟨expr⟩ [, ⟨ponto⟩] |
```

```
                (ampliar reduzir) ⟨expr⟩ [, ⟨expr⟩ [, ⟨ponto⟩]] |
                espelhar ⟨ponto⟩, ⟨ponto⟩

⟨parâmetro efetivo⟩ ::= ⟨expr⟩ {, ⟨expr⟩ } |
                        ⟨identificador⟩ = ⟨expr⟩ {, ⟨identificador⟩ = ⟨expr⟩ }

⟨identificador⟩ ::= ⟨letra⟩ {⟨letra⟩|⟨dígito⟩|_} [.⟨identificador⟩ [(.x|.y)] ]

⟨valor⟩ ::= ⟨número⟩| ⟨identificador⟩ | ⟨cadeia⟩

⟨número⟩ ::= [+|-] {⟨dígito⟩}1(.{ ⟨dígito⟩ } [e [+|-] {⟨dígito⟩}1]|
             e[+|-] {⟨dígito⟩}1)

⟨cadeia⟩ ::= " ⟨letra⟩ { ⟨letra⟩ } "

⟨ponto⟩ ::= [+|-] '[' [⟨expr⟩],[⟨expr⟩ [^] ] ']'

⟨rótulo numérico⟩ ::= {⟨dígito⟩}1

⟨expr⟩ ::= ⟨expr⟩ ⟨operador⟩ ⟨expr⟩ | -⟨expr⟩ | nao ⟨expr⟩ |
           '('⟨expr⟩')' | ⟨identificador⟩. | ⟨número⟩ | ⟨ponto⟩
           | ⟨chamada de função⟩

⟨operador⟩ ::= + | - | * | / | mod | ** | > | < | <> | = | >= | <= | et | ou

⟨chamada de função⟩ ::= ⟨identificador⟩ '('{ ⟨expr⟩ }')'
```

dígito e *letra* têm o significado habitual.

Apêndice B

Variáveis e constantes pré-definidas

São as seguintes:

nome	tipo	espécie	valor (ou valor padrão)
dist_pd	valor numérico	variável	2
alt_pd	valor numérico	variável	1
larg_pd	valor numérico	variável	2
raio_pd	valor numérico	variável	1
uni_angulo	valor numérico	variável	1
cc	ponto	variável	[0,0]
dc	valor numérico	variável	leste
norte	valor numérico	constante	90°
sul	valor numérico	constante	270°
leste	valor numérico	constante	0°
oeste	valor numérico	constante	180°
espess_pd	valor enumerado	variável	fina
estilo_pd	valor enumerado	variável	continuo
tipo_pd	valor enumerado	variável	ponto
tam_mk_pd	valor enumerado	variável	pequeno
tam_seta_pd	valor enumerado	variável	pequeno
conteudo_pd	valor enumerado	variável	vazio
hachura_pd	valor enumerado	variável	a135
alinham_pd	valor enumerado	variável	cei
visib_pd	valor enumerado	variável	visivel
pi	valor numérico	constante	π

Apêndice C

Exemplo de tradução

Neste apêndice é mostrado um exemplo de programa em FIG, com sua tradução e resultado de execução.

O desenho a ser obtido é um exemplo simples de hierarquia e representa três estágios de um registrador de deslocamento. Um estágio do registrador é composto de um transistor ("etrans"), de um inversor ("inverter") e de algumas linhas. O inversor por sua vez é composto de dois transistores ("etrans" e "dtrans"), de duas instâncias do símbolo da potência ("power") e também de algumas linhas. Finalmente os elementos básicos, que são os transistores e o símbolo da potência são compostos apenas por linhas. O exemplo foi retirado de [30].

A seguir o programa em FIG.

```
desenho esquematico
origem = [4,8]

figura power
inicio
linha [0,0], [0,-1], [0.5,-1], [0,-2], [-0.5,-1], [0,-1]
fim

figura etrans
resultados {p1,p2,p3}
inicio
linha [-2,0], [0,0]
linha [0,-0.5], [0,0.5]
linha [1,-1.5], [1,1.5]
```

```
linha [2,-3], [2,-1], [1,-1]
linha [2,3], [2,1], [1,1]
p1 := [2,3]
p2 := [2,-3]
p3 := [-2,0]
fim

figura dtrans
resultados {p1,p2}
inicio
linha [0,0], [-1,0], [-1,-2], [2,-2]
linha [2,-3], [2,-1], [1,-1]
linha [2,3], [2,1], [1,1]
linha [1,-1.5], [1,1.5]
linha [1.3,-1], [1.3,1]
linha [0,-0.5], [0,0.5]
p1 := [2,3]
p2 := [2,-3]
fim

figura inverter
resultados {p1}
inicio
a: etrans
b: dtrans com .p2 em a.p1
power com .cs em a.p2
rodar 180^ power com .ci em b.p1
linha a.p1, +[1,] , [,3] +[1,]
p1 := a.p3
fim

figura shift_register
inicio
a: rodar -90^ etrans
inverter com .p1 em a.p1
linha a.p3, +[,8]
linha [cc.x - 3,cc.y], +[12,]
fim
! programa principal
```

```
repetir 3 vezes { shift_register }
fim desenho
```

Este programa se executado produz a figura C.1. Na figura C.2 estão indicados os componentes do registrador de deslocamento.

Abaixo está a tradução do programa FIG acima. As rotinas básicas, que se baseiam naquelas apresentadas em [30], não estão mostradas. Convém notar também que esta tradução apresenta uma série de simplificações que aproveitam as características deste particular desenho para não tornar muito longo o programa Pascal resultante. Finalmente, a versão de Pascal deste programa é da implementação Turbo [34], inclusive as rotinas gráficas.

```
program esquematico;

(* aqui se iniciam as declaracoes pre-definidas *)

const
   cor = 3;
   ty = 199;
   maxreal = 1e37;
   f = 8;

type
   ponto = record
      x,y: real
      end;
   t_matrix = array [1..3,1..3] of real;
   points = array [1..50] of ponto;
   index_type = 1..3;
   retenv = record     (* parte comum do retangulo envolvente *)
      cei,cds,centro: ponto;
      larg,alt: real;
      end;
   direcao = (leste,norte,oeste,sul);
   tamanho_pilha = 0..30;

var
```

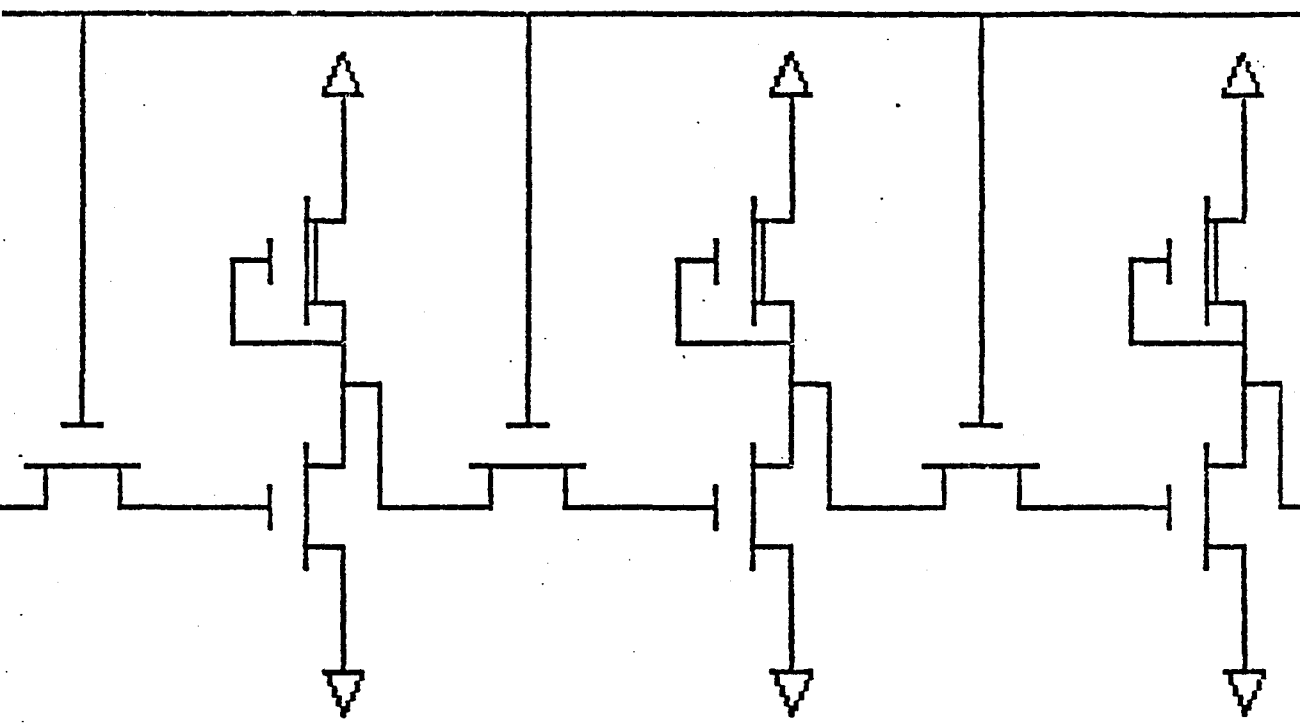

Figura C.1: Um exemplo de desenho hierárquico. Baseado em [30, pag. 319].

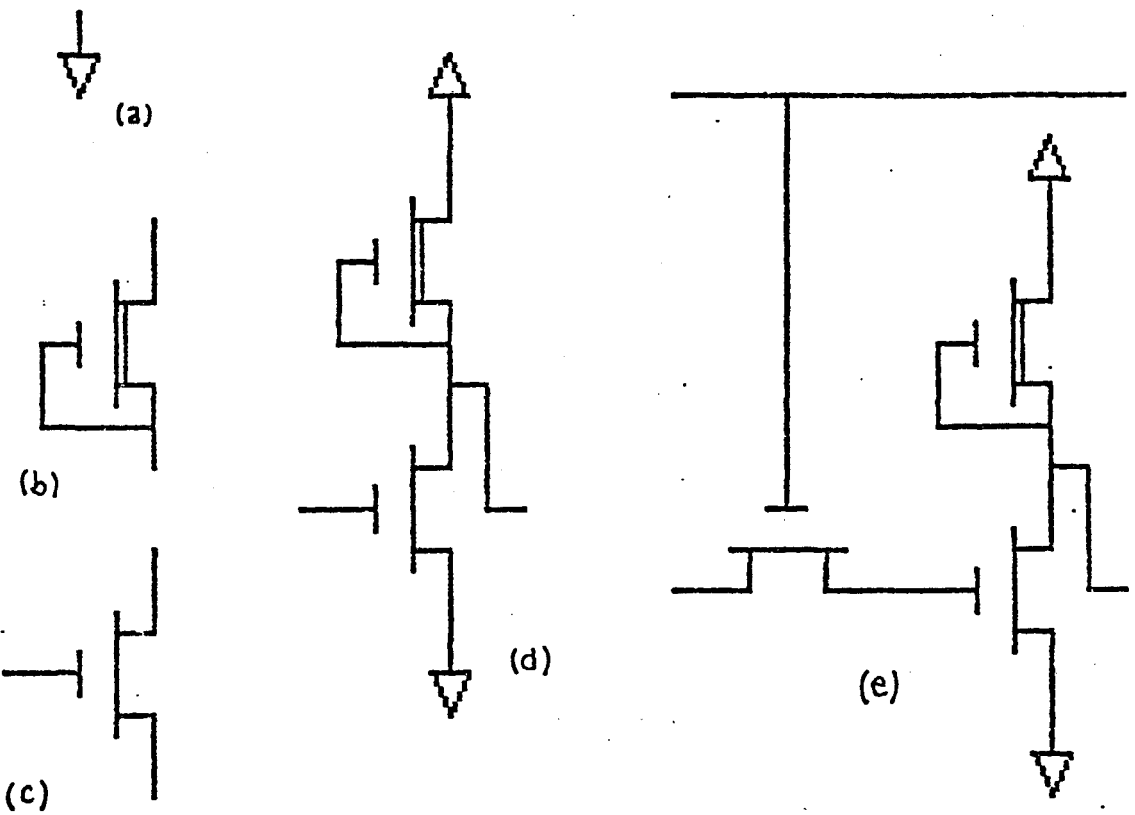

Figura C.2: Componentes do desenho mostrado na figura anterior. (a) power (b) dtrans (c) etrans (d) inverter (e) shift register

```
    lp,p: points;     (* usados para pontos da linha *)
    mt_corr,mt_temp: t_matrix; (* MTC e uma matriz auxiliar *)
    rtv_corr: retenv; (* retangulo envolvente corrente *)
    grau_rad: real;
    i_pilha: 0..30;
    i: integer;
    pilha: array[1..30] of t_matrix;
    r_pilha: array [0..30] of retenv;
    s_pilha: array [0..30] of boolean;
    dir: direcao;
    d: points;  (* para armazenar deslocamentos relativos *)
    cc: ponto;

(* aqui se iniciam declaracoes deste particular programa *)

type
  linha_retenv = record
     fixo: retenv;
     prim,ult: ponto;
     end;
  power_retenv = record
     fixo: retenv;
     end;
  etrans_retenv = record
     fixo: retenv;
     p1,p2,p3: ponto;
     end;
  dtrans_retenv = record
     fixo: retenv;
     p1,p2: ponto;
     end;
  inverter_retenv = record
     fixo: retenv;
     p1: ponto;
     end;
  sr_retenv = record
     fixo: retenv;
     end;
```

```
var
   r_shift_register1,r_shift_register2: sr_retenv;
   r_linha: linha_retenv;

(* aqui deveriam aparecer as rotinas basicas pre-definidas; estas
   seguem as rotinas de Sproull, Sutherland e Ullner (1985) *)

procedure linha(n: integer; l: points; var r_linha: linha_retenv);
(* primitiva de FIG - procedimento de biblioteca *)
var
   i: integer;
   lt: points;
begin
   fig_r_init;
   for i := 1 to n do begin
      t_transform(l[i].x,l[i].y,lt[i].x,lt[i].y);
      fig_r_atu_xy(lt[i].x,lt[i].y);
      end;
   fig_r_set_lhc;
   r_linha.fixo := rtv_corr;
   r_linha.prim := l[1];
   r_linha.ult := l[n];
   if s_pilha[i_pilha] then  (* saida grafica? *)
      for i := 1 to n-1 do
  (* aqui esta' a chamada do pacote grafico; neste caso e' a rotina
     do Turbo Pascal que desenha uma linha dados dois extremos *)
         draw(round(f*lt[i].x),ty-round(f*lt[i].y),round(f*lt[i+1].x),
         ty-round(f*lt[i+1].y),cor);
end; (* linha *)

(* aqui se inicia o resultado da traducao *)

procedure power(var r_power: power_retenv);
var
cc: ponto; dir: direcao;
begin
fig_r_init;
cc.x := 0; cc.y := 0; dir := leste;
lp[1].x := 0; lp[1].y := 0;
```

```
lp[2].x := 0; lp[2].y := -1;
lp[3].x := 0.5; lp[3].y := -1;
lp[4].x := 0; lp[4].y := -2;
lp[5].x := -0.5; lp[5].y := -1;
lp[6].x := 0; lp[6].y := -1;
linha(6,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
fig_r_set_lhc;
r_power.fixo := rtv_corr;
end;

procedure etrans(var r_etrans: etrans_retenv);
var
cc,pos_inic: ponto; dir: direcao;
begin
fig_r_init;
cc.x := 0; cc.y := 0;  dir := leste;
pos_inic := cc;
lp[1].x := -2; lp[1].y := 0;
lp[2].x := 0; lp[2].y := 0;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 0; lp[1].y := -0.5;
lp[2].x := 0; lp[2].y := 0.5;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 1; lp[1].y := -1.5;
lp[2].x := 1; lp[2].y := 1.5;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 2; lp[1].y := -3;
lp[2].x := 2; lp[2].y := -1;
lp[3].x := 1; lp[3].y := -1;
linha(3,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 2; lp[1].y := 3;
lp[2].x := 2; lp[2].y := 1;
lp[3].x := 1; lp[3].y := 1;
linha(3,lp,r_linha);
```

```
fig_r_atu_r(r_linha.fixo);
fig_r_set_lhc;
r_etrans.fixo := rtv_corr;
(* resultados explicitos *)
r_etrans.p1.x := 2; r_etrans.p1.y := 3;
r_etrans.p2.x := 2; r_etrans.p2.y := -3;
r_etrans.p3.x := -2; r_etrans.p3.y := 0;
end;

procedure dtrans(var r_dtrans: dtrans_retenv);
var
cc: ponto; dir: direcao;
begin
fig_r_init;
cc.x := 0; cc.y := 0; dir := leste;
lp[1].x := 0; lp[1].y := 0;
lp[2].x := -1; lp[2].y := 0;
lp[3].x := -1; lp[3].y := -2;
lp[4].x := 2; lp[4].y := -2;
linha(4,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 2; lp[1].y := -3;
lp[2].x := 2; lp[2].y := -1;
lp[3].x := 1; lp[3].y := -1;
linha(3,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 2; lp[1].y := 3;
lp[2].x := 2; lp[2].y := 1;
lp[3].x := 1; lp[3].y := 1;
linha(3,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 1; lp[1].y := -1.5;
lp[2].x := 1; lp[2].y := 1.5;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
lp[1].x := 1.3; lp[1].y := -1;
lp[2].x := 1.3; lp[2].y := 1;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
```

```
lp[1].x := 0; lp[1].y := -0.5;
lp[2].x := 0; lp[2].y := 0.5;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
fig_r_set_lhc;
r_dtrans.fixo := rtv_corr;
(* resultados explicitos *)
r_dtrans.p1.x := 2; r_dtrans.p1.y := 3;
r_dtrans.p2.x := 2; r_dtrans.p2.y := -3;
end;

procedure inverter(var r_inverter: inverter_retenv);
var
cc: ponto; dir: direcao;
a_etrans: etrans_retenv;
b_dtrans: dtrans_retenv;
r_power: power_retenv;
d: points;  (* para armazenar deslocamentos relativos *)
begin
fig_r_init;
cc.x := 0; cc.y := 0; dir := leste;
(* inicialmente todas as instanciacoes sao virtuais *)
fig_t_save(false);
t_set_ident(mt_corr);
etrans(a_etrans);
fig_t_restore;
fig_t_save(false);
t_set_ident(mt_corr);
dtrans(b_dtrans);
fig_t_restore;
(* o vetor d armazena os deslocamentos determinados pelo
 posicionamento relativo *)
d[1].x := a_etrans.p1.x - b_dtrans.p2.x ;
d[1].y := a_etrans.p1.y - b_dtrans.p2.y ;
fig_t_save(false);
t_set_ident(mt_corr);
power(r_power);
fig_t_restore;
d[2].x := a_etrans.p2.x - r_power.fixo.centro.x;
```

```
d[2].y := a_etrans.p2.y - r_power.fixo.cds.y;
fig_t_save(false);
t_set_ident(mt_corr);
fig_t_rotate(180);
power(r_power);
fig_t_restore;
d[3].x := b_dtrans.p1.x - r_power.fixo.centro.x  + d[1].x;
d[3].y := b_dtrans.p1.y - r_power.fixo.cei.y + d[1].y;

(* apos determinacao de todos os deslocamentos, podem ser
 realizadas as instanciacoes reais *)
fig_t_save(true);
etrans(a_etrans);
fig_t_restore;
fig_r_atu_r(a_etrans.fixo);
fig_t_save(true);
fig_t_translate(d[1].x,d[1].y);
dtrans(b_dtrans);
fig_t_restore;
fig_r_atu_r(b_dtrans.fixo);
fig_t_save(true);
fig_t_translate(d[2].x,d[2].y);
power(r_power);
fig_t_restore;
fig_r_atu_r(r_power.fixo);
fig_t_save(true);
fig_t_translate(d[3].x,d[3].y);
fig_t_rotate(180);
power(r_power);
fig_t_restore;
fig_r_atu_r(r_power.fixo);
lp[1].x := a_etrans.p1.x; lp[1].y := a_etrans.p1.y;
lp[2].x := lp[1].x + 1; lp[2].y := lp[1].y;
lp[3].x := lp[2].x; lp[3].y := lp[2].y - 3;
lp[4].x := lp[3].x + 1; lp[4].y := lp[3].y;
fig_t_save(true);
linha(4,lp,r_linha);
fig_t_restore;
fig_r_atu_r(r_linha.fixo);
```

```
fig_r_set_lhc;
r_inverter.fixo := rtv_corr;
r_inverter.p1.x := a_etrans.p3.x;
r_inverter.p1.y := a_etrans.p3.y;
end;

procedure shift_register(var r_shift_register: sr_retenv);
var
cc: ponto; dir: direcao;
pos_inic: ponto;
a_etrans: etrans_retenv;
a1,a2,a3: ponto;
r_inverter: inverter_retenv;
d: points;  (* para armazenar deslocamentos relativos *)
begin
cc.x := 0; cc.y := 0; dir := leste;
pos_inic := cc;
fig_r_init;
fig_t_save(false);
t_set_ident(mt_corr);
fig_t_rotate(-90);
etrans(a_etrans);
(* como houve uma rotacao, e' preciso transformar os resultados
 desta instancia para que a rotacao seja levada em conta *)
t_transform(a_etrans.p1.x,a_etrans.p1.y,a1.x,a1.y);
t_transform(a_etrans.p2.x,a_etrans.p2.y,a2.x,a2.y);
t_transform(a_etrans.p3.x,a_etrans.p3.y,a3.x,a3.y);
fig_t_restore;
fig_t_save(false);
t_set_ident(mt_corr);
inverter(r_inverter);
fig_t_restore;
d[1].x := a1.x - r_inverter.p1.x;
d[1].y := a1.y - r_inverter.p1.y;

fig_t_save(true);
fig_t_rotate(-90);
etrans(a_etrans);
fig_t_restore;
```

```
fig_r_atu_r(a_etrans.fixo);
fig_t_save(true);
fig_t_translate(d[1].x,d[1].y);
inverter(r_inverter);
fig_t_restore;
fig_r_atu_r(r_inverter.fixo);
lp[1].x := a3.x; lp[1].y := a3.y;
lp[2].x := lp[1].x; lp[2].y := lp[1].y + 8;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
cc.x := r_linha.ult.x; cc.y := r_linha.ult.y;
lp[1].x := cc.x - 3; lp[1].y := cc.y;
lp[2].x := lp[1].x + 12; lp[2].y := lp[1].y;
linha(2,lp,r_linha);
fig_r_atu_r(r_linha.fixo);
fig_r_set_lhc;
r_shift_register.fixo := rtv_corr;
end;

(* programa principal - comandos pre-definidos *)
begin
graphmode;
grau_rad := pi/180;
i_pilha := 0;
s_pilha[0] := true;
t_set_ident(mt_corr);
fig_r_init;
(* comandos traduzidos *)
cc.x := 4; cc.y := 8; dir := leste;
fig_t_save(true);
fig_t_translate(cc.x,cc.y);
shift_register(r_shift_register1);
fig_t_restore;
cc.x := r_shift_register1.fixo.cds.x;
cc.y := r_shift_register1.fixo.cei.y;
for i := 1 to 2 do begin
   fig_t_save(false);
   t_set_ident(mt_corr);
   shift_register(r_shift_register2);
```

```
    fig_t_restore;
    (* este e' um caso particular do movimento implicito *)
    d[1].x := r_shift_register1.fixo.cds.x - r_shift_register2.fixo.cei.x;
    d[1].y := 8;
    fig_t_save(true);
    fig_t_translate(d[1].x,d[1].y);
    shift_register(r_shift_register2);
    fig_t_restore;
    r_shift_register1 := r_shift_register2;
    end;

readln;
textmode;
end.
```

Bibliografia

[1] Abelson, H. e A. A. diSessa, *Turtle Geometry*, MIT Press, 1981.

[2] Aho, A., R. Sethi e J. D. Ulmann, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[3] Bentley, J. L. e B. W. Kernighan, "GRAP - A language for typesetting graphs. Tutorial and User Manual", Computing Science Technical Report no. 114, AT&T Bell Laboratories, 1984.

[4] Brinch Hansen, P., "The Programming Language Concurrent Pascal", IEEE *Trans. on Soft. Eng.* , **1** (2), pp. 199–207, 1975.

[5] Domingues, A. L. e J. C. Setubal, "Aplicação de números complexos para resolução de problemas em geometria", manuscrito não publicado, 1985.

[6] Enderle, G., K. Kansy e G. Pfaff, *Computer Graphics Programming: GKS - The Graphics Standard*, Springer-Verlag, 1984.

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

[8] Graphical Kernel System (GKS), Functional Description, Draft International Standard ISO/DIS 7942 (ISO TC 97/SC5/WG2 N 163), 1982.

[9] Graphics Languages, Anais de simpósio promovido por ACM SIGPLAN e ACM SIGGRAPH, Florida International University, 26–27 de abril, 1976 (publicado também como volume 11, número 6 de SIGPLAN Notices, 1976).

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

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

[12] Kernighan, B. W., M. E. Lesk e J. F. Ossana, "UNIX Time-Sharing System: Document Preparation", Bell Syst. Tech. J., 57(6), pp. 2115-2135, 1978.

[13] Knuth, D. E., *TEX and METAFONT: New Directions in Typesetting*, Digital Press, 1979.

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

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

[16] Kulsrud, H. E., "A General Purpose Graphic Language", Communications of the ACM, vol. 11, no. 4, pp. 247–254, abril, 1968.

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

[18] Lanier, J. Z., citado no comentário da capa de *Scientific American*, 251, 3, setembro, 1984.

[19] Magalhães, L. P., *Computação Gráfica: Interfaces em Sistemas de Computação Gráfica*, Editora da Unicamp, Campinas, 1986.

[20] Magnenat-Thalmann, N. e D. Thalmann, "A Graphical Pascal Extension Based on Graphical Types", *Software – Practice and Experience*, vol. 11, no. 1, pp. 53–62, 1981.

[21] Magnenat-Thalmann, N. e D. Thalmann, "MIRA-3D: A Three- Dimensional Graphical Extension of Pascal", *Software - Practice and Experience*, vol. 13, no. 9, pp. 797–808, 1983.

[22] Mallgren, W. R., "Formal Specification of Graphic Data Types", *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 687–710, 1982.

[23] Mallgren, W. R. e A. C. Shaw, "Graphical Transformations and Hierarchic Picture Structures", *Computer Graphics and Image Processing*, 8, pp. 237–258, outubro, 1978.

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

[25] Newman, W. M., "An Informal Graphics System Based on the LOGO Language", NCC, AFIPS Press, pp. 651–655, 1973.

[26] Newman, W. M. e R. F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, 1979.

[27] Ng, N. e T. A. Marsland, "Introducing Graphics Capabilities to Several High-Level Languages", *Software - Practice and Experience*, vol. 8, pp. 629–639, 1981.

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

[29] Schrack, G., "Design, Implementation and Experience with a Higher-Level Graphics Language for Interactive Computer-Aided Design Purposes", in [9], pp. 10–17, 1976.

[30] Sproull, R. F., W. R. Sutherland e M. K. Ullner, *Device-Independent Graphics*, McGraw-Hill, 1985.

[31] "Status Report of the Graphics Standards Committee", *Computer Graphics*, 13(3), agosto, 1979.

[32] Sutherland, I. E., "SKETCHPAD: A Man-Machine Graphical Communication System", MIT Lincoln Lab. Tech. Rep. 296, maio 1965.

[33] Tremblay, J.-P. e P. G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985.

[34] Turbo Pascal Reference Manual, Borland, Inc. 1984.

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

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

[37] VAX-11 Pascal Language Reference Manual, Digital Equipment Corporation, 1982.

[38] Yaglom, I. M., *Complex Numbers in Geometry*, Academic Press, 1968 (traduzido da edição em russo de 1963).

[39] Yip, C. K., "The Pascal Graphics System", *Software - Practice and Experience*, vol. 14, no. 2, pp. 101–118, 1984.