UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM EDITOR ORIENTADO POR ESTRUTURA PARA LINGUAGENS DIAGRAMÁTICAS

por

ELOI LUIZ FAVERO

Dissertação submetida como requisito parcial para
a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Roberto Tom Price
Orientador

Porto Alegre, dezembro de 1989.

CATALOGAÇÃO NA FONTE

Favero, Eloi Luiz.
Um editor orientado por estrutura para linguagens diagramáticas. Porto Alegre, PGCC da UFRGS, 1989.

Diss.(mestr. ci. comp.) UFRGS-PGCC. Porto Alegre, BR-RS, 1989.

Dissertação: Engenharia de Software: Editores diagramáticos: Editores orientados por estrutura: Geradores de Editores: Gramática de atributos.

(...)

Só se pode verdadeiramente conhecer e explicar quando se reduzem as intuições a uma apreciação exata dos fatos e das suas conexões lógicas.

Um investigador honesto terá de admitir que nem sempre é possível tal redução, mas será desonesto de sua parte não ter isto sempre presente no espírito... É uma ilusão comum acreditarmos que o que sabemos hoje é tudo o que podemos saber sempre. **Nada é mais vulnerável que uma teoria científica, apenas uma tentativa efêmera para explicar os fatos, e nunca uma verdade eterna.**

Carl G. Jung (O homem e seus símbolos, 1964)

Qualquer caminho é apenas um caminho e não constitui insulto algum - para si mesmo ou para os outros - abandoná-lo quando assim ordena o seu coração. (...)

Olhe cada caminho com cuidado e atenção. Tente-o tantas vezes quantas julgar necessário ... Então, faça a si mesmo e apenas a si mesmo uma pergunta: possui este caminho um coração? Em caso afirmativo, o caminho é bom. Caso contrário, esse caminho não possui importância alguma.

Carlos Castañeda (Os ensinamentos de Don Juan, 1980)

À minha mãe,
à Bete, ao Moa, à Maura, ...

à Flori, ...

à memória de meu pai.

Agradecimentos

Agradeço ao meu orientador Prof. Roberto T. Price pelo apoio e pelas críticas que direcionaram este trabalho.

Agradeço a todos professores do CPGCC que direta ou indiretamente motivaram o desenvolvimento deste trabalho.

Um especial agradecimento aos integrantes do projeto ADS (Lúcia, Mônica, Alexandre, Walcélio, Javan, Flávio, Pimenta, Paulo, ...) com os quais convivi nestes três anos como colega de projeto e como amigo.

Este trabalho é fruto da vivência acadêmica onde participam professores, alunos e funcionários. Um agradecimento especial a todos.

SUMÁRIO

LISTA DE ABREVIATURAS

ADS ambiente de desenvolvimento de software

ADSs ambientes de desenvolvimento de software

AS árvore sintática

ASA árvore sintática abstrata

DD dicionário de dados (relacionado com análise estruturada)

ET editor (textual) orientado por estrutura

ED editor (diagramático) orientado por estrutura

EDG editor diagramático generalizado

GA gramática de atributos

GED gerador de editores diagramáticos dirigidos por sintaxe

GET gerador de editores textuais dirigidos por sintaxe

GLC gramática livre de contexto

inh (inherited) cláusula para definição de atributos do tipo herdado

LDE linguagem de especificação baseada em gramática de atributos

syn (synthesized) cláusula para definição de atributos do tipo sintetizado

LISTA DE FIGURAS

RESUMO

Este trabalho introduz uma nova abordagem na construção de editores para linguagens diagramáticas como as usadas na engenharia de software (por exemplo diagrama de fluxo de dados e diagrama de Nassi-Shneiderman). Esta nova abordagem tem por base a construção do editor a partir da descrição da linguagem diagramática no formalismo chamado gramática de atributos, que é usado na construção de reconhecedores/editores para linguagens textuais. Uma gramática de atributos estende uma gramática livre de contexto com equações semânticas. Assim,a linguagem é descrita tanto a nível sintático (livre de contexto) como de semântica estática (sensível ao contexto). O nível sintático compreende os aspectos relacionados com a estrutura da linguagem, estruturas de grafos (nodos/arcos) ou de árvore. O nível de semântica estática compreende todos os aspectos que não podem ser especificados na sintaxe; por exemplo, as verificações que se fazem sobre as informações das tabelas de símbolos (nomes não declarados, nomes duplicados, etc.).

Uma vez que o editor é orientado pela estrutura da gramática, torna-se adaptável para distintas linguagens pelo uso de distintas descrições gramaticais.

Além dos aspectos relacionados com a geração de editores diagramáticos, o trabalho sugere a integração de editores através do compartilhamento de informações em tabelas de símbolos; informações estas que são mantidas permanentemente consistentes pelo mecanismo de gramática de atributos. Esta integração pode ocorrer entre diferentes editores diagramáticos e/ou textuais.

ABSTRACT

This work introduces a new approach to construct editors for diagrammatic languages used in software engineering (e.g. data flow diagram - DFD, Nassi-Shneiderman chart). In this approach an editor is constructed from the description of the language based on the attribute grammar formalism. This formalism is commonly used to build parsers/language editors for textual languages. An attribute grammar extends a context free grammar with semantic equations. However, a language is described at syntatic (context free) and semantic (context dependent) layers. All aspects related to the structure of the languagem, e.g. graph (nodes/arcs) or tree are described at syntatic layer. The other aspects, wich can not be described in syntatic level, are handled in the semantic layer, for example validations of the symbol table informations (dupicated names, undefined names, etc.)

This kind of editors, oriented by grammar structure, can be adapted for several languages, by the use of different grammars.

This work, also, discusses the integration of the editors by sharing symbol table informations; such informations are collected by the attribute grammar mechanism. This integration occurs between differents textual/diagrammatic editors.

1 INTRODUÇÃO

O objetivo deste trabalho é investigar o uso do formalismo de gramática de atributos (GA) [AHO 86] para a geração de editores para linguagens diagramáticas, como as usadas na engenharia de software (diagrama de fluxo de dados (DFD), diagrama de Nassi-Shneiderman, etc.). Como resultado, com base num protótipo, é apresentada uma proposta de construção de um editor orientado por estrutura para notações diagramáticas.

Um editor orientado por estrutura implementa facilidades de edição independentes de linguagens, i.e., as facilidades de edição são utilizadas para distintas linguagens. As informações estruturais que guiam a edição são representadas em tabelas. Um programa tradutor lê a descrição (da sintaxe e semântica) de uma notação e gera as tabelas usadas pelo editor.

O sistema (editor e programa tradutor) descrito neste trabalho é chamado de Gerador de Editores Diagramáticos (GED). As linguagens para o GED são descritas na metalinguagem de especificação LDE [ESP 89b] baseada em gramáticas de atributos.

1.1 Geradores de ferramentas

Nos últimos anos vem crescendo o interesse em Ambientes de Desenvolvimento de Software (ADSs), como uma coleção de ferramentas textuais (editores de texto, editores para linguagens de programação) e/ou diagramáticas operando de forma integrada. Uma das formas de se construir ADSs é com o uso de geradores de editores.

A necessidade de se utilizar geradores de editores para a construção de ADSs é evidente. Surgem a cada dia novas notações textuais e/ou diagramáticas nas metodologias de desenvolvimento de software. Por outro lado, as notações já existentes e padronizadas com o tempo tendem a evoluir, e então torna-se necessário modificar e adaptar os compiladores, interpretadores, editores, etc. Um gerador de editores reduz os esforços na criação e na manutenção destas ferramentas.

Um gerador de editores é util também na construção de ambientes de edição para outras áreas tais como Comunicação de Dados, Microeletrônica, etc.

1.2 O uso do formalismo gramática de atributos

As modernas ferramentas interativas (e.g. planilhas eletrônicas, processadores de texto, editores orientados por estrutura), baseadas no conceito WYSIWYG ("What you see is what you get"), surgiram com o crescente avanço no desenvolvimento de microcomputadores e estações de trabalho.

A importância da descrição das notações diagramáticas em GA está na existência de algoritmos de avaliação incremental de GAs, o que possibilita a construção de ferramentas sob o paradigma de edição WYSIWYG. É a avaliação incremental que permite a exibição imediata do efeito dos comandos de edição sobre o documento [REP 87].

Na construção de ADS existe um problema de integração de ferramentas pelo compartilhamento de

informações em um banco de dados. Horwvitz [HOR 86] propôs uma solução para este problema estendendo o formalismo de GA com tabelas relacionais, onde as informações das ferramentas que constituem um ambiente de edição são compartilhadas através de um banco de dados.

Também, cabe lembrar que o desenvolvimento das técnicas existentes para a construção de compiladores se deve ao surgimento das linguagens formais. Com a especificação das notações diagramáticas (vistas como linguagens), através do formalismo de gramática de atributos, pode-se alcançar o mesmo avanço técnico.

1.3 O contexto do trabalho no Projeto ADS

Este trabalho foi desenvolvido dentro do Projeto ADS (Ambiente de Desenvolvimento de Software). O projeto ADS tem por objetivo a "criação de um ambiente experimental para construção e experimentação de ferramentas e metodologias de desenvolvimento de software, tais como editores de diagramas e de linguagens orientados por estruturas". Para atingir este objetivo, no Projeto ADS, estão em desenvolvimento sistemas geradores de ferramentas.

Está em desenvolvimento um editor orientado por estrutura para linguagens textuais, chamado Gerador de Editores Textuais (GET) [PRI 84], [FAV 87], [ESP 89b]. O GET aceita como entrada descrições de linguagens textuais na metalinguagem LDE baseada em GA. O formalismo GA mostrou-se adequado para a descrição de linguagens textuais para geradores de compiladores (GAG [KAS 82], [DER 88]) e também para editores orientados por estrutura (Synthesizer Generator [REP 87]).

Também, no Projeto ADS, Melo [MEL 89] desenvolveu um Editor Diagramático Generalizado (EDG) baseado no conceito de grafo: arcos e nodos. Atualmente o EDG está sendo reescrito, buscando-se obter um sistema utilizável em salas de aula e laboratórios. A nível de pesquisa, a proposta do gerador de editores diagramáticos (GED) visa dar continuidade ao trabalho desenvolvido com o EDG.

No GED a especificação da linguagem se faz na metalinguagem LDE, enquanto que no EDG (editor diagramático generalizado) a especificação é feita interativamente. No EDG define-se apenas os nodos e os arcos, o que corresponde ao nível sintático da LDE usada pelo GED.

Originalmente a metalinguagem LDE, utilizada no projeto ADS, foi desenvolvida para a descrição de linguagens textuais. Este trabalho estende esta metalinguagem para a descrição das notações diagramáticas o GED.

O GED apresenta duas vantagens com relação ao EDG:

A primeira é a especificação dos aspectos semânticos e de formatação das notações diagramáticas. Na semântica identifica-se nomes duplicados, nomes não declarados, etc. A especificação da formatação define o posicionamento dos nodos na tela. Por exemplo, pode-se ter uma estrutura hierárquica (árvore) para os nodos.

A segunda vantagem é a integração de distintos editores diagramáticos e/ou textuais pelo acesso às informações que ficam representadas em tabelas de símbolos globais em um banco de dados associado. A integração de ferramentas textuais e diagramáticas deve-se ao uso de uma

única metalinguagem LDE para ambas as notações.

Esta integração de editores textuais e diagramáticos possibilita a criação de sofisticados ambientes de trabalho. Por exemplo, Peters [PET 87] apresenta uma proposta de automatização do processo de desenvolvimento de software que faz uso das ferramentas da tabela da figura 1.1.

modelo orientado a \ fases de desenvolvimento	análise estruturada	projeto estruturado	implementação
processos	-Diagrama de Fluxo de Dados -Dicionário(1) -Pseudocódigo	-Diagrama de Fluxo de Dados -Dicionário(2) -Pseudocódigo(4)	-Diagrama de Fluxo de Dados -Dicionário(3) -Pseudocódigo
informação	-Diagrama de Entidades e Relacionamentos (E-R) -Dicionário(1)	-Projeto Lógico do Banco de Dados -Dicionário(2)	-Implementação do Banco de Dados -Dicionário(3)
eventos	- Modelo de Eventos(5) - Dicionário (1)	- Modelo de Eventos(6) - Dicionário (2)	- Modelo de Eventos(7) - Dicionário (3)

figura 1.1: Ferramentas associadas às etapas do processo de desenvolvimento de software [PET 87].

O modelo orientado a processos trata principalmente do processamento ou transformações dos dados.

O modelo orientado a eventos trata dos aspectos de tempo real do sistema de software.

O modelo orientado a informação trata da definição das informações que devem ser processadas, bem como das relações existentes entre os ítens de informação.

1. Dicionário de dados (DD) da fase de análise: contém todas as definições de dados, pseudocódigos, descrições de ítens de dados, etc.

2. DD da fase de projeto: contém todas as informações do dicionário da fase de análise atualizadas para a fase de projeto, com variáveis de controle, pseudocódigo para cada processo, etc.

3. DD de implementação: contém todas as informações do projeto modificadas com as considerações da fase de implementação, como a lista dos módulos onde uma variável é utilizada.

4. Pseudocódigo: nesta fase a representação do pseudocódigo da fase da análise é expandida com a inclusão de variáveis de controle, etc. Todos os nomes referenciados no pseudocódigo devem estar definidos no dicionário de dados.

5. Modelo de Eventos (ME) da análise: é uma forma simplificada de um diagrama de transição de estados.

6. Modelo de Eventos do projeto: é o ME da análise detalhado com base nas considerações de projeto, tais como: detalhes de tempo e restrições de implementação.

7. Modelo de Eventos da implementação: na fase de implementação o modelo de eventos consiste na definição dos processos, que são executados em paralelo, e dos mecanismos de sincronização entre eles.

Um Ambiente de Desenvolvimento de Software integrado deve prover um conjunto mínimo de facilidades, tais como:

a) listagem dos ítens dos DD em ordem alfabética; listagens parciais; referência cruzada entre ítens no dicionário de dados: identificando ítens usados mas não definidos, ítens definidos mas não usados, etc.

b) identificação de ítens de dados definidos no DD e não definidos numa notação associada (pseudocódigo, ou DFD); e vice versa.

c) referência cruzada entre DFD e diagrama de eventos (DE); entre DFD e pseudocódigo; etc.

d) identificação de todos os documentos que referenciam um determinado item de dados; ou exibição do possível efeito de uma remoção sobre todos os documentos associados; etc.

A construção de protótipos de ambientes, com as

características citadas acima, requer um grande esforço no desenvolvimento e integração de ferramentas (textuais e diagramáticas). No projeto ADS, a construção do GED (diagramático) integrado ao GET (textual) viabiliza a geração de protótipos de ADSs como o exemplificado acima.

1.4 A estrutura do trabalho

O capítulo 2 introduz um conjunto de conceitos relacionados ao uso do formalismo GA na especificação de linguagens de programação. Estes conceitos são discutidos a partir de um exemplo que descreve uma minilinguagem textual.

O capítulo 3 introduz o tema linguagens diagramáticas, classificando-as em 2 tipos: com estrutura em forma de árvore e de grafo. Neste capítulo também é apresentada uma metalinguagem para descrever os elementos léxicos de linguagens diagramáticas.

Os próximos 3 capítulos apresentam exemplos que descrevem aspectos de linguagens diagramáticas.

O capítulo 4 - diagrama de Nassi-Schneiderman - ilustra um exemplo de um diagrama com estrutura de árvore (com hierarquia) ; mostra-se o uso de GA na formatação do diagrama.

O capítulo 5 - diagrama de fluxo de dados - mostra um exemplo de uma linguagem do tipo grafo, com formatação livre. Neste capítulo, a gramática para descrever a sintaxe (GLC) é estendida para permitir a especificação de estruturas de grafo (com nodos compartilhados). Mostra-se, também, o uso de GA para análise semântica.

O capítulo 6 - diagrama de entidades e relacionamentos (E-R) - mostra uma linguagem com estrutura do tipo grafo com formatação hierárquica. Combinam-se as facilidades apresentadas nos capítulos 4 e 5.

Os capítulos 7 a 10 se preocupam com a especifição do protótipo do GED.

O capítulo 7 introduz um gerador de editores diagramáticos (GED). É apresentada uma arquitetura para o GED e, a partir da arquitetura, são detalhados os principais componentes. Também, neste capítulo, comenta-se a relação entre o GED e o gerador de editores textuais (GET).

O capítulo 8 descreve a implementação das operações estruturais para o GED, que são executadas sobre a estrutura interna de representação dos diagramas, a árvore sintática (AS). Descreve-se a geração de menus, a partir da sintaxe, para as operações de inclusão e remoção.

O capítulo 9 comenta as operações não-estruturais. Este capítulo praticamente descreve uma interface entre a linguagem de edição de léxicos apresentada no capítulo 3 e as operações estruturais apresentadas no capítulo 8.

O capítulo 10 apresenta os principais procedimentos usados para decompor um documento diagramático em páginas. O documento é visto como uma lista de páginas.

O capítulo 11 resume os trabalhos relacionados ao tema. Os trabalhos mais importantes são discutidos diretamente no capítulo que introduz o tema específico; estes trabalhos são apenas citados no capítulo 11.

Na conclusão são citadas as futuras extensões ao GED e são comentandos os principais resultados.

2 ALGUNS CONCEITOS DE LINGUAGENS FORMAIS

O objetivo deste capítulo é introduzir os conceitos relacionados com gramáticas livres de contexto (GLCs), gramáticas de atributos (GA) e árvores sintáticas (ASs).

Estes conceitos são introduzidos a partir de um exemplo de uma minilinguagem para expressões, adaptada de [REP 87], como mostra a figura 2.1.

```
1. Let  a = 1 in
2.    Let b = (2 + a) in
3.       Let a = a + a in
4.             (( a + b ) + 1)
   VALUE = 6
```

figura 2.1: um exemplo de expressão para a minilinguagem

2.1 Gramática Livre de Contexto (GLC)

A figura 2.2 apresenta uma gramática livre de contexto (GLC) descrevendo a minilinguagem de expressões com os operadores de soma (+) e divisão (/), e uma cláusula Let que associa o valor de uma expressão a um identificador.

```
Exp --> 'Let' !ID '=' Exp 'in' Exp
Exp --> Exp '+' Exp
Exp --> Exp '/' Exp
Exp --> '(' Exp ')'
Exp --> ID
Exp --> INTEGER
```

figura 2.2: uma GLC para a minilinguagem de expressões

Os símbolos terminais são representados por letras maiúsculas (ID, INTEGER), as palavras reservadas são delimitadas por apóstrofes e o símbolo **Exp** é não-terminal.

2.1.1 Árvore sintática (AS)

Uma árvore sintática ("parse-tree") representa o caminho de derivação percorrido no reconhecimento de uma sentença. A raiz da árvore está associada ao símbolo inicial da gramática, as folhas (fronteira) representam os símbolos léxicos da sentença e os nodos internos estão associados a não-terminais da GLC.

Cada produção da GLC gera uma subárvore. Por exemplo a produção **Exp --> Exp + Exp** gera uma subárvore onde **Exp** (lado esquerdo do símbolo **-->**) é pai de **Exp, +, Exp**, figura 2.3 (a).

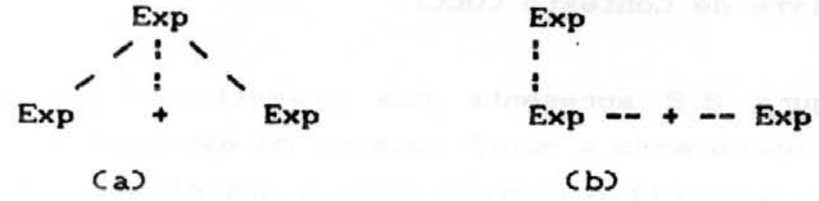


figura 2.3: (a) e (b) representações de ASs

Nos exemplos apresentados neste trabalho utilizam-se árvores representadas por nodos binários, i.e., um nodo pai se liga com o filho mais à esquerda e cada irmão da esquerda se liga com o irmão da direita, figura 2.3 (b). Cada nodo se liga com no máximo 2 outros nodos: um filho e um irmão.

2.1.2 Árvore sintática abstrata

Na literatura relacionada com editores orientados por estrutura, para se fazer referência à estrutura interna de representação, é comum o uso do termo árvore sintática abstrata - ASA ("abstract syntax tree", ou "syntax tree" [AHO 86]). Uma ASA possui a mesma estrutura que uma AS, porém, na primeira, os detalhes cosméticos não são representados. Estes detalhes compreendem a pontuação, palavras reservadas e derivações intermediárias de produções; figura 2.4. As informações sobre os detalhes cosméticos podem ser buscadas nas produções, figura 2.4 (a).

(a) Produção:

```
ifStmt --> "if" Exp "then" Stmt "else" Stmt ";"
```

(b) AS:

```
 ifStmt
 !
"if" -- Exp -- "then" -- Stmt -- "else" -- stmt--";"
```

c) ASA:

```
 ifStmt
 !
Exp -- Stmt -- Stmt
```

figura 2.4: a AS (b) representada como ASA em (c).

Nas GLCs para linguagens diagramáticas o açúcar sintático ("caixinhas", "setas", padrões de cores, etc.) não é especificado. Estas informações devem ser especificadas no nível léxico. Assim, no contexto deste trabalho, não existe a necessidade de diferenciar AS de ASA; no decorrer do trabalho a árvore será referida por AS.

2.2 Gramática de atributos (GA)

Para introduzir os conceitos relacionados com GA são especificados os aspectos dependentes de contexto da minilinguagem: (a) a avaliação do valor da expressão; (b) a indicação de divisão por zero (na avaliação retorna-se zero para a subexpressão) e de nomes não definidos (na avaliação retorna-se zero para o nome).

Para definir a minilinguagem é introduzida uma tabela de símbolos (TS) que armazena os nomes associados aos seus valores. Esta TS é representada por uma lista ordenada de tuplas **Id-Valor**; a figura 2.5 mostra a linha onde os valores da tupla são incluídos na lista.

```
1. Let  a = 1 in
2.    Let b = (2 + a) in
3.       Let a = a + a in
4.          (( a + b ) + 1 )
   VALUE = 6
```

Tabela de símbolos

Linha	Id-Valor	Lista(Id-Valor)
1	<a, 1>	{<a,1>}
2	<b, 3>	{<a,1>, <b,3>}
3	<a, 2>	{<a,1>, <b,3>, <a,2>}

figura 2.5: texto e tabela de símbolos associada

A minilinguagem utilizada nos exemplos apresenta dois conceitos que são comuns para as linguagens de programação do tipo bloco estruturadas: o conceito de ordem de escrita e de escopo.

No texto da figura 2.5 mostra-se o conceito de ordem de escrita: a variável "a" está definida duas vezes; na avaliação da expressão busca-se a última ocorrência,

antes do uso, no sentido da ordem de escrita. A pesquisa do valor deve começar do último elemento da lista de tuplas para o primeiro.

Na figura 2.6 mostra-se o conceito de escopo (contexto): a variável "c", definida na linha 2, é utilizada no escopo da linha 2.1. No escopo da linha 4 a variável "c" não está definida ("undefined"). Cada comando Let introduz uma nova variável, válida no contexto dado pela expressão que segue a palavra reservada "in"; ver por exemplo a linha 2 da tabela de símbolo da figura 2.6. Na avaliação da expressão busca-se no contexto adequado o valor do último nome declarado.

```
1.    Let  a = 1
2.    in   Let b = Let c = (2 + a)
2.1                in  (c + 2)
3.         in  Let a = a + a
4.             in  (( a + b ) + c <---Undefined )
    VALUE = 7
```

Tabela de símbolos

Linha	Id-Valor	Lista(Id-Valor)	
1	<a, 1>	{<a,1>}	
2		{<a,1>}	para "Let c=(2+a) in (c+2)"
2	<c, 3>	{<a,1>, <c,3>}	para "(c+2)"
2.1		{<a,1>, <c,3>}	
2	<b, 5>	{<a,1>, <b,5>}	para "Let a = a+a ..."
3	<a, 2>	{<a,1>, <b,5>, <a,2>}	

figura 2.6: texto e tabela de símbolos com escopo

Uma GA é uma gramática livre de contexto (GLC) estendida por um conjunto de equações semânticas. As equações, associadas às produções, são definidas em função dos atributos associados aos símbolos da GLC.

A especificação de uma linguagem consiste na

definição das equações semânticas (figura 2.7) e das funções utilizadas na representação da TS (figura 2.8).

```
( inh ENV env   : Exp;
  syn INT value : Exp;
  ext INT value : !INTEGER;
  ext STR name  : !ID; )

Gr    --> Exp
   ( Exp.env = emptyList; )

Exp1 --> 'Let' !ID '=' Exp2 in' Exp3
  ( Exp3.env   =  List(<!ID.name, Exp2.value>, Exp1.env);
    Exp2.env   =  Exp1.env;
    Exp1.value =  Exp3.value; )

Exp1 --> Exp2 '+' Exp3
( Exp3.env   =  Exp1.env;
  Exp2.env   =  Exp1.env;
  Exp1.value =  Exp2.value + Exp3.value; )

Exp1 --> Exp2 '/' Exp3
( Local STR error;
  error  = if Exp3.value = 0 then "<---Division by zero"
                             else "";
  Exp3.env   =  Exp1.env;
  Exp2.env   =  Exp1.env;
  Exp1.value = if Exp3.value = 0
                  then 0
                  else Exp2.value / Exp2.value;)

Exp --> !ID
( Local STR error;
  error  = if not LookUpId(!ID.name, Exp.env)
               then "<---Undefined"
               else "";
  Exp.value = LookUpValue(!ID.name, Exp.env); )

Exp --> !INTEGER
( Exp.value = !INTEGER.value;)
```

figura 2.7: GA para a minilinguagem de expressões

Uma especificação em LDE [ESP 89b] inicia com a declaração dos atributos, seguida pela lista de produções associadas a equações semânticas. Cada equação semântica

define o valor de um atributo em função dos atributos visíveis na produção.

Os símbolos terminais (!ID e !INTEGER) na LDE são prefixados pelo operador "!" para diferenciá-los dos símbolos não-terminais (Gr, Exp).

Associado ao símbolo terminal !ID foi declarado o atributo **name**, que representa a cadeia de caracteres do identificador. Para o terminal !INTEGER foi declarado o atributo **value**, que representa o valor calculado a partir da cadeia de dígitos. Para o não-terminal **Exp** foram definidos os atributos **Env**, que representa a TS, e o atributo **value**, que representa o valor da expressão. Os tipos (inh, ext, synt, local) de cada atributos são discutidos no próximo item.

```
( /* definição para ser anexada a descrição-LDE */
Id       = String;
Value    = Integer;
IdValue = Id x Value;

type emptyList   :                    --> List;
type List        : IdValue x List     --> List;
type LookUpId    : Id x List          --> Boolean;
type LookUpValue : Id x List          --> Value;

/*
  LookUpId(a,List(<a,v>,l)) = true;
  LookUpId(a, emptyList   ) = false;
  LookUpId(a,List(<b,v>,l)) = LookUpId(a,L);

  LookUpValue(a,List(<a,v>,l)) = v;
  LookUpValue(a,emptyList    ) = undefined;
  LookUpValue(a,List(<b,v>,l)) = LookUpValue(a,l);

   -- undefined é valor nulo  para GA -- */ )
```

figura 2.8: definição da tabela de símbolos

O cálculo do valor da expressão consiste numa

avaliação das equações a partir do atributos associados aos símbolos terminais. Os atributos **name e value** associados aos elementos terminais são avaliados pelo reconhecedor léxico.

A figura 2.8 descreve os cabeçalhos das funções que implementam a TS. As descrições das figuras 2.7 e 2.8 são submetidas ao mecanismo de GA. O comentário, delimitado pelo par /* */, da figura 2.8 descreve uma possível implementação para as funções; estas funções devem ser codificadas na linguagem Pascal [ESP 89b].

As funções **emptyList** e **List** implementam, respectivamente, as operações de inicialização e de inserção de elementos na lista. A função **LookUpId(,)** retorna **true** se o identificador está definido e retorna **false** caso contrário. A função **LooKUpValue(,)** retorna o valor do identificador ou undefined.

Conceitos

Na produção listada abaixo, da figura 2.7, pode-se identificar:

```
1. Exp1 --> 'Let' !ID '=' Exp2 'in' Exp3
1.1 ( Exp3.env   =  List(<!ID.name, Exp2.value>, Exp1.env);
1.2   Exp2.env   =  Exp1.env;
1.3   Exp1.value =  Exp3.value; )
```

Produção, linha 1.

Equações, linhas 1.1, 1.2, 1.3.

Atributos sintetizados, linha 1.3 Exp1.value.

Atributos herdados, linha 1.1 Exp3.env, e pela 1.2 Exp2.env.

Grafo de dependências, linha 1.1 Exp_3.env depende de Exp_1.env
Exp_3.env depende de !ID.Name
Exp_3.env depende de Exp_2.value

Os conceitos exemplificados acima são brevemente descritos nos ítens que seguem.

2.2.1 Atributos

Um atributo representa um valor semântico de um símbolo da GLC. Por exemplo, o atributo **value** associado a produção **Exp** representa o valor de cada subexpressão definida pela fronteira da subárvore.

Numa GA [DER 88] existem duas classes disjuntas de atributos: os sintetizados e os herdados. Os valores de atributos **sintetizados** sobem na estrutura da AS, equanto que os valores dos atributos **herdados** descem na estrutura da AS.

```
1. Exp1 --> 'Let' !ID '=' Exp2 'in' Exp3
1.1 ( Exp3.env   =  List(<!ID.name, Exp2.value>, Exp1.env);
1.2   Exp2.env   =  Exp1.env;
1.3   Exp1.value =  Exp3.value; )
```

Na árvore sintática, o símbolo do lado esquerdo da seta (-->) da produção é pai dos símbolos que estão ao lado direito da seta. Assim, um atributo definido por uma equação (no lado esquerdo do sinal de igualdade da equação) é **sintetizado** (sobe), se está associado ao símbolo não-terminal do lado esquerdo da seta (Exp_1); e, é **herdado** se está associado a um dos símbolos que estão ao lado direito da seta (e.g. Exp_2, Exp_3).

Atributos do tipo Local e External

Os atributos **Local** e **External** definem uma extensão à GA pura, permitindo a entrada e a saída de valores do fluxo da gramática de atributos.

Os atributos externos permitem a entrada de valores para o fluxo da GA. Por exemplo os valores externos dos atributos !ID.**name** e !INTEGER.**value**, da figura 2.7, são trazidos para dentro do fluxo da GA. Não existem equações que definem seus valores. Os valores são calculados pelo reconhecedor léxico.

Atributos externos também podem estar associados a símbolos não-terminais. Por exemplo no nodo raiz (associado a símbolo inicial da gramática) não se pode ter atributos do tipo herdado, pois este nodo não tem pai. No entanto, pode-se ter um atributo do tipo **ext** que, por exemplo, guarda o valor das coordenadas da janela corrente de edição.

O que caracteriza um atributo externo é a inexistência de uma equação que o defina. Os atributos externos são visíveis nas produções que usam os símbolos a que estão associados.

Tradicionalmente os símbolos terminais não possuem atributos. Por exemplo, o atributo !INTEGER.**value** seria definido como uma função aplicada sobre o símbolo terminal INTEGER ("Exp.value := fc(INTEGER);" ver [AHO 86]). Da mesma forma não existem atributos externos associados aos símbolos não-terminais; estes seriam definidos pelas, também chamadas, **ações semânticas** (ver [AHO 86]). No entanto, os algoritmos de avaliação incremental de GA são ativados a partir de um conjunto de valores de atributos inconsistentes e não de funções/ações inconsistentes. Na LDE, os atributos

externos permitem a entrada de valores externos ao fluxo da GA sem complicar o algoritmo de avaliação com o uso de artifícios: quando um atributo externo é modificado, basta informar para o avaliador para que seja executada a reavaliação dos atributos afetados (ver [ESP 89b]).

Na terceira alternativa da produção Exp na figura 2.7 existe a declaração do atributo error do tipo "local". Este atributo é utilizado para armazenar o valor de uma mensagem de erro que é coletada a partir do fluxo de valores da GA. Esta mensagem é exibida intercalada ao texto, figura 2.6.

Os atributos do tipo local não podem ser usados em expressões que definem outros atributos; normalmente armazenam as mensagens de erro que são exibidas na formatação do documento.

A declaração de um atributo local é válida somente para a alternativa da produção em que foi declarado, diferentemente das declarações de atributos do tipo **syn, inh, ext,** (início da figura 2.7) que são válidas para todas alternativas de um símbolo gramatical.

A construção local é devida à linguagem baseada em GA utilizada pelo Synthesizer Generator [REP 87].

2.2.2 Grafo de Dependências

Numa especificação em GA não se define a ordem de avaliação das equações; por isso o formalismo GA também é classificado como uma linguagem denotacional [REP 87]. No

momento da avaliação é necessário descobrir a ordem correta de avaliação das equações.

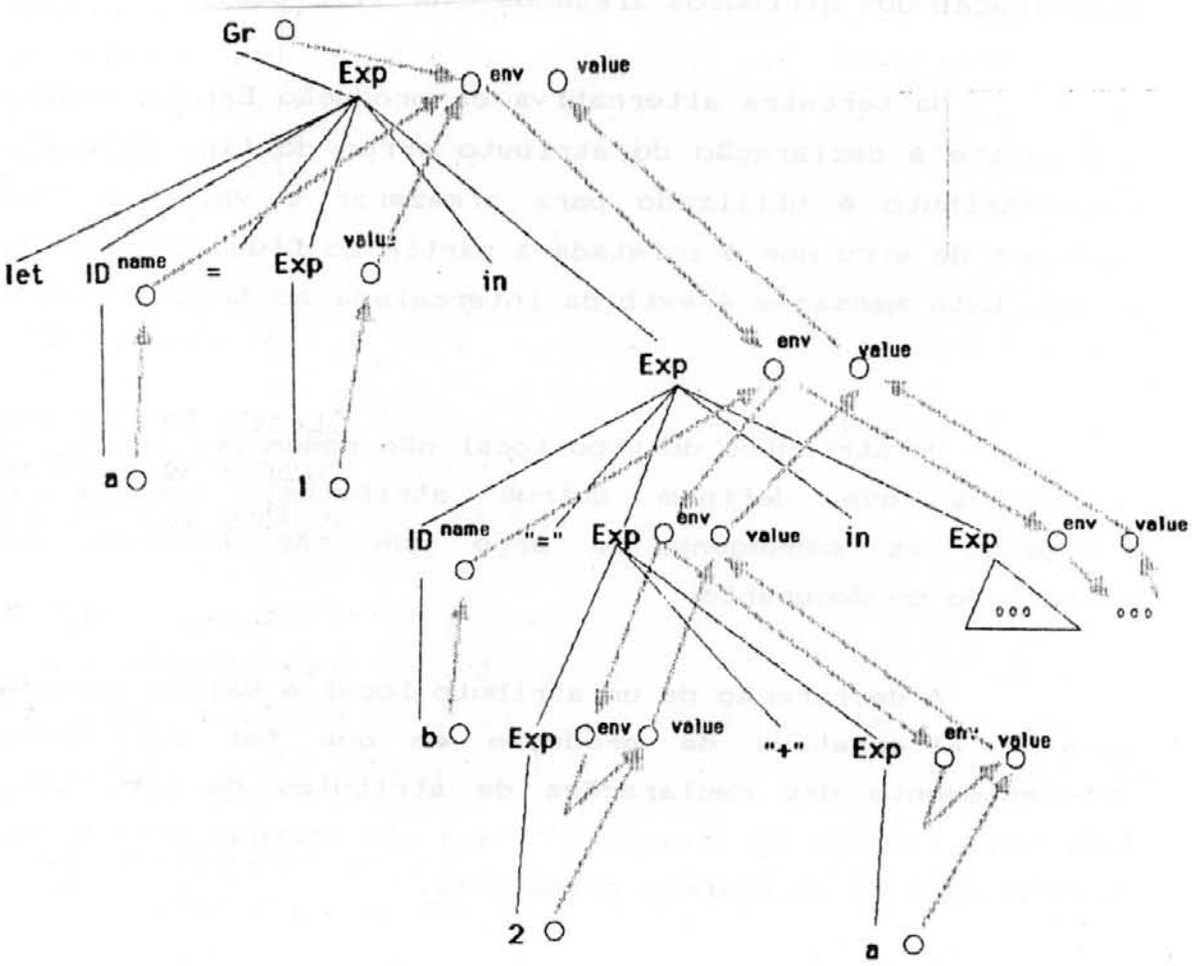


figura 2.9: grafo de dependências para a expressão: "Let a=1 in Let b=2+a in ... "

A avaliação de uma equação consiste em calcular o valor da expressão do lado direito do sinal de igualdade e atribuí-lo ao atributo à esquerda do sinal de igualdade da equação. Uma equação somente pode ser calculada se os valores dos atributos referenciados no lado direito do sinal

de igualdade já estão definidos; a partir deste fato pode-se descobrir qual é a ordem de avaliação das equações.

O grafo de dependência representa o caminho de computação das equações associadas às produções. É obtido pela construção de um grafo a partir dos subgrafos associados a cada produção. Um subgrafo é formado por todos os arcos do tipo depende de definidos pelas equações associadas às produções:

```
1. Exp1 --> 'Let' !ID '=' Exp2 'in' Exp3
1.1 { Exp3.env   =  List(<!ID.name, Exp2.value>, Exp1.env);}
```

Grafo de dependências, linha 1.1 Exp3.env depende de Exp1.env
Exp3.env depende de !ID.Name
Exp3.env depende de Exp2.value

Numa expressão da forma "a_0 := f(a_1,a_2,...,a_n);" criam-se os sub-arcos a_0 depende de a_i, onde $1 \leq i \leq n$ e a_0, a_i são atributos locais à produção da GA (i.e. para se calcular o valor a_0 necessita-se dos valores a_1 ... a_n).

A figura 2.9 ilustra o grafo de dependências para um fragmento de texto da minilinguagem de expressões.

A avaliação se processa a partir das extremidades iniciais do grafo de dependências, onde existem valores dos atributos externos; no exemplo da figura 2.7 as extremidades são os atributos !ID.name e !INTEGER.value. A avaliação termina quando todos os valores dos atributos são consistentes, i.e. quando todas as equações (como igualdades) são satisfeitas.

Uma mesma equação é definida para todos os nodos

da AS associados à produção, porém, em cada nodo a equação é avaliada com valores distintos.

2.3 GA estendida por tabelas relacionais

A TS da figura 2.8 foi definida por funções que implementam listas. Em [HOR 86] é proposta uma forma alternativa e complementar às funções/listas para a construção de TSs: a utilização de um sistema de álgebra relacional [DAT 84] acoplado ao mecanismo de GA. Com esta proposta um documento é representado internamente como uma AS associada a tabelas relacionais, combinando atributos em tuplas de relações, figura 2.10.

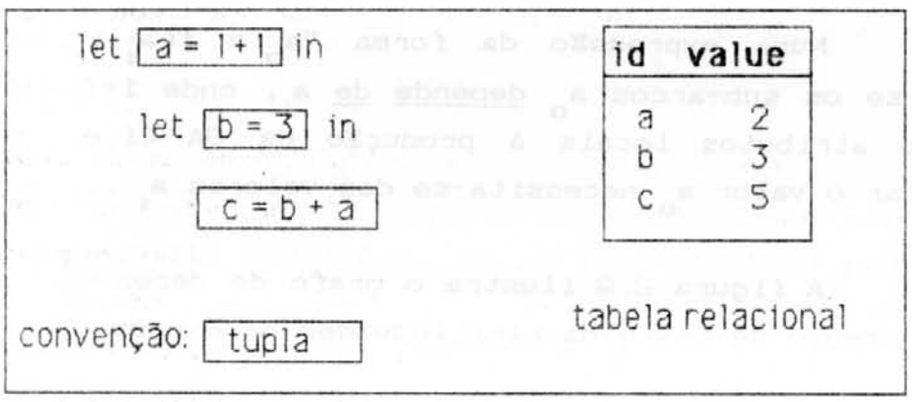


figura 2.10: tabela de símbolos representada por tabela relacional

As tuplas são associadas a produções para armazenar valores de atributos visíveis na produção (na verdade podem conter também informações estruturais, tais como: símbolos terminais, ponteiros sobre AS, etc.). Quando a estrutura da produção associada a uma tupla é criada,

cria-se também a tupla da relação. Quando é removida remove-se também a tupla da relação.

Num mecanismo puro de GA um atributo só pode ser definido em função de seus atributos vizinhos; com esta extensão um atributo pode ser definido também em função de atributos distantes na AS, representados nas relações globais. As relações mantêm agrupados os atributos que estão dispersos pela extensão da AS.

O modelo relacional para a implementação de TS apresenta três limitações. A primeira se deve ao fato do modelo relacional trabalhar com conjuntos (uma coleção de valores não duplicados e não ordenados). Esta limitação se refere ao processamento dependente da ordem de escrita (fecho transitivo da GLC - ordem sobre a estrutura da árvore). O conceito de ordem de escrita foi exemplificado no item 2.2.

O segundo problema é a inexistência de operadores aritméticos no modelo relacional. O terceiro, se refere ao processamente dependente da ordenação das tuplas (por exemplo, ordem lexicográfica, ordem dos números reais, naturais, etc). As implementações de sistemas relacionais já incluem extensões para suprir este terceiro problema (ver [DAT 84]).

Uma associação entre GA e álgebra relacional pode apresentar as seguintes vantagens:

i) A computação fora do escopo do modelo relacional é executada pelas equações da GA [HOR 86]. É o caso das operações aritméticas e do problema de processamento dependente de ordem de escrita.

```
{ Table TABSYMB{ Id:String; Value:Integer;};
  syn INT value :Exp;
  ext STR name  :!ID;
  ext INT value :!INTEGER }

Gr    --> Exp

Exp1 --> 'Let' !ID '=' Exp2 'in' Exp3
  { <!ID.name, Exp2.value> in TABSYMB;
    Exp1.value = Exp3.value; }

Exp1 --> Exp2 '+' Exp3
  { Exp1.value =  Exp2.value + Exp3.value; }

Exp1 --> Exp2 '/' Exp3
{ local STR error;
 error  = if Exp3.value = 0 then "<---Division by zero"
                            else "";
  Exp1.value = if Exp3.value = 0
               then 0
               else Exp2.value / Exp2.value; }

Exp1 --> !ID
{ local STR error1, error2;
 error1  = if !ID.name  ∉  TABSYMB[ Id ]
           then "<---Undefined"
           else "";
  error2  = if Card(TABSYMB where TAMSYMB.Id = ID.Name )
           > 1 then "<---Identifier duplicated"
           else "";
  Exp1.value = if !ID.name  ∈  TABSYMB[ Id ]
                then  (TABSYMB[ Value ]
                       where TABSYMB.Id  = ID.name)
                else 0;
Exp --> !INTEGER
{ Exp.value = !INTEGER.value;}
```

OBS: A função **Card** retorna a cardinalidade da tabela.

figura 2.11: uma GA para a minilinguagem de expressões

ii) O mecanismo de GA mantém, dinamicamente, consistentes as informações representadas nas tuplas de relações. Esta facilidade possibilita a integração de ferramentas através

de um BD. Por exemplo, pode-se fazer referências cruzadas em TSs de diferentes linguagens.

2.3.1 Exemplo de uso de tabelas relacionais associadas a GA

Para ilustrar uso de tabelas reescreve-se a especificação da minilinguagem de expressões com a TS representada por tabelas relacionais, figura 2.10-2.11.

A expressão <!ID.name, Exp2.value> in TABSYMB, figura 2.11, denota que os atributos **!ID.name** e **Exp2.value** são inseridos na TABSYMB toda vez que um nodo, produção Exp, é inserido na AS. Quando um valor de um destes atributos é atualizado, automaticamente atualiza-se o valor da tupla na tabela.

Conceitualmente esta descrição difere da figura 2.7, pois foram perdidos os conceitos de ordem de escrita e de escopo.

Na substituição direta da tabela de símbolos List, implementada como uma lista, por um conjunto (tabela relacional), perdeu-se o conceito de ordem de escrita; não existe mais a ordem, para os símbolos, de aparecer antes no texto.

Na figura 2.7 é introduzido um novo contexto (o conteúdo da lista é diferente) para cada cláusula Let. O contexto é representado por listas aninhadas. Na substituição destas listas aninhadas por uma única tabela relacional perde-se o conceito do escopo. Pode-se anexar campos na tabela relacional para representar os conceitos de

ordem e de escopo, porém o cálculo dos valores destes campos é a própria especificação descrita na figura 2.7.

A falta da ordem de escrita e escopo deixa a minilinguagem de expressões ambígua. Por exemplo, se um identificador "a" é defindo duas vezes, não se sabe qual é a última definição. Para tirar a ambigüidade, sem complicar o problema, restringiu-se o uso dos nomes: um nome deve ser único em todo o texto da expressão. A equação que gera a mensagem "**indentifier duplicated**" define esta restrição.

Este exemplo mostrou que as tabelas relacionais apresentam algumas limitações para serem utilizadas como tabelas de símbolos para linguagens que possuem os conceitos de ordem de escrita e escopo.

Apesar destas limitações, as tabelas relacionais associadas a GA são utilizadas como tabelas de símbolos para a classe das linguagens diagramáticas que são (principalmente) baseadas no conceito de grafo (nodos e arcos) e não possuem o conceito de ordem de escrita. Também, como será ilustrado nos próximos capítulos, as tabelas relacionais são vistas como um mecanismo que permite o compartilhamento de informações entre distintas notações diagramáticas.

2.4 Textos relacionados com o tema GA

O tema GA é recente. O formalismo GA, apenas com atributos sintetizados, surgiu no início da década de 60 (ver [AHO 86]). O modelo de GA, incluindo atributos herdados, utilizado para descrição da semântica de linguagem de programação foi introduzido por [KNU 68].

Neste trabalho são utilizadas as idéias de avaliação incremental de GA (introduzida por [REP 83]) e extensões relacionadas com a implementação de tabelas de símbolos [HOR 86], [REP 87], [HOO 87].

A computação incremental consiste em, após uma alteração do documento fonte, reavaliar somente as equações que realmente foram afetadas pela modificação. Reps [REP 83] definiu, para uma sub-classe das GAs, um algoritmo que considera a melhor ordem de reavaliação das equações após uma operação de edição.

Horwitz [HOR 86] introduziu as tabelas relacionais como TSs. Hood [HOO 87] discute tópicos relacionados com a otimização de avaliadores de GA e a implementação de TSs.

Aho, no texto clássico de compiladores [AHO 86], apresenta uma boa introdução ao tema GA. A fonte de consultas mais importante é o trabalho [DER 88] onde são classificadas mais de 600 referências sobre o tema GA: é apresentada uma revisão sobre os conceitos relacionados a GA e uma revisão dos principais sistemas baseados em GA (tradutores, compiladores, editores orientados por estrutura, etc.; mais de trinta destes sistemas).

3 LINGUAGENS DIAGRAMÁTICAS

Neste capítulo, as linguagens diagramáticas são classificadas em dois tipos pela forma da estrutura interna: árvore e grafo. Estes dois tipos de linguagens diagramáticas divergem um pouco a nível de especificação e, conseqüentemente, a nível de implementação. Alguns exemplos ilustram esta classificação.

As notações exemplificadas neste capítulo são descritas em LDE nos próximos capítulos. As facilidades a nível de metalinguagem são gradualmente introduzidas nas especificações.

No final deste capítulo apresenta-se o formalismo para descrever o nível léxico para as notações diagramáticas.

3.1 Classificação das linguagens diagramáticas

Do ponto de vista de representação interna existem dois tipos de linguagens diagramáticas: estrutura interna em forma de árvore (sem nodos compartilhados) e em forma de grafo (com nodos compartilhados).

O diagrama de Nassi-Schneiderman (figura 3.1) possui a representação interna hierárquica em forma de árvore, i.e. pode ser representado na estrutura de nodos e arcos sem nodos compartilhados. Esta estrutura hierárquica também é um grafo, porém será chamada apenas de árvore.

O diagrama de fluxo de dados (DFD) da figura 3.2 possui uma estrutura de grafo: possui nodos compartilhados.

O nodo x é compartilhado por três arcos.

Outro exemplo de linguagem diagramática do tipo grafo é a variante do diagrama de E-R apresentado em [MAR 87], figura 3.3. Neste diagrama, cada entidade pode ser compartilhada por diferentes relações.

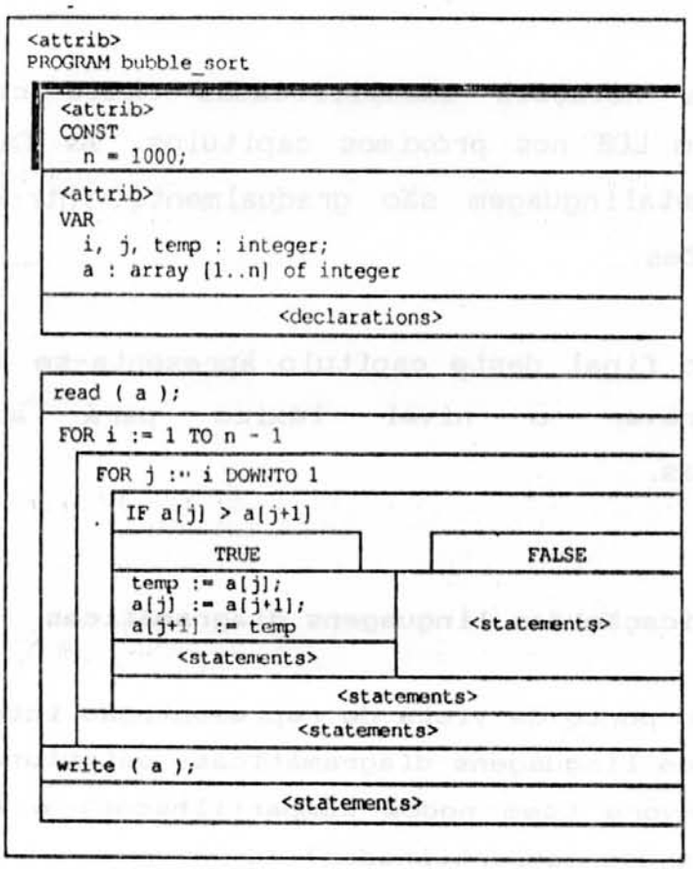


figura 3.1: Diagrama de Nassi-Schneiderman [HAL 88]

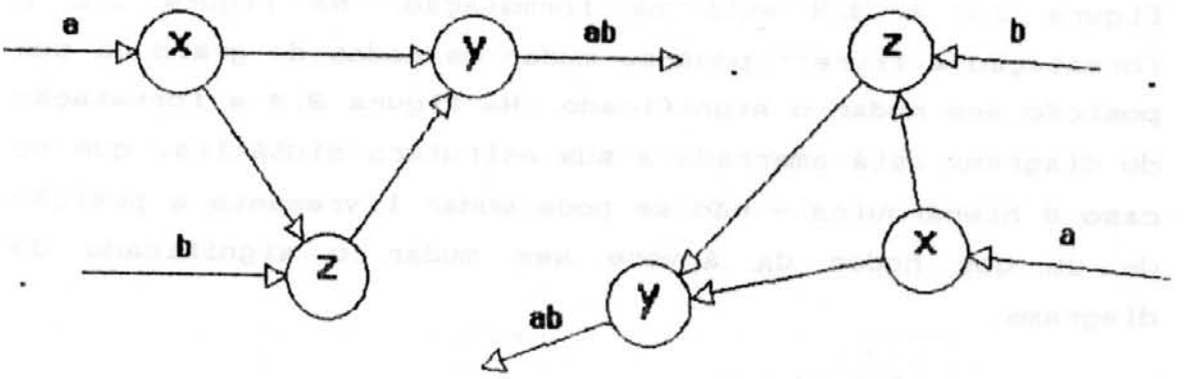


figura 3.2: linguagem do tipo grafo com formatação livre; duas representações (a) e (b) para um mesmo diagrama de fluxo de dados (DFD)

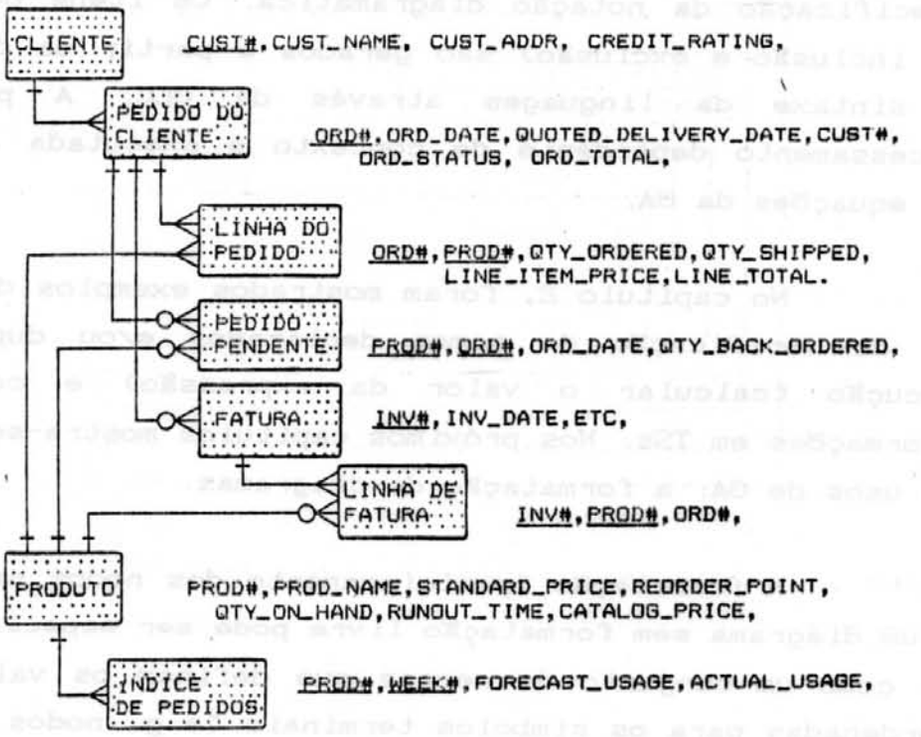


figura 3.3: linguagem tipo grafo com formatação hierárquica; variante de E-R [MAR 87]

A diferença entre as notações do tipo grafo da figura 3.2 e 3.3 está na formatação. Na figura 3.2 a formatação é livre - pode-se mudar os nodos do grafo da sua posição sem mudar o significado. Na figura 3.3 a formatação do diagrama está amarrada a sua estrutura sintática, que no caso é hierárquica - não se pode mudar livremente a posição de um dos nodos da árvore sem mudar o significado do diagrama.

3.2 Gramática de atributos para diagramas

O objetivo de descrever linguagens diagramáticas em GA é a possibilidade de geração de um editor a partir da especificação da notação diagramática. Os ítens dos menus (de inclusão e exclusão) são gerados a partir da descrição da sintaxe da linguagem através da GLC. A parte de processamento dependente de contexto é executada a partir das equações da GA.

No capítulo 2, foram mostrados exemplos de uso de GA: de verificação de nomes declarados e/ou duplicados, execução (calcular o valor da expressão) e coleta de informações em TSs. Nos próximos capítulos mostra-se mais um dos usos de GA: a formatação de diagramas.

A formatação (posicionamento dos nodos na página) de um diagrama sem formatação livre pode ser especificada em GA, como um conjunto de regras que definem os valores das coordenadas para os símbolos terminais (e.g. nodos e arcos) que são exibidos na tela.

A classificação das linguagens tem várias implicações na contrução do GED.

As linguagens diagramáticas baseadas em estrutura de árvore são próximas às linguagens de programação. Por exemplo a versão de diagramas de Nassi-Shneiderman, descrita em [HAL 88], possui uma estrutura similar a da linguagem de programação Pascal. Para este tipo de linguagem a construção do editor tem por base a tecnologia desenvolvida para as linguagens textuais: a sintaxe (estrutura de árvore) é descrita por produções de uma GLC tradicional e os aspectos semânticos por regras de um GA.

Para as linguagens com estrutura em grafo é necessário estender a metalinguagem LDE, a nível de sintaxe, para permitir a especificação de nodos compartilhados. No capítulo 5 é discutida uma extensão, sobre a GLC, que permite expressar arcos e nodos.

Na especificação da semântica de muitas linguagens deve-se considerar os conceitos de **ordem de escrita** e **escopo**. No entanto, em muitas das linguagens do tipo grafo não existe o conceito de ordem de escrita; por exemplo, as com formatação livre. As TSs para estas liguagens podem facilmente ser representadas como tabelas relacionais.

Níveis da gramática de atributos

Na especificação de uma linguagem, usualmente, são considerados três níveis:

i) o **nível léxico**: neste nível são definidos os elementos léxicos terminais da descrição sintática. Por exemplo para diagramas os léxicos são as "caixinhas" do diagrama. Para linguagens textuais neste nível são definidos os símbolos terminais: palavras reservadas, valores numéricos, nomes de variáveis, etc.

ii) o **nível sintático**: neste nível são definidas todas as combinações válidas dos elementos léxicos. É usualmente definido por uma GLC. Este nível especifica a estrutura da linguagem. Por exemplo, para diagrama do tipo grafo são especificados os nodos e os arcos.

iii) o **nível semântico**: neste nível são definidos todos os detalhes da linguagem que não são especificáveis adequadamente no nível sintático; é feita a verificão de nomes declarados, verificação de tipos; pode ser gerado código intermediário; são definidas TSs; pode ser definida a formatação; etc.

No próximo item é detalhado o nível léxico para linguagens diagramáticas. Os níveis sintático e semântico são discutidos nos próximos capítulos.

3.3 Nível léxico

Um elemento léxico ("caixinha", "bolha", segmentos de reta, etc.) para uma linguagem diagramática pode ser visto como um objeto gráfico composto por um conjunto de primitivas geométricas. Existem duas abordagens para a especificação dos elementos léxicos: (a) a descrição textual baseda em regras de formação; (b) a descrição de forma interativa do elemento pela escolha de primitivas geométricas de um cardápio.

Nos ítens que seguem comentam-se estas abordagens.

3.3.1 Uma linguagem do tipo "constraint-based"

[BAR 87] descreve um sistema gráfico interativo, para a modelagem de sólidos, usando uma linguagem do tipo "constraint-based". Esta linguagem também pode ser utilizada para a representação de objetos gráficos em duas dimensões, como os elementos léxicos das notações diagramáticas.

Numa linguagem deste tipo os elementos léxicos, como objetos compostos de primitivas geométricas (pontos, retas, polígonos, etc.), são descritos por duas construções: a classe (CLASS) e a restrição (CONSTRAINT).

A classe armazena os valores de um conjunto de variáveis que definem um objeto geométrico primitivo. As variáveis especificam as características do objeto como por exemplo, o tamanho e localização dos objetos. Uma restrição define um conjunto de equações que devem ser satisfeitas quando aplicadas às variáveis.

A figura 3.4 mostra um exemplo, adaptado de [BAR 87], de especificação de um retângulo combinando-se restrições e classes.

Na figura 3.4 (a) é definida uma restrição, PtId, que tem como parâmetro dois pontos (todo o texto delimitado por aspas é comentário). Esta restrição define a igualdade de dois pontos - a restrição é satisfeita, somente se os dois pontos passados como parâmetros são iguais.

```
a) "Constraint definition for point identification"
CONSTRAINT PtId "Identify two points" ON
  p1 : point "a point to make same as another";
  p2 : point "the point to make the first one the same as";
CONSTRAINED BY
  p1.x = p2.x;
  p1.y = p2.y;
END

b) "Class definition for line segments"
CLASS lineSeg "2-D line segment" IS
  p1 : point = ( x=1, y=1); "an endpoint"
  p2 : point = ( x=2, y=4); "another endpoint"
CONSTRAINED BY
   ((p1.x - p2.x)**2 + (p1.y - p2.y)**2) > 3**2;
END

c) "Class definition for rectangle"
CONSTRAINT lineHor "horizontal line" ON
  line : lineSeg;
CONSTRAINED BY
  line.p1.y = line.p2.y; "the same y value for p1 and p2"
END

CONSTRAINT lineVer "vertical line" ON
  line : lineSeg;
CONSTRAINED BY
  line.p1.x = line.p2.x; "the same x value for p1 and p2"
END

CLASS rectangle IS
  s1 : lineSeg = (p1=(x=1,y=1), p2(x=1,y=3)) "side #1";
  s2 : lineSeg = (p1=(x=1,y=3), p2(x=3,y=3)) "side #2";
  s3 : lineSeg = (p1=(x=3,y=3), p2(x=3,y=1)) "side #3";
  s4 : lineSeg = (p1=(x=3,y=1), p2(x=1,y=1)) "side #4";
CONSTRAINED BY
  PtId(s1.p2, s2.p1);
  PtId(s2.p2, s3.p1);
  PtId(s3.p2, s4.p1);
  PtId(s4.p2, s1,p1);
  LineVer(s1);LineHor(s2);LineVer(s3);lineHor(s4);
END
```

figura 3.4: a especificação de um retângulo

Na figura 3.4 (b) a classe **lineseg** define uma segmento de reta. Esta classe é definida em função de duas variáveis do tipo ponto, que recebem um valor inicial ("default"). Uma restrição garante que o tamanho mínimo de um segmento é maior que 3 unidades. Esta classe pode assumir quaisquer valores, desde que a restrição seja satisfeita.

Na figura 3.4 (c) é definida a figura geométrica retângulo. Um retângulo é formado por quatro retas com os extremos comums, adequadamente combinados. Para garantir que dois lados sejam verticais e dois horizontais foram definidas as restrições **lineVer** e **lineHor**. Esta classe **rectangle**, também, faz uso da classe **lineseg** e da restrição **PtId**.

Na exibição do retângulo assume-se que a classe **lineSeg** possui uma representação concreta pré-definida; o retângulo é exibido a partir das quatro retas.

```
CLASS rectangle IS
  rp1 : point = (x=1,y=1);
  rp2 : point = (x=3,y=3);
CONSTRAINED BY
   ((rp1.x - rp2.x)**2 + (rp1.y - rp2.y)**2) > 3**2;
SHOWN AS
  s1 : lineSeg = (p1=(rp1.x, rp1.y), p2=(rp1.x, rp2.y));
  s2 : lineSeg = (p1=(rp1.x, rp2.y), p2=(rp2.x, rp2.y));
  s3 : lineSeg = (p1=(rp2.x, rp2.y), p2=(rp1.x, rp2.y));
  s4 : lineSeg = (p2=(rp1.x, rp2.y), p1=(rp1.x, rp1.y));
END
```

figura 3.5: uma descrição alternativa para o retângulo

Uma forma alternativa para definir um retângulo é com uso de apenas dois pontos. Porém, neste caso é necessário estender a linguagem com facilidades para

especificação da representação concreta (SHOWN AS), figura 3.5.

A edição de classes

O sistema gráfico, baseado nas regras de formação, provê operações de criação e remoção de classes e restrições, e operações de edição de classes: aumento/redução do tamanho e movimentação. Os comandos de edição atualizam os valores iniciais das classes. Na edição de uma classe composta por diversas subclasses a modificação de um valor é propagada para todas as subclasses afetadas. As restrições definem os interrelacionamentos entre as subclasses. Existem algoritmos, similares aos utilizados para GA, que a partir das restrições propagam a atualização dos valores das variáveis.

Envelope

Cada classe, como primitiva geométrica, possui um envelope. O envelope é um retângulo (invisível) que contorna toda a área ocupada pela classe. Se a classe é composta de subclasses, o envelope da classe deve conter todos os envelopes das subclasses.

Na edição, quando um objeto está selecionado, como objeto corrente, os pegadores são exibidos sobre a borda do envelope. Os pegadores são usados nas operações de aumento e redução. O aumento é executado pelo "arrasto" do pegador para uma nova posição, afastada em relação ao centro (figura 3.6). Para arrastar o pegador é necessário posicionar o "mouse" sobre o pegador.

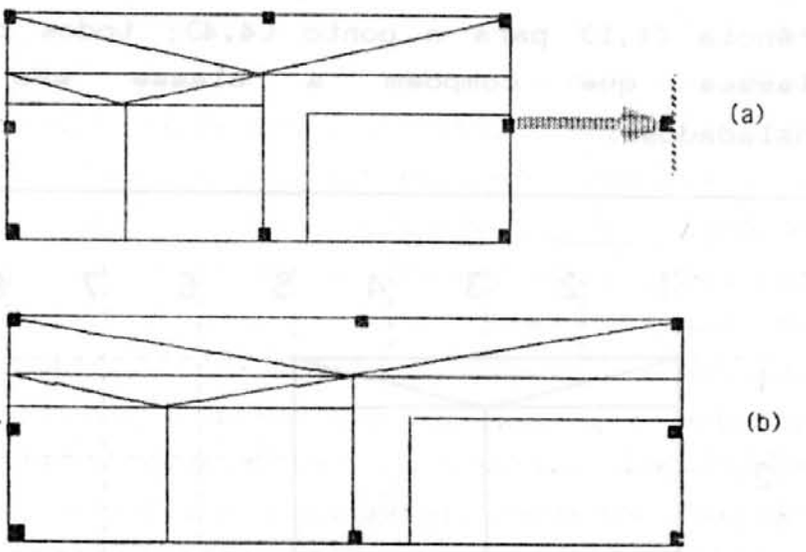


figura 3.6: os pegadores de um envelope: o aumento do tamanho da classe pelo arrasto do pegador

Uma descrição detalhada do envelope e das operações é encontrada em [MEL 89].

Movimentação

Pode-se pensar que cada objeto geométrico possui como ponto de referência (sua posição na tela) um ponto do envelope, por exemplo o canto superior esquerdo. Assim, quando o envelope é movimentado a classe associada é, juntamente, movimentada. Usualmente, na movimentação o "mouse" deve ser posicionado no interior do objeto selecionado e, nas operações de aumento e redução o "mouse" é posicionado sobre um dos pegadores.

A movimentação do envelope implica na atualização dos valores da classe. Após a movimentação, o objeto é reescrito na tela com os novos valores. A atualização

implica numa translação de todos os pontos da classe. A figura 3.7 ilustra a movimentação de uma classe do ponto de referência (1,1) para o ponto (4,4); todos os pontos das subclasses que compõem a classe são atualizados (transladados).

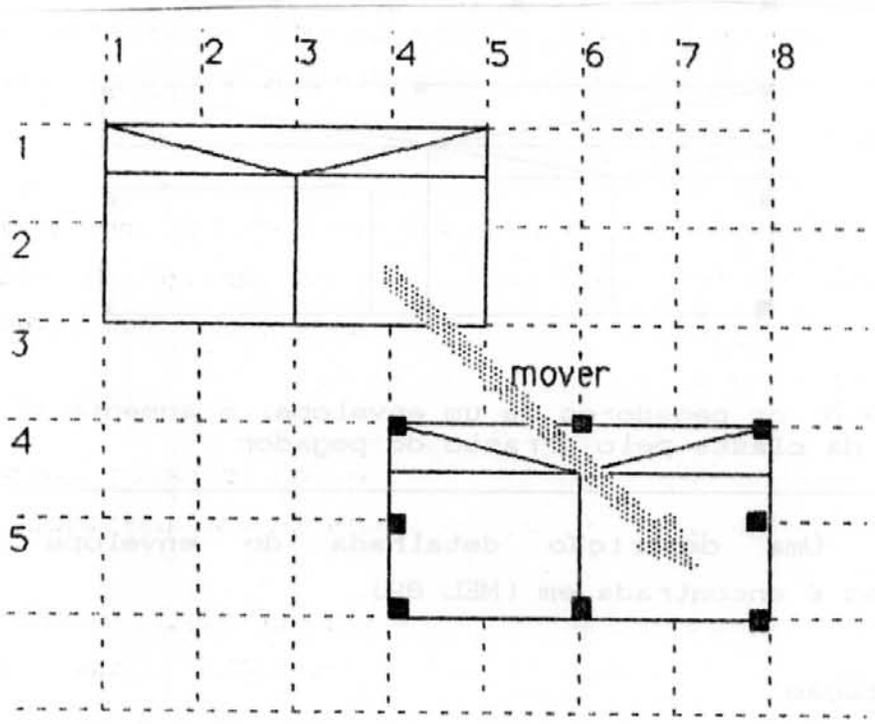


figura 3.7: movimentação de uma classe

Aumento e redução

O aumento/redução do envelope implica no aumento/redução do objeto geométrico. Na figura 3.6 exemplifica-se o aumento de uma classe.

Tanto na movimentação como no aumento/redução a atualização dos valores da classe ocorre somente se as restrições são verificadas.

O uso da linguagem "constraint-based"

As notações diagramáticas possuem como léxicos objetos geométricos compostos como "bolhas", "caixinhas", "setas", etc. Obviamente que as construções classe e restrição, introduzidas acima, permitem definir os objetos geométricos utilizado nas notações diagramáticas.

As principais vantagens da utilização de uma linguagem deste tipo para especificação do nível léxico de uma linguagem diagramática são:

- é uma notação formal: (a) no sentido de poder ser processada por um computador; (b) no sentido de poder ser trabalhada matematicamente; por exemplo, as restrições podem ser vistas como equações geométricas.

- a linguagem textual que define as classes serve como documentação da especificação; sendo também portável.

- existem algoritmos que, automaticamente, recalculam as variáveis e avaliam as restrições das classes interrelacionadas, após as operações de edição.

Está fora do escopo deste trabalho um estudo sobre a atualização das variáveis das classes com base nas restrições. Este problema é semelhante à avaliação das regras de uma gramática de atributos. Para isto sugere-se a leitura do trabalho aqui referido [BAR 87].

3.3.2 Especificação por seleção em cardápio

Em [MEL 89] é apresentada uma forma de especificação de elementos léxicos diagramáticos a partir de um alfabeto de classes geométricas primitivas.

A especificação dos elementos léxicos se faz de forma interativa. Escolhe-se componentes a partir de um cardápio de classes primitivas, e combina-se estes na formação de uma classe composta. Na figura 3.8 é identificado um objeto gráfico como elemento léxico: "caixinha" de um comando condicional (if) de um diagrama de N-S. Para formar esta "caixinha" combina-se, por exemplo, um retângulo e três segmentos de retas.

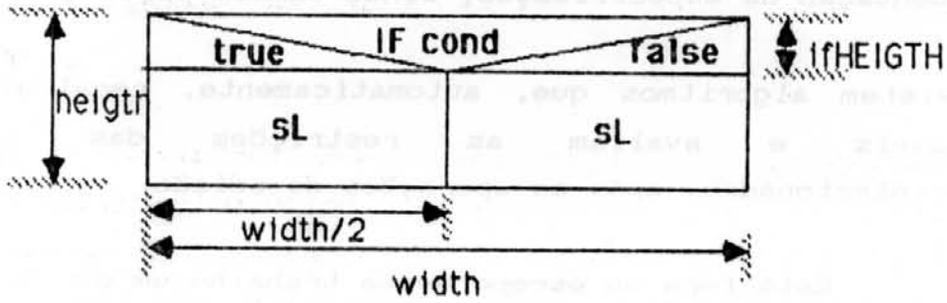


figura 3.8: a especificação de um elemento léxico

Numa analogia, as restrições da linguagem apresentada no item anterior ficam implícitas na sobreposição das figuras geométricas combinadas dentro de um envelope do elemento léxico.

O alfabeto de classes primitivas inclui os objetos geométricos:

- retângulo,
- retângulo sem cantos,
- elipse,
- polígono regular,
- reta,
- textos, etc.

3.3.3 Atributos associados a léxicos

Mostrou-se que um elemento léxico é definido a partir da combinação de classes primitivas. Uma classe primitiva é definida por variáveis que definem as coordenadas na página física, tamanhos dos elementos geométricos, etc. Para a GA um elemento léxico pode ser considerado como uma "caixa preta", com alguns parâmetros de entrada (atributos herdados) ou de saída (atributos externos).

A definição de uma classe associada ao símbolo terminal !IF é feita em função de atributos e constantes. Na figura 3.9-b está a descrição do elemento léxico da figura 3.8; usou-se os atributos da figura 3.9-a.

A associação entre a classe, que define o objeto geométrico no nível léxico, e o símbolo terminal para o nível da GA é feita pelo nome da classe seguido do identificador do símbolo terminal. Na figura 3.9 a construção

CLASS ClassIf:!IF IS ...

associa o terminal da GA !IF com a classe nomeada ClassIF.

Além da associação dos nomes é necessário fazer um mapeamento entre variáveis e atributos; na figura 3.9 cada atributo está associado à variável de mesmo nome. Por exemplo, os atributos herdados **x, y, width, heigth** declarados para o terminal !IF na linha

```
inh   INTEGER x, y, width, heigth : !IF;
```

são usados na equação que define o retângulo:

```
r   = rectangle(rp1=(x,y), rp2=(x+width, y+heigth));
```

```
( const UnitHEIGTH = 1;
        ifHEIGTH   = 2*UnitHEIGTH;
  inh   INTEGER x, y,
                width, heigth : !IF;
  ext   STRING  cond          : !IF; )
```

(a) atributos utilizados na definição do léxico if

```
CLASS ClassIf:!IF IS
   r   = rectangle(rp1=(x,y), rp2=(x+width, y+heigth));
   l1  = lineSeg( (x,y), (x+width/2, y+ifHEIGTH));
   l2  = lineSeg( (x+width/2,y+ifHEIGTH), (x+width,y);
   l3  = lineSeg( (x,y+ifHEIGTH), (x+width,y+ifHEIGTH));
   t1  =text(cond = "", (x,y), (x+width, y+ifHEIGTH/2));
   t2  =text("true",(x,y+ifHEIGTH/2),(x+width/2,y+ifHEIGTH);
   t3  =text("false", (x+width/2,y+ifHEIGTH/2),
                      (width,    y+ifHEIGTH);
END
```

(b) associação entre os atributos e a classe

figura 3.9: associação entre atributos e uma classe

Na classe da figura 3.9, que define o léxico da figura 3.8, os atributos **x** e **y** representam a posição associada ao canto superior esquerdo da "caixinha" do IF, e os atributos **width** e **heigth** definem respectivamente a largura e a altura da "caixinha".

O triângulo interno à "caixinha", figura 3.8, é desenhado a partir do valor da constante **ifHEIGTH**, e do valor **IF.width / 2**. A constante **ifHEIGTH** foi definida como duas vezes a **unitHEIGTH**, pois no interior da "caixinha" do **if** são utilizadas duas linhas, onde é desenhado um "triângulo".

A classe primitiva **text** exibe o texto centralizado no interior do envelope definido pelos dois pontos.

O atributo **cond** é do tipo externo ("ext") - ver capítulo 2. Este atributo é inicializado com brancos. O valor final deste atributo é lido da tela - é editável a nível léxico.

A nível de GA, cada elemento léxico é visto e tratado como uma entidade fechada. Associado a cada léxico pode-se ter atributos externos ou herdados. Os atributos do tipo herdado são recebidos de nodos superiores na AS. Os atributos do tipo externo são visíveis aos nodos superiores na AS.

4 DIAGRAMA DE NASSI-SHNEIDERMAN (N-S)

Este capítulo descreve uma especificação do diagrama de Nassi-Shneiderman, dando ênfase para a formatação e a integração entre notações diagramáticas e textuais. Tomou-se por base a publicação de uma implementação de um editor [HAL 87] para esta notação. A especificação considera apenas uma página do diagrama; numa implementação real deve-se considerar o problema do refinamento de elementos da página em outras páginas. Este problema é discutido no próximo capítulo.

O diagrama de Nassi-Shneiderman, figura 4.1, é um exemplo de uma notação com estrutura em forma de árvore (sem nodos compartilhados).

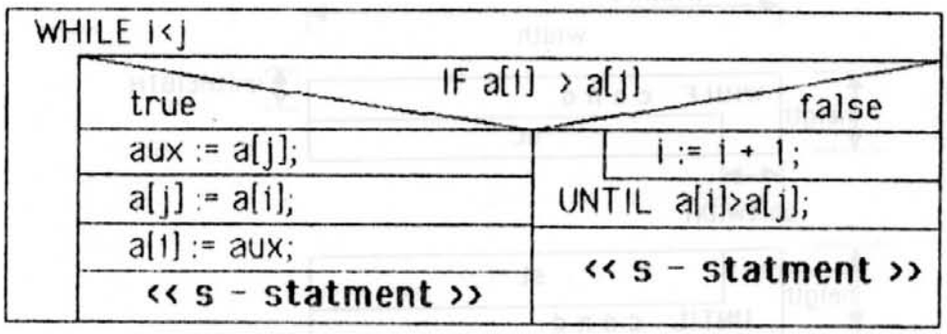


figura 4.1: Diagrama Nassi-Shneiderman (N-S)

Os itens 4.1, 4.2 e 4.3 discutem, respectivamente, os aspectos associados ao nível léxico, sintático e semântico. O item 4.4 apresenta a integração a nível de LDE para notações textuais e diagramáticas.

4.1 Especificação dos elementos léxicos

Cada classe associada a um elemento léxico do tipo diagramático é definida em função de um conjunto de atributos e algumas constantes, que são usadas para a exibição do elemento léxico na tela. As constantes **unitWIDTH** e **unitHEIGTH** definem respectivamente a menor unidade no sentido vertical e a menor unidade no sentido horizontal (figura 4.2).

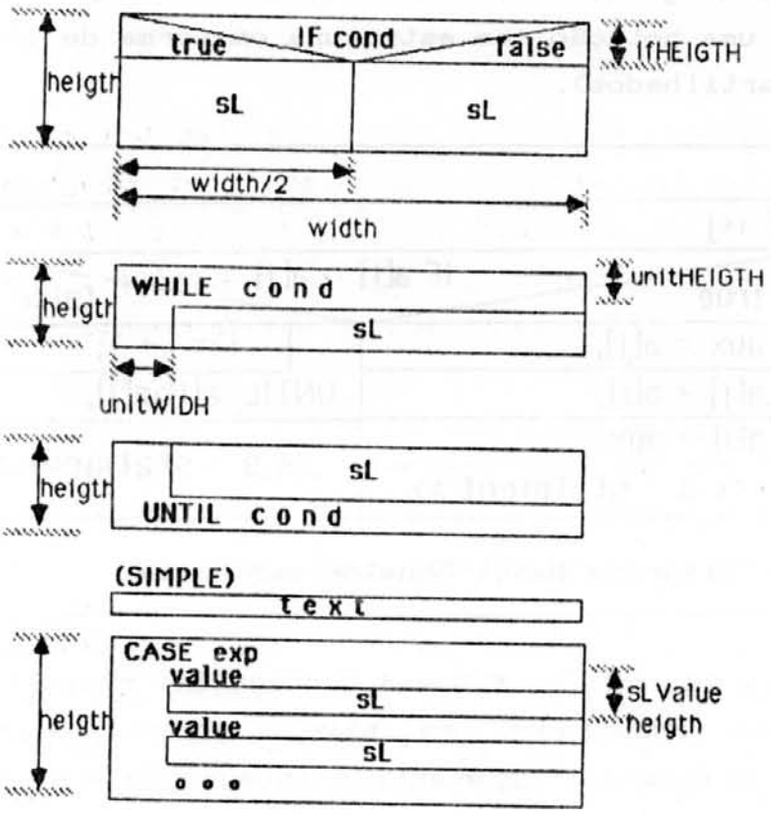


figura 4.2: Elementos léxicos para o N-S

A declaração abaixo define quatro atributos (heigth, width, x, y) herdados para os terminais !IF, !WHILE, !REPEAT, !CASE, !SIMPLE.

```
(inh INTEGER heigth, width, x, y : !IF, !WHILE, !REPEAT,
                                   !CASE, !SIMPLE; )
```

A especificação do léxico !IF foi discutida no capítulo 3. Os demais léxicos seriam definidos de maneira análoga.

4.2 Especificação da sintaxe

Como se vê na figura 4.1, o diagrama é constituído de caixinhas decoradas com textos - o diagrama possui aspectos textuais e diagramáticos. No final deste capítulo discute-se a integração entre notações textuais e diagramáticas na descrição em LDE, com base neste exemplo.

Os elementos terminais na LDE são pré-fixados pelo operador "!": tem-se o símbolo terminal do tipo texto !value e diversos símbolos terminais diagramáticos escritos em letras maiúsculas (!IF, !CASE, etc.).

No exemplo de especificação do léxico !IF no capítulo 3 o símbolo Cond foi considerado um atributo externo, editável como um texto qualquer. No entanto, o texto associado ao símbolo Cond possui uma sintaxe: é uma expressão que deve resultar num valor booleano. A sintaxe deste não-terminal pode ser descrita por um conjunto de produções textuais.

```
%significado das abreviações para identificadores
s: (statement) comando
L: (list) lista
sL: (statement-list) lista de comandos
sLValue: (statement-list-value) lista de claúsulas de um
        comando case
valueStmt: uma claúsula de um comando case
!IF, !WHILE, !REPEAT,
!CASE, !SIMPLE          :terminais diagramáticos
cond : expressão do tipo booleano
exp  : expressão numérica
!value :valor escalar (terminal do tipo texto)
stmt : (statement) comando textual%


gr --> sL
s  --> if
s  --> case
s  --> while
s  --> repeat
s  --> simple

if     --> !IF     cond sL sL
while  --> !WHILE  cond sL
repeat --> !REPEAT cond sL
case   --> !CASE   exp  sLValue
simple --> !SIMPLE stmt

sLValue --> valueStmt sLValue
sLValue --> ^

valueStmt --> !value sL

sL --> s sL
sL --> ^

cond --> exp '<' exp
cond --> ...

exp --> exp '+' exp
exp --> exp '/' exp
exp --> ...
stmt --> ...
```

figura 4.3: GLC para o diagrama de Nassi-Shneiderman

A figura 4.3 mostra a descrição da sintaxe do

diagrama, onde aparecem produções textuais e produções diagramáticas. Os não-terminais associados aos textos (Cond, Exp, Stmt) não são detalhados. O modo de especificar linguagens textuais foi exemplificado no capítulo 2.

A especificação da sintaxe define a combinação dos diferentes elementos léxicos que estão representados na figura 4.2, formando diagramas como o da figura 4.1. Por exemplo, um comando if é formado pela combinação da "caixinha" !IF com a condição Cond e com duas listas de comandos sL, figura 4.3.

Cada retângulo corresponde a uma lista de comandos sL. O comando case apresenta uma estrutura composta por condição Cond seguida de uma lista de retângulos associados com um valor (case, sLValue). As demais produções sintáticas são obtidas de maneira análoga a partir da figura 4.2.

4.3 Especificação da formatação

Na parte semântica do N-S descreve-se apenas a formatação. As verificações dependentes de contexto (nomes não declarados, duplicados, etc.) seriam especificadas de forma similar aos exemplos apresentados no capítulo 2.

A formatação física consiste em exibir na tela as caixinhas com base nos valores dos atributos associados, calculados pelo formalismo de GA.

O ponto de partida para a especificação da formatação é, por um lado, a definição dos atributos utilizados pelo nível léxico e, por outro lado, a estrutura

de árvore, definida na sintaxe, onde os atributos podem passar de nodo em nodo. Estas informações guiam o processo da especificação.

Os símbolos do tipo texto (exp, cond, stmt, !value) possuem os atributos (x, y, width) que definem o envelope para o texto. Assume-se que o texto é exibido em apenas uma linha; assim, não é necessário o atributo heigth. O editor textual usará estes atributos para definir a janela de edição, quando é ativado.

Para denotar que todos os valores de atributos (com mesmo nome) associados a um símbolo são passados para outro símbolo usa-se a notação abreviada "*". Por exemplo, a equação que define os atributos do terminal !SIMPLE

```
(inh  INTEGER width, x, y : !SIMPLE;
 !SIMPLE.* = s.*;)
```

é equivalente a

```
(!SIMPLE.x = s.x; !SIMPLE.y = s.y; !SIMPLE.width=s.width;).
```

Os valores dos atributos que definem o canto superior esquerdo de cada caixinha são herdados a partir dos valores iniciais da produção raiz (gr), figura 4.4. O atributo **width** também é passado para baixo na AS, subtraindo-se o valor utilizado para cada construção. E o atributo **heigth** (sintetizado) é calculado a partir das estruturas mais internas para as mais externas, figura 4.4.

Para se entender a especificação é necessário acompanhar, comparar, cada produção da figura 4.4, com a sua respectiva caixinha na figura 4.2. Abaixo é mostrada a especificação da produção **if**.

```
if      --> !IF !cond sL1 sL2
 ( sL1.width = if.width / 2;
   sL2.width = if.width / 2;
   if.heigth = Max(sL1.heigth, sL2.heigth);
   sL1.y     = if.y + ifHEIGTH;
   sl2.y     = if.y + ifHEIGTH;
   sL1.x     = if.x;
   sL2.x     = if.x + if.width / 2;
   !IF.*     = if.*;
   cond.*    = if.*;)
```

Na produção if o atributo **sL1.width**, da sublista de comandos, recebe o valor **if.width/2**; metade do valor recebido pelo **if**. O valor do canto superior esquerdo da caixinha da segunda sublista (**sL2**) é **if.y+ifHEIGTH** para **y** e **if.x+if.width/2** para **x**. O atributo **if.heigth**, sintetizado, recebe a maior das alturas das sublistas (**sL1**, **sL2**).

```
(inh  INTEGER heigth, width, x, y : !IF, !WHILE, !REPEAT,
                                    !CASE, !SIMPLE;
 synt INTEGER heigth              : if, while, repeat,
                                    case, simple, s, sl;
 inh  INTEGER width, x, y         : if, while, repeat,
                                    case, simple, s, sl
                                    sLvalue, valueStmt,
                                    exp, cond, stmt, !value;)

( Const unitWIDTH  = 1;
        maxWIDTH   = 80;
        unitHEIGTH = 1;
        ifHEIGTH   = 2 * unitHEIGTH; )

gr --> sL
 ( sL.width = maxWIDTH;
   sL.x     = 0;
   sL.y     = 0;)

s  --> if
  ( if.x = s.x;
    if.y = s.y;
    if.width = s.width;
    s.heigth = if.heigth; )


s  --> case
( case.x = s.x;
  case.y = s.y;                                continua ...
```

```
   case.width = s.width;
   s.heigth   = case.heigth; }

s  --> while
{ while.x = s.x;
   while.y = s.y;
   while.width = s.width;
   s.heigth    = while.heigth; }

s  --> repeat
{ repeat.x = s.x;
   repeat.y = s.y;
   repeat.width = s.width;
   s.heigth     = repeat.heigth; }

s --> simple
{ simple.x = s.x;
   simple.y = s.y;
   simple.width = s.width;
   s.heigth     = simple.heigth; }

simple  --> !SIMPLE  stmt
   { simple.heigth  = unitHEIGTH;
     !SIMPLE.* = simple.*;
     stmt      = simple.*;}

if      --> !IF cond sL1 sL2
 { sL1.width = if.width / 2;
    sL2.width = if.width / 2;
    if.heigth = Max(sL1.heigth, sL2.heigth);
    sL1.y     = if.y + ifHEIGTH;
    sl2.y     = if.y + ifHEIGTH;
    sL1.x     = if.x;
    sL2.x     = if.x + if.width / 2;
    !IF.*     = if.*;
    cond.*    = if.*; }

while  --> !WHILE cond sL
 { while.heigth = sl.heigth   + unitHEIGTH;
    sl.width     = while.width - unitWIDTH;
    sL.y     = while.y + unitHEIGTH;
    sL.x     = while.x - unitWIDTH;
    !WHILE.* = while.*;
    cond.*  = while.*; }

repeat --> !REPEAT cond sL
 { repeat.heigth = sL.heigth    + unitHEIGTH;
    sL.width      = repeat.width - unitWIDTH;
    sL.y          = if.y;
    sL.x          = if.x - unitWIDTH;
```

continua ...

```
    cond.y        = repeat.y + sl.heigth;
    cond.x        = repeat.x;
    cond.width    = repeat.width;
    !REPEAT.*     = repeat.*;}

case    --> !CASE exp sLValue
 { case.heigth    = sLValue.heigth + unitHEIGTH;
   sLValue.width = case.width    - unitWIDTH;
   sLValue.y = case.y + unitHEIGTH;
   sLValue.x = case.x - UnitWIDTH;
   !CASE.*    = case.*;
   exp.*      = case.*;}

sLValue --> valueStmt sLValue
  { sLValue1.heigth  = valueStmt.heigth + sLValue2.heigth;
    sLValue2.width   = sLValue1.width;
    valueStmt.width  = sLValue1.width;
    sLValue2.y  =  sLValue1.y + valueStmt.heigth
    sLValue2.x  =  sLValue1.x;}

sLValue --> ^
 { sLValue.heigth = 0;}

valueStmt --> !value sL
  { valueStmt.heigth = unitHEIGTH + sL.heigth;
    sL.width          = valueStmt.width;
    sL.y      = valueStmt.y + unitHEIGTH;
    sL.x      = valueStmt.x;
    !value.* = valueStmt.*;}

sL --> s sL
  { sL1.heigth = s.heigth + sL2.heigth;
    sL2.width  = sL1.width;
    sL2.y = sL1.y + s.heigth;
    sL2.x = sL1.x;
    s.*   = sL1.*;}

sL --> ^
  { sL.heigth = 0; }

cond --> exp '<' exp
cond --> ...

exp --> exp '+' exp
exp --> exp '/' exp
exp --> ...
stmt --> ...
```

figura 4.4: GA para o diagrama de Nassi-Shneiderman

Na especificação define-se um valor constante para a largura do diagrama (maxWIDTH); o comprimento não tem um limite fixo, é o valor sintetizado pelo atributo **sL.heigth** no nodo raiz (Gr) da árvore.

Na hora de exibir o diagrama na tela é necessário ajustar o comprimento real do diagrama com o tamanho da tela. A altura total do diagrama (**sL.heigth**) pode ser usada para isto; o ajuste se dá pela relação entre o número de pontos da tela e o valor **sL.heigth**.

4.4 Integração entre o editor de texto e diagramas

A integração entre o editor textual e o diagramático é feita a nível de páginas - uma página pode ser textual ou diagramática.

Usualmente, os documentos são divididos em páginas lógicas. Uma página é uma unidade (porção) conceitual de um documento. Por exemplo, o N-S como um todo é visto como uma página, e cada texto dentro de uma caixinha (por exemplo Cond) também corresponde a uma página (mesmo sendo de apenas uma linha de texto).

Cada página é descrita por um conjunto de produções. No caso do N-S, figura 4.5, as produções **gr**, **cond**, **exp** e **stmt** estão associadas a páginas. A página gr é a página inicial.

A associação do conceito de página à definição gramatical é feita pelos prefixos GED (editor diagramático) e GET (editor textual) figura 5.4.

```
% O significado dos identificadores
  esta descrito no início da figura 4.3 %

GED ::    gr --> sL
          s  --> if
          s  --> case
          s  --> while
          s  --> repeat
          s  --> simple

          if     --> !IF     cond sL sL
          while  --> !WHILE  cond sL
          repeat --> !REPEAT cond sL
          case   --> !CASE   exp  sLValue
          simple --> !SIMPLE cmdo

          sLValue --> valueStmt sLValue
          sLValue --> ^

          valueStmt --> !value sL

          sL --> s sL
          sL --> ^

GET ::    cond --> exp '<' exp
          cond --> ...

GET ::    exp --> exp '+' exp
          exp --> exp '/' exp
          exp --> ...

GET ::    stmt --> ...
```

figura 4.5: integração entre textos e diagramas na LDE

Os prefixos GED e GET indicam respectivamente a ativação do editor diagramático e do editor de texto. Cada editor é ativado sobre uma página lógica. O GED e GET podem ser recursivamente ativados. Pela descrição acima, quando o GED expande a produção (if --> !IF cond sL sL) o GET é ativado. Quando o GET é ativado para a edição da página associada ao símbolo cond deverá consultar os valores dos atributos deste símbolo para saber a posição exata onde será criada a janela para a edição.

A integração a nível da arquitetura de implementação é discutida no capítulo 6.

5 DIAGRAMA DE FLUXO DE DADOS

Neste capítulo as facilidades para especificação de linguagens do tipo grafo são introduzidas a partir de um exemplo de diagrama de fluxo de dados (DFD), apresentado na figura 5.1

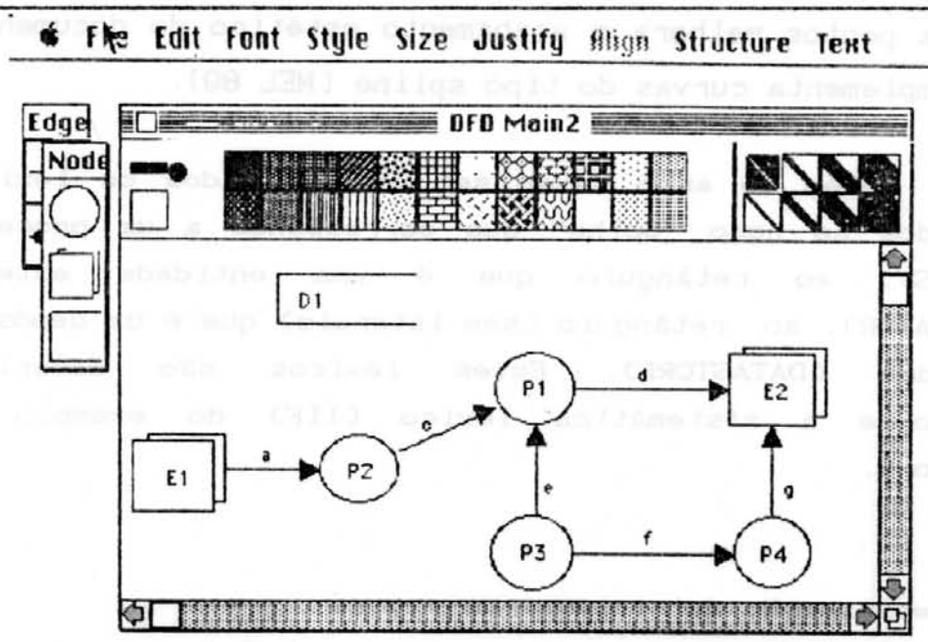


figura 5.1: uma página de um DFD; criado com o EDG [MEL 89]

5.1 Nível léxico

Um diagrama em forma de grafo possui elementos léxicos do tipo arco e nodo.

Um arco liga dois elementos do tipo nodo. Numa forma simples, o arco !ARC pode ser especificado como um elemento geométrico seta (reta com uma ponta) que liga dois pontos **point1** e **point2**. O arco possui também um nome **Name**

que é exibido no ponto médio do arco. Segue a definição dos atributos do arco:

```
( inh   TpPOINT point1, point2 : !ARC;
  ext   STRING  name           : !ARC;)
```

Ao invés de se utilizar retas para desenhar o arco, pode-se utilizar curvas; uma curva passando por diversos pontos melhora o acabamento estético do documento; o EDG implementa curvas do tipo spline [MEL 89].

Além do arco devem ser especificados os léxicos associados ao nodo "bolha" que corresponde a um processo (PROCESS), ao retângulo que é uma entidade externa (TERMINATOR), ao retângulo (sem laterais) que é um depósito de dados (DATASTORE). Estes léxicos são descritos seguindo-se a sistemática léxico (!IF) do exemplo do capítulo 3.

5.2 Especificação da sintaxe

Na sintaxe do DFD deve ser especificado o conceito de grafo (arcos e nodos). A partir da figura 5.1 pode-se observar alguns aspectos estruturais de um DFD:

i) Uma página lógica de um DFD é composta de um identificador (nome de página), um conjunto* de entidades (retângulos), um conjunto de processos (bolhas), um conjunto de depósitos (retângulos) e um conjunto de ligações (arcos).

ii) Cada entidade, depósito ou processo é considerado como

---

*NOTA: usa-se aqui o termo conjunto, para diferenciar do termo lista; conjunto não possui uma relação de ordem entre os elementos e nem elementos duplicados.

um item léxico, i.e. um objeto fechado para o nível sintático.

iii) Uma ligação (fluxo de dados) ocorre entre dois elementos do tipo entidade, processo ou depósito. Um elemento pode participar de várias ligações e em toda ligação deve existir pelo menos um processo.

Numa notação de GLC não existe o conceito de conjunto (i), porém pode-se descrever um conjunto por uma lista e uma regra semântica que verifica a unicidade dos elementos da lista. Assim, descreve-se "um conjunto de entidades" como:

```
terminator-list  --> !TERMINATOR  terminator-list
terminator-list  --> ^
```

A regra semântica, de unicidade, é definida a partir da tabela TERMINATOR que contém todos os nomes das entidades. A restrição é uma equação em álgebra relacional que verifica se existem dois nomes iguais na tabela TERMINATOR. Reescreve-se a produção acima utilizando só as iniciais dos identificadores para compactação:

```
< table*  TERMINATOR( Name:string ); >

tL  -->  !Tg tL
  ( <Tg.Nome> In TERMINATOR;
   error = if card(TERMINATOR
                   Where Tg.name = TERMINATOR.Name)
                   > 1  then "Name duplicated" else ""; )
```

Na minilinguagem de expressões, capítulo 2 - figura 2.3, o atributo error, local ao não-terminal, devia ser exibido intercalado ao texto; para o DFD este atributo

---

*NOTA: Assume-se que a tabela pode conter tuplas duplicadas.

pode ser exibido na linha de mensagens de erro da janela de edição.

Assumindo-se que a orientação do arco é dirigida da ocorrência do primeiro nodo para a ocorrência do segundo nodo, a produção **DataFlow** descreve o fluxo de dados. Os símbolos !TERMINATOR, !DATASTORE, !PROCESS e !ARC são terminais do tipo gráfico.

```
DataFlow --> !ARC Node Node
Node     --> !TERMINATOR
Node     --> !DATASTORE
Node     --> !PROCESS
```

Esta produção apenas descreve que um fluxo de dados é a combinação de um arco com dois nodos; para indicar que estes dois nodos são compartilháveis reescreve-se a produção prefixando com o operador (?), ao invés de (!), os elementos terminais que serão compartilhados.

```
DataFlow --> !ARC Node Node
Node     --> ?TERMINATOR
Node     --> ?DATASTORE
Node     --> ?PROCESS
```

Agora, estas produções podem ser lidas como: "um fluxo de dados é gerado pela criação de um terminal !ARC seguido de dois nodos **node**, sendo que cada **node** é formado pelo compartilhamento de um nodo terminal do tipo !TERMINATOR, !DATASTORE ou !PROCESS."

```
%Usa-se só as iniciais de cada simbolo para facilitar a
escrita das regras semânticas. Segue o significado dos
identificadores:
Sp : subprocess (subprocesso)
tL : terminator-list (lista de entidades)
pL : process-list (lista de processos)
dL : datastore-list (lista de depósitos)
fL : dataflow-list (Lista de fluxos)
Osp: optional subprocess (Sub-processo opcional)
Node : classe de elementos para apontamento
P  : process (nao terminal processo)
Pg : process-graphic (terminal)
Tg : terminator-graphic (terminal)
Ag : arc-graphic(terminal)
Dg : datastore-graphic(terminal) %

Dfd --> Sp
Sp  --> !Name tL dL pL fL
tL  --> !Tg tL
tL  --> ^
dL  --> !Dg dL
dL  --> ^
pL  --> P  pL
pL  --> ^
fL  --> F fL
fL  --> ^
F   --> !Ag Node Node
Node  --> ?Tg
Node  --> ?Pg
Node  --> ?Dg
P   --> Pg Osp
Osp --> Sp
Osp --> ^
```

figura 5.2: descrição estrutural de um DFD

A figura 5.2 apresenta a descrição da estrutura do DFD. Nesta figura existem dois tipos de elementos terminais: os prefixados por (!) e os prefixados por (?). A diferença entre eles é que o último denota símbolos terminais compartilhados na AS.

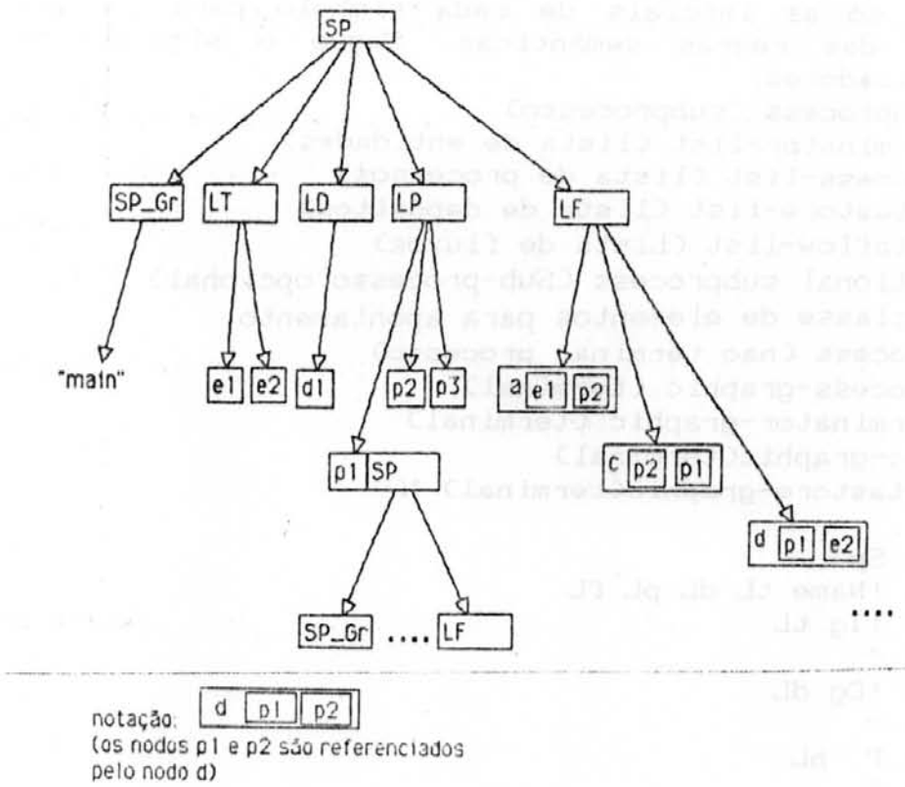


figura 5.3: a árvore sintática (como grafo) da página do DFD da figura 5.1.

A notação proposta explora de forma simples a possibilidade de compartilhamento de nodos na AS, permitindo que um operador de compartilhamento (?), prefixado a um elemento da produção, indique que a produção referida deva já existir na AS. Na AS é criado um ponteiro para cada instância de nodo compartilhado, figura 5.3.

Na descrição sintática está definido o conceito de refinamento de processo pelas produções (P --> Pg Osp, Osp --> Sp). Uma extensão na notação que permite a associação do conceito de página à descrição gramatical é discutida no item 5.6.

5.3 O formalismo gramatical (GLC estendida)

As regras de uma GLC ou as regras de um formalismo gramatical utilizadas para a descrição de aspectos estruturais (livres de contexto) podem ser classificadas em cinco tipos (ver por ex. [DON 84], [REP 84] [NAG 87]):

tipo seleção: para seleção de um conjunto finito de alternativas, por exemplo:

```
Node  --> ?Tg
Node  --> ?Pg
Node  --> ?Dg
```

tipo opcional: para expressar que uma estrutura é opcional, por exemplo:

```
Osp --> Sp
Osp --> ^
```

tipo lista: define uma ou mais ocorrências de elementos agrupados sob este nodo, por exemplo (pL, tL, dL, fL):

```
fL  --> !Fg fL
fL  --> ^
```

tipo agregação: um nodo é construído pelo agrupamento de diversos elementos constituintes, por exemplo (Sp, F):

```
Sp  --> !Name tL dL pL fL
```

tipo elementar: é um item terminal na GLC (!Ng, !Dg, !Fg, ?Tg, ?Pg). Existem duas classes de itens terminais: as que geram nodos na ASA (!) e os que compartilham nodos (?) gerando ponteiros entre nodos da ASA, como exemplificado nas figuras 5.2 e 5.3.

Todo elemento gráfico (que possui uma representação concreta na tela) deve estar associado a um tipo elementar, visto como um objeto léxico fechado para o nível sintático. O tipo elementar corresponde a uma folha na estrutura de representação interna. Cada símbolo do tipo elementar é graficamente definido por uma classe (capítulo 2).

5.4 Normalização das regras da GLC

A fim de facilitar o desenvolvimento do protótipo, foram impostas duas restrições para a descrição das produções:

a) as produções devem definir uma GLC tipo LL(1) [AHO 86];

b) todas a produções deve estar normalizadas, i.e. podem ser classificadas nos 5 tipos citados acima.

As produções que não podem ser classificados nos tipos descritos acima devem ser reescritas. Por exemplo a produção

```
A --> B C,
A --> D E,
A --> ^
```

que é ao mesmo tempo do tipo seleção e opcional, deve ser reescrita, para as produções abaixo:

```
Ao --> A,
Ao --> ^,
A  --> B C,
A  --> D E,
```

agora a primeira é do tipo opcional e a segunda, do tipo alternativo.

5.5 Extensão semântica para grafos

A extensão para o conceito de grafo está relacionada ao tipo elementar. Arcos são criados pelo compartilhamento de nodos. Os nodos do tipo elementar de apontamento (?) são também chamados de donatários (que recebem uma doação). Na representação interna uma instância de um nodo donatário pode ser vista como um ponteiro para o nodo elementar compartilhável que está sendo referenciado (figura 5.3). Os nodos referenciados pelos nodos donatários devem ter sido criados anteriormente.

Numa estrutura de grafo a remoção de um nodo implica na remoção de todos os arcos incidentes. Portando, numa operação de remoção de um nodo compartilhado deve-se propagar o efeito para os nodos donatários - i. e., remove-se o nodo compartilhado juntamente com todos os nodos donatários.

A consistência para a GLC estendida pelo conceito de grafo é baseada em duas regras:

- é obrigatória a existência do nodo compartilhado (a ser apontado) no momento da criação de um nodo donatário;
- na remoção de um nodo compartilhado remove-se juntamente todos os nodos donatários.

A notação proposta apenas introduz o conceito de nodo compartilhado. O conceito de grafo (nodos e ligações) é assumido na interpretação: por exemplo, na especificação do DFD assume-se o conceito de "arco direcionado" (assumindo a orientação do arco do primeiro para o segundo elemento da produção F --> !Ag Node Node).

Discutiu-se a extensão da notação GLC para permitir o conceito de nodos compartilhados apenas a nível de elementos terminais da GLC, i.e. apenas podem ser compartilhados elementos léxicos. O conceito de compartilhamento aplica-se também para não-terminais da GLC. Por exemplo em um documento de "hiper-texto" os elementos compartilhados podem ser fragmentos de texto, onde cada fragmento está associado a um não-terminal. O compartilhamento de não-terminais é tema de trabalhos futuros.

5.6 O conceito de página

No capítulo anterior discutiu-se uma extensão que permite associar páginas às produções da linguagem LDE, prefixando-se as produções por GET ou GED.

```
%Significado dos identificadores
Sp : subprocess (subprocesso)
Osp: optional subprocess (subprocesso opcional)
P  : process (processo, não-terminal)
pL : process-list (lista de processos)
Pg : process-graphic (elemento léxico de uma processo)%

1. GED   :: Dfd --> Sp
2. GED   :: Sp  --> !Name tL dL pL fL
         ...
3.          pL  --> P pL
4.          P   --> Pg Osp
5.          Osp --> Sp
6.          Osp --> ^
```

figura 5.4: associação de páginas na descrição gramatical

Para o DFD, o primeiro nível corresponde a uma página; cada refinamento de um processo, nos demais níveis, também corresponde a uma página.

Cada página é descrita por um conjunto de produções. No caso do DFD uma página está associada a produção Sp (Subprocesso). A produção Dfd --> Sp, figura 5.4, define a página inicial, nível zero.

A hierarquia de páginas do DFD é definida pela decomposição opcional de um processo em subprocessos: as produções 2-3, figura 5.4, definem uma página de DFD contendo uma lista de processos; a produção 4 define um processo como um componente léxico Pg e um componente Osp; as produções 5-6 definem o componente Osp como subprocesso (opcional), chamando recursivamente a produção associada à página Sp.

A associação do conceito de página à definição gramatical é feita pelo prefixo GED (editor diagramático). O prefixo GED na primeira linha corresponde a uma página vazia - não é necessária a inclusão de elementos para se expandir a produção Dfd --> Sp; a primeira página com elementos é associada a produção Sp.

```
DATAFLOW  ( Name :string,
            From :string, TypeFrom :string,
            To   :string, TypeTo   :String )
TERMINATOR( Name :string )
DATASTORE ( Name :string )
PROCESS   ( Name :string )

1. "Datastore without inputs or outputs"
let tab = (DATASTORE[ Name ]
            MINUS
           (DATAFLOW[ From ] WHERE TypeFrom = "DataStore"
            UNION
           DATAFLOW[ To   ] WHERE TypeTo   = "DataStore"))
in if Card(tab) > 0
      then  writeln( "datastore without inputs or outputs",
                     tab);

2. "Process without inputs or outputs"
PROCESS[ Name ] MINUS
((DATAFLOW[ From ] WHERE TypeFrom = "Process")
   UNION
 (DATAFLOW[ To   ] WHERE TypeTo   = "Process"))

2.1 "Process without outputs"
PROCESS[ Name ] MINUS
(DATAFLOW[ From ] WHERE TypeFrom = "Process")

2.2 "Process without inputs"
PROCESS[ Name ] MINUS
(DATAFLOW[ To ] WHERE TypeTo = "Process")

3. "Terminator without inputs or outputs"
TERMINATOR[ Name ] MINUS
((DATAFLOW[ From ] WHERE TypeFrom = "Terminator")
   UNION
 (DATAFLOW[ To   ] WHERE TypeTo   = "Terminator"))

4. "DataFlow without process"
DATAFLOW[ Name ] WHERE     (TypeFrom <> "Process")
                       and (TypeTo   <> "Process")
```

figura 5.5: especificação das verificações sobre a TS

5.7 Especificação da semântica

Existem dois tipos de consistência que podem ser executados em um DFD:

i) internas a GA - em tempo de edição: por exemplo, a inclusão de um nome duplicado, especificado no item 5.2;

ii) externas a GA - são verificações expressas em álgebra relacional sobre as informações das TSs: estas verificações podem ser ativadas a qualquer momento durante a edição ou somente após a edição. Como exemplo pode-se listar todos os processos sem entrada, processos sem saída, fluxos sem processo, depósito de dados sem entrada ou saída, etc.

Verificações externas

As verificações externas à GA podem ser ativadas a qualquer momento porque as informações das tabelas relacionais estão sempre consistentes; são mantidas pela GA.

A partir das tabelas TERMINATOR, PROCESS, DATASTORE e DATAFLOW define-se as verificações, como equações em álgebra relacional, para alguns estados inconsistentes de um DFD, figura 5.5.

Equações semânticas

A descrição semântica inclui as equações definidas para coletar as informações das TSs, para coletar valores dos atributos associados aos símbolos terminais e para fazer a consistência em tempo de edição (interna a GA).

```
< ext STRING  Name             : !Tg, !Pg, !Dg, !Ng, F, !Ag;
  ext TpPOINT point            : !Tg, P, !Pg, !Dg;
  inh TpPOINT point1, point2   : !Ag;
  syn TpPOINT point            : Node;
  syn STRING  Name, Type       : Node;>

1.0 Dfd --> Sp
2.0 Sp  --> !ID tL dL pL fL
3.0 tL  --> !Tg tL
     ( <!Tg.Name> In TERMINATOR;
       error = CheckName( !Tg.Name, TERMINATOR); )
3.1 tL  --> ^
4.0 dL  --> !Dg dL
             ( <!Dg.Name> In DATASTORE;
               error = CheckName(!Dg.Name, DATASTORE);>
4.1 dL  --> ^
5.0 pL  --> P pL
            ( <P.Name> In PROCESS;
              error = CheckName( P.Name, PROCESS); )
5.1 pL  --> ^
6.0 fL  --> F fL
6.1 fL  --> ^
7.0 F   --> !Ag Node Node
            ( <F.Name,
               Node1.Name, Node1.Type
               Node2.Name, Node2.Type > In DATAFLOW;
               error = CheckName(!Ag.Name, DATAFLOW);
               !Ag.Point1 = Node1.Point;
               !Ag.Point2 = Node2.Point; )
8.0 Node  --> ?Tg
              ( Node.Point = ?Tg.Point;
                Node.Name  = ?Tg.Name;
                Node.Type  = "Terminator" )
8.1 Node  --> ?Pg
              ( Node.Point = ?Pg.Point;
                Node.Name  = ?Pg.Name;
                Node.Type  = "Process" )
8.2 Node  --> ?Dg
              ( Node.Point = ?Dg.Point;
                Node.Name  = ?Dg.Name;
                Node.Type  = "Datastore";)
9.0 P  ---> !Pg Osp  ( P.Point = !Pg.Point; )
10.0 Osp ---> Sp
10.1 Osp ---> ^
```

figura 5.6: GA para o DFD

Os léxicos do tipo nodo possuem os atributos externos **Name** e **Point** que definem, respectivamente, o nome do nodo e as coordenadas (x,y) na tela. O léxico do tipo arco possui um atributo externo **Name** e dois atributos **Point** que são herdados dos dois nodos que estão ligados; correspondem às extremidades do arco. Os atributos do tipo ext são lidos da tela no momento da edição ou são valores "default" da classe que define os léxicos. A declaração dos atributos dos símbolos terminais está no início da figura 5.6.

GA e nodos compartilhados

Os atributos para os nodos donatários (pré-fixados por "?") não são declarados, pois são os mesmos atributos do nodos referenciados. Por exemplo, na produção 8.0 o símbolo ?Tg tem os atributos **point** e **name** que são os mesmos do nodo referenciado !Tg.

Quando o nodo compartilhado (!) é atualizado, automaticamente todos os atributos dos nodos donatários (?) são atualizados.

Verificações internas à GA

Na consistência interna à GA, definida pela equações da figura 5.6, a unicidade dos nomes é verificada pela função **CheckName**, descrita abaixo.

```
type CheckName: STRING x TABLE ---> STRING
CheckName( n, TAB ) =
        if card(TAB where TAB.Name = n) > 1
            then "Name duplicated" else "";
```

A especificação do DFD como um todo inclui:

i) nível léxico: Especificação dos elementos léxicos, associados aos atributos.

ii) nível sintático: Descrição da sintaxe na GLC estendida (figura 5.2) e da associação das páginas às produções (figura 5.4).

iii) nível semântico: A parte dependende de contexto inclui:

iii.1) as equações para definir os atributos do nível léxico (figura 5.6).

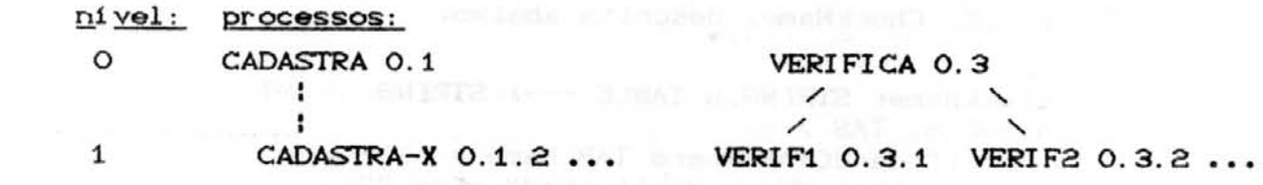
iii.2) definição das TSs associadas a GA: definição das tabelas relacionais, bem como das equações que coletam os valores para as tabelas (figura 5.6).

iii.3) as equações que definem a consistência interna a GA (as mensagens mostradas em tempo de edição), (figura 5.6).

iii.4) as verificações expressas em álgebra relacional que são definidas sobre as informações das TSs (figura 5.5).

5.8 Enumeração automática das bolhas

Numa descrição de um sistema como um diagrama DFD é comum a criação de documentos com várias páginas organizadas hierarquicamente. Cada página corresponde a um processo que recebe uma enumeração como é ilustrado abaixo:



Este problema da enumeração das "bolhas" é um problema de escopo. O escopo pode ser representado por um atributo associado a cada Processo. Este atributo pode ser um string composto de campos numéricos separados por pontos.

Por exemplo, no primeiro nível pode-se ter os processo "CADASTRA 0.1", "VERIFICA 0.3", etc.; no segundo nível pode-se ter os processos "CADASTRA-X 0.1.2", "VERIF1 0.3.1", "VERIF2 0.3.2", etc., como é ilustrado acima.

Esta enumeração pode ser definida por regras semânticas como mostra a figura 5.7. A cada nível é acrescido um "." e um dígito na enumeração do escopo; e, dentro de uma página, é feita a enumeração dos processos; os processos filhos herdam a enumeração do pai.

```
< inh STR scope :Sp, pL, P, !Pg, Osp; >

Dfd ---> Sp
  < Sp.scope = "0"; >

Sp  ---> !Name tL dL pL fL
  < pL.scope = Sp.scope; >

pL  ---> P pL
  < pL1.nro   = pL2.nro + 1;
    P.scope   = pL1.scope + "." + pL.nro;
    pL2.scope = pL1.scope; >

pl  ---> ^
  < pl.nro = 0; >

P  ---> !Pg Osp
  < !Pg.scope = P.scope;
    Osp.scope = !Pg.scope;

Osp ---> Sp
  < Sp.scope = Osp.scope; >
```

figura 5.7: a enumeração dos processos para o DFD

6 DIAGRAMA ENTIDADE RELACIONAMENTO (E-R)

Esta especificação do diagrama entidade relacionamento ilustra uma notação diagramática do tipo grafo que não possui formatação livre. A formatação dos nodos segue uma estrutura hierárquica, que deve estar representada na descrição da sintaxe.

Na descrição semântica é exemplificada a formatação do diagrama.

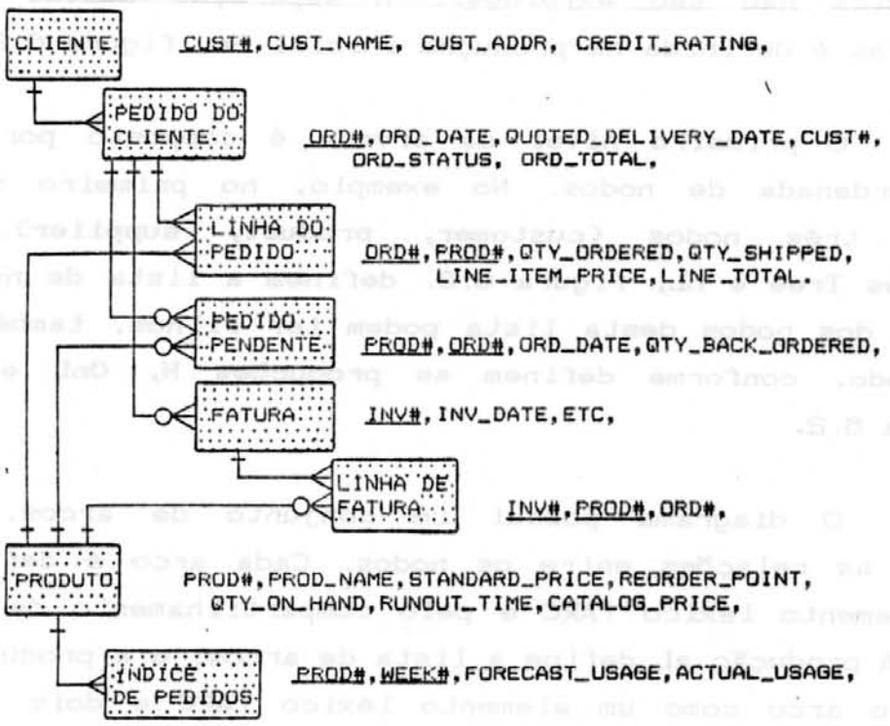


figura 6.1: variante do diagrama E-R. Fonte:[MAR 87].

A figura 6.1 ilustra uma variante, sugerida por [MAR 88], da notação E-R tradicional. O que diferencia esta variante do modelo tradicional de E-R é o conceito de hierarquia nas entidades. Por exemplo, a entidade order é

pai das entidades **order-line, backorder e invoice.**

A especificação dos léxicos é feita seguindo-se a sistemática apresentada no capítulo 3.

6.1 Especificação da sintaxe

A descrição da sintaxe deve considerar a estrutura de grafo, com nodos compartilhados e a estrutura hierárquica em forma de árvore (os arcos que ligam os nodos da estrutura hierárquica não são exibidos). A separação destas duas estruturas é definida na produção inicial **Gr**, figura 6.2.

O primeiro nível da árvore é composto por uma lista ordenada de nodos. No exemplo, no primeiro nível existem três nodos (**customer, product, supplier**). As produções **Tree e nL**, figura 6.2. definem a lista de nodos. Cada um dos nodos desta lista podem ter filhos, também do tipo nodo, conforme definem as produções **N, OnL e nL**, nafigura 6.2.

O diagrama possui um conjunto de arcos, que definem as relações entre os nodos. Cada arco é definido pelo elemento léxico **!ARC** e pelo compartilhamento de dois nodos. A produção **aL** define a lista de arcos; e a produção **A** define o arco como um elemento léxico **!ARC** e dois nodos compartilhados.

```
%Significado dos identificadores:
Tree: estrutura de árvore
nL : (node list) lista de nodos
OnL: (opcional node list) lista-de-nodos opcional
N  : (node) nodo não-terminal
A  : (arcs) arcos
aL : (arc list) lista de arcos %

Gr    --> Tree  Graph

Tree --> nL
N     --> !NODE OnL

OnL   --> nL
OnL   --> ^

nL    --> N nL
nL    --> ^

Graph --> aL

La    --> A aL
La    --> ^

A     --> !ARC ?NODE ?NODE
```

figura 6.2: E-R: grafo com formatação hierárquica

6.2 Especificação da formatação

As equações semânticas para a formatação são associadas às produções que definem a estrutura hierárquica.

Na especificação da formatação do diagrama E-R são utilizadas duas constantes: dy, dx. A constante dy define a distância entre dois nodos contíguos no sentido vertical, entre os cantos superiores esquerdos dos nodos; e, dx define a distância dos nodos no sentido horizontal, entre pais e

filhos; também em relação ao canto superior esquerdo dos nodos, figura 6.1.

O tamanho das "caixinhas" (!NODE) é definido pela constantes **unitWIDTH** e **unitHEIGTH.** Cada nodo terminal (!NODE) possui os atributos **x** e **y** que definem a posição do canto superior esquerdo da "caixinha" dentro da página. Cada nodo filho herda do pai o atributo **x** somado de um deslocamento **dx** (produção **N**, figura 6.3).

O atributo **brothers** armazena o número de irmãos somado ao número de filhos de cada irmão (produção **nL**). Dizendo em outras palavras: cada nodo filho envia para o pai o número de nodos filhos somados aos filhos dos filhos recursivamente. Este atributo é usado no cálculo da próxima posição na vertical entre dois nodos irmãos: "dy * (sons + 1)", produção **nl**, figura 6.3.

```
( Const unitWIDTH  = 12;
        unitHEIGTH = 4;)
        dx = unitWIDTH  + unitWIDTH / 3;
        dy = unitHEIGTH + unitHEIGTH / 4;

  inh INT x,y          : !NODE, nL, OnL;
  inh INT x1,y1,x2,y2  : !ARC;
  ext INT ex1,ey1,ex2,ey2,ex0 : !ARC;
  syn INT sons         : N, OnL;
  syn INT brothers     : nL; )

Gr  -->  Tree  Arcs

Tree --> nL
(   nL.x  = 0;
    nL.y  = 0; )

N    --> !NODE OnL
  ( N.sons   = OnL.sons;
    OnL.x    = N.x + dx;
    Onl.y    = N.y;
    !NODE.x = N.x;
    !NODE.y = N.y; )                          continua ...
```

```
OnL --> nL
  ( OnL.sons = nL.brothers;
    nL.x     = OnL.x;
    nL.y     = OnL.y; )

OnL --> ^
  ( OnL.sons = 0; )

nL --> N nL
  ( nL1.brothers = N.sons + 1 + nL2.brothers;
    N.x          = nL1.x;
    N.y          = nL1.y;
    nL2.x        = nL1.x;
    nL2.y        = nL1.y + dy * (N.sons + 1); )

nL --> ^
   (nL.brothers := 0; )

Arcs --> aL

aL  --> A aL
aL  --> ^

A  -->  !ARC  ?NODE  ?NODE
 ( !ARC.x1 = ?NODE1.x;
   !ARC.x2 = ?NODE2.x;
   !ARC.y1 = ?NODE1.y;
   !ARC.y2 = ?NODE2.y; )
```

figura 6.3: GA para a formatação do diagrama E-R

O léxico arco possui dois pares de atributos herdados, (x1, y1) e (x2, y2), que definem os valores das coordenadas dos nodos que são ligados pelo arco. Estas informações são utilizadas para restringir os valores dos atributos locais a1, a2 ao terminal!ARC que definem a posição exata do ponto da ligação (ver classe listada abaixo).

A classe listada abaixo define esta restrição, supondo que a ligação é feita sempre na lateral esquerda. O arco, similar aos da figura 6.1, é formado por 3 linhas

retas; neste arco os extremos são ligados sempre na parte esquerda da caixinha.

```
CONSTRAINT toLefthSide IS
  p1 : point;
  r1 : point;
CONSTRAINED BY
    ((p1.x = r1.x) and  (r1.y <= p1.y <= r1.y + unitHEIGTH))
END
CLASS arc IS
  ex0: integer;
  p1 : point = (x1, y1);
  p2 : point = (x2, y2);
  a1, a2 : point;
CONSTRAINED BY
  toLefthSide(a1,p1);
  toLefthSide(a2,p2);
SHOWN AS
  lineSeg((ex0,a1.y),(ex0,a2.y));
  lineSeg((ex0,a1.y),(x1,a1.y));
  lineSeg((ex0,a2.y),(x2,a2.y));
  arrow((ex0,a2.y),(x2,a2.y));
END
```

O atributo **ex0** define a coordenada **x** por onde passará a segmento vertical do arco. A seta, definida pela classe **arrow**, é direcionada do primeiro para o segundo ponto, é exibida sobre o segundo ponto.

Os valores **a1, ... ,a2** são definidos por operações de apontamento na tela. A restrição **toLefthSide** verifica se os pontos editados estão sobre a borda esquerda da "caixinha"; a borda da caixinha é calculada a partir do canto superior esquerdo e da constante **unitHEIGTH** que definem o seu tamanho vertical.

Uma solução um pouco mais sofisticada consiste em calcular automaticamente a posição de chegada ou de partida de um arco numa "caixinha", tendo-se por base o número de nodos que chegam ou partem do nodo. As informações necessárias para o cálculo poderiam ser coletadas em uma TS.

7 UM PROTÓTIPO DE UM GERADOR DE EDITORES DIAGRAMÁTICOS

Este capítulo define uma arquitetura para o protótipo do GED. Na definição da arquitetura busca-se a integração do GED com o GET atendendo a necessidade de:

i) facilidades de edição textual e diagramática em um mesmo ambiente;

ii) possibilidade de criação de um ambiente, no qual diversas notações compartilham informações em um banco de dados;

Neste trabalho discutiu-se o compartilhamento de atributos em tabelas relacionais. No entanto existem outras formas de compartilhamento de informações entre diferentes notações; por exemplo, uma forma seria usar a saída de uma ferramenta como a entrada para outra: a saída do DFD pode ser a entrada para o diagrama estruturado [PET 87]. Esta forma de compartilhamento não ocorre apenas a nível de atributos, mas também a nível de informações estruturais, como árvores sintáticas (AS). É necessário traduzir uma AS de uma notação para a AS da outra notação.

Na especificação de arquitetura é considerado o fato de se compartilhar informações a nível de estruturas sintáticas. Este é um dos temas de pesquisa no Projeto ADS.

Com o objetivo de dar uma visão geral dos sistemas GED e GET, os principais componentes da arquitetura proposta são detalhados a nível de módulos. O GED reutiliza os módulos que implementam o mecanismo de GA, desenvolvidos para o GET.

Os módulos relacionados com o formalismo gramatical que implementam o conceito de nodos compartilhados foram desenvolvidos exclusivamente para o GED e são especificados nos próximos capítulos.

7.1 O esquema funcional do GED

A figura 7.1 abaixo mostra o esquema funcional do GED. O GED consiste de dois sistemas principais: o gerador de tabelas e o editor baseado em tabelas (ED). O gerador lê a descrição de uma linguagem alvo em GA e gera as tabelas para o editor. O editor interpreta as tabelas geradas para distintos editores específicos.

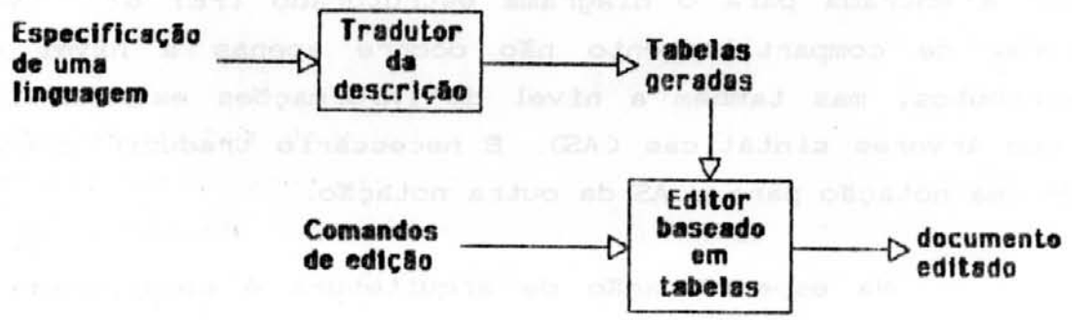


figura 7.1: o esquema do GED

Este esquema utilizado pelo GED é também utilizado pelo GET [ESP 89b]. Ambos os editores, ED (diagramático) e ET (textual), interpretam as tabelas geradas pelo tradutor da LDE.

A descrição da metalinguagem LDE (sem as facilidades de tabelas relacionais), bem como sua implementação (o tradutor) é encontrada em [ESP 89b].

7.2 Uma arquitetura para o ED/ET

Na definição da arquitetura dos editores baseados em tabelas ED/ET são considerados os níveis lógicos da metalinguagem LDE para compartilhamento de informações.

Os níveis lógicos da LDE

Como foi comentado no capítulo 2, a especificação de uma linguagem compreende três níveis lógicos: léxico, sintático e semântico. O compartilhamento de informações acontece nos níveis sintático e semântico, como ilustra a figura 7.2.

O nível sintático trata apenas das informações estruturais das linguagens. A integração neste nível faz uso de traduções entre diferentes notações, por exemplo a saída de uma ferramenta é a entrada para outra.

O nível semântico utiliza-se de dois formalismos para manipular as informações: o mecanismo de GA e de álgebra relacional.

Com o mecanismo de GA as informações semânticas das TSs são manipuladas como listas/funções. Para compartilhar informações entre diferentes ferramentas é necessário representar estas listas numa área de dados comum. As listas são necessárias para representação de

informações associadas pelo conceito de ordem de escrita.

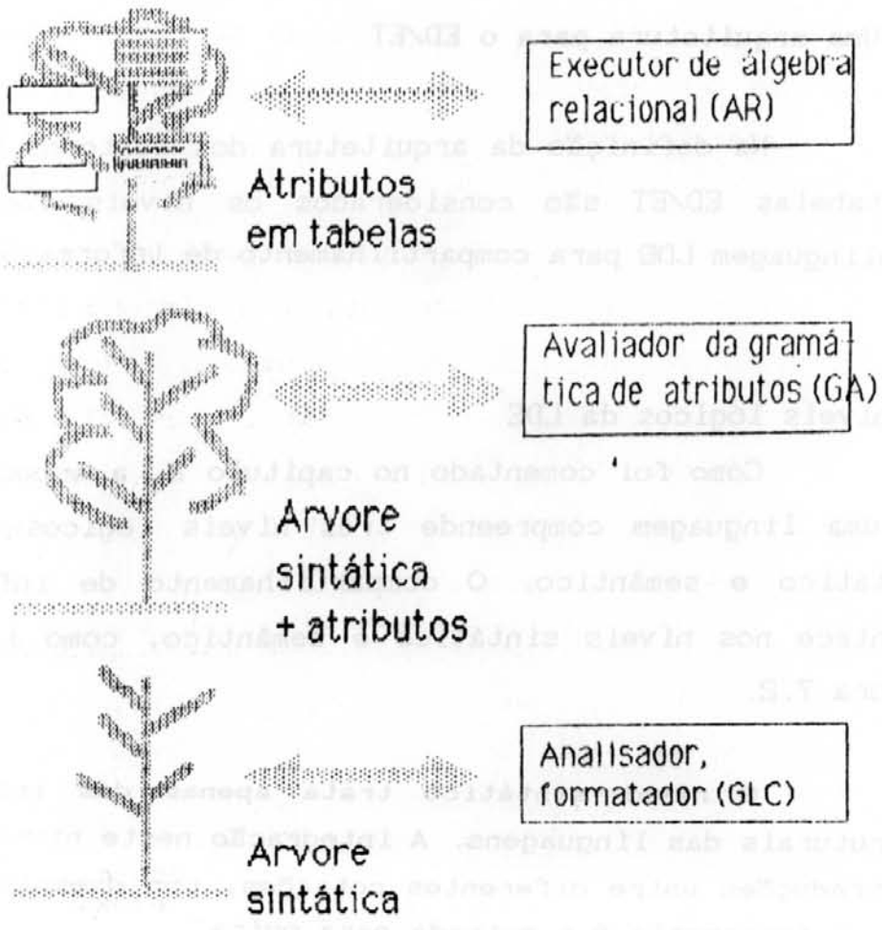


figura 7.2: níveis lógicos para compartilhamento de informações para a metalinguagem LDE

Uma abordagem alternativa é a representação de TSs por tabelas relacionais. Um exemplo típico de uso das tabelas relacionais é a representação de um Dicionário de Dados [MRA 89] que permite a referência cruzada entre informações de diferentes ferramentas: de dentro uma ferramenta é possível se fazer acesso às informações de outras ferramentas.

Uma arquitetura comum para o editor ED/ET

Com base nos níveis lógicos da LDE (léxico, sintático e semântico) e no compartilhamento de informações em um BD define-se a arquitetura do núcleo executor dos editores ED/ET (figura 7.3). O nome ED está associado ao gerador de editores GED e o nome ET está associado ao gerador de editores GET.

O nível léxico é distinto para os dois editores: léxico textual e léxico diagramático. A sintaxe abstrata e o mecanismo de GA são comuns para os dois editores. O sistema de álgebra relacional dá suporte para a criação de tabelas associadas à GA. Todas as informações (elementos léxicos, ASs, TSs como listas e tabelas relacionais) são representadas no BD global.

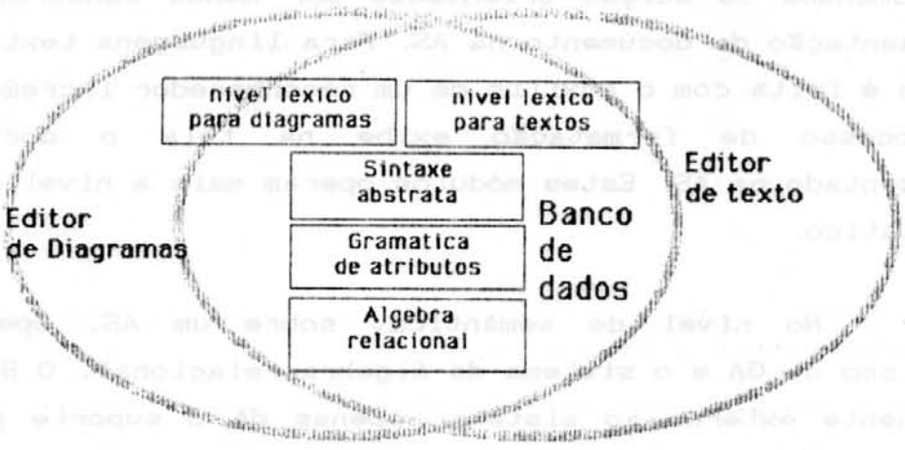


figura 7.3: Uma arquitetura comum para o ED/ET.

A arquitetura do ED/ET permite identificar as partes comuns direcionando os esforços de implementação nos componentes exclusivos do ED.

As principais diferenças entre o ED e o ET estão no nível da sintaxe. No protótipo são desenvolvidos somente os aspectos particulares do ED:

. extensão do GLC com o conceito de grafo (sintaxe abstrata)
. nível léxico para o ED (editor de elementos léxicos)

A nível de implementação os componentes lógicos da arquitetura são detalhados em módulos funcionais. O item que segue pretende dar uma visão geral destes módulos funcionais.

7.3 Os componentes funcionais do ED/ET

Sob o ponto de vista operacional o ED/ET executa duas principais funções: a edição e a formatação. A partir dos comandos de edição orientados por menus constrói-se a representação do documento na AS. Para linguagens textuais a edição é feita com o auxílio de um reconhecedor incremental. O processo de formatação exibe na tela o documento representado na AS. Estes módulos operam mais a nível léxico e sintático.

No nível de semântica, sobre um AS, operam o mecanismo de GA e o sistema de álgebra relacional. O BD é um componente externo ao sistema: apenas dá o suporte para o armazenamento e recuperação das informações utilizadas pelos componentes do sistema.

Abaixo está ilustrado o primeiro nível de decomposição do sistema ED/ET.

0.1 módulo de edição
0.2 formatador
0.3 mecanimo de GA
0.4 álgebra relacional
0.5 banco de dados

ED/ET

Nos próximos ítens descreve-se brevemente cada um dos componentes, seu estado atual de desenvolvimento no projeto ADS e suas referências.

7.4 O módulo de edição

Para um editor textual existem duas formas de interação com o usuário. Na primeira o texto é analisado por um reconhecedor incremental a medida que vai sendo digitado. Na segunda, a edição é orientada por menus e gabaritos, i.e. a edição consiste na seleção de menus e preenchimento de gabaritos. Há sistemas onde o usuário pode passar de uma forma para outra [SNE 85] e [REP 87].

A edição de linguagem diagramáticas é orientada por menus. O programa que controla a edição executa dois tipos de operações: as operações estruturais e as não estruturais. As operações estruturais modificam a estrutura sintática do documento, enquanto que as não estruturais modificam os atributos dos elementos léxicos. As operações estruturais e não estruturais são detalhadas, respectivamente, nos capítulos 8 e 9.

Componentes do módulo de edição

O módulo de edição é decomposto nos seis componentes apresentados acima. Segue uma breve descrição destes componentes.

1.0 reconhecedor incremental
1.1 editor baseado em menus
1.2 comandos de navegação
1.3 interface com GA
1.4 editor de texto
1.5 editor de léxicos

reconhecedor

1.0 reconhecedor incremental: implementa um reconhecedor de texto. O usuário digita livremente o texto que é analisado a medida que vai sendo digitado. O texto reconhecido é incluído na AS.

> Para a execução da análise de forma incremental foi desenvolvido para o Projeto ADS um reconhecedor incremental [ESP 89a]. Este reconhecedor aceita como entrada uma gramática do tipo LL(1).

1.1 editor baseado em menus: implementa um editor baseado em menus e gabaritos, podendo ser utilizado tanto para linguagens textuais como para linguagens diagramáticas.

> Nos capítulos 8 e 9 descreve-se os algoritmos que implementam este módulo para o protótipo do GED.

1.2 comandos de navegação: este módulo é responsável pela execução dos comandos que movimentam o cursor.

> A implementação deste módulo para linguagens textuais é descrita em [FAV 88]. No capítulo 9 comenta-se o apontamento de objeto gráficos, o que corresponde ao movimento do cursor para linguagens textuais.

1.3 interface com GA: informa ao avaliador da GA quais os atributos que foram incluídos ou modificados na AS, a fim de que possa ser feita a reavaliação das equações semânticas, após uma operação de edicão.

Nos trabalhos [KOL 88] e [ESP 89b] é comentada a construção deste módulo.

1.4 editor de texto: implementa um editor de texto; este editor é utilizado para editar os léxicos textuais ou atributos do tipo texto.

1.5 editor de léxicos: implementa as funções de manipulação dos objetos léxicos do tipo gráfico: seleção e edição.

No capítulo 3 mostrou-se uma linguagem para a descrição dos léxicos; está em andamento a implementação de uma versão desta linguagem no Projeto ADS.

Nos capítulos que seguem são apresentados algoritmos para a seleção dos elementos léxicos na tela.

7.5. O formatador

Para linguagens textuais o formatador exibe o texto formatado na tela. Normalmente o formatador para um editor orientado por estruturas para linguagens textuais é especificado por regras associadas a GLC (ver por exemplo [REP 87]).

Um breve comentário sobre a implementação de formatadores para o ET é encontrado em [FAV 88] e [ESP 89a].

Para as linguagens diagramáticas, como formatação livre (o usuário move livremente os nodos), o processo de

formatação consiste em exibir na tela os elementos léxicos na posição em que foram criados. É o caso do DFD, capítulo 5.

Para as notações diagramáticas de formatação livre pode-se oferecer também facilidades de formatação semi-automática: inicialmente o documento é formatado com base numa heurística de "placing and routing" [SIL 89] e numa segunda etapa são providas operações para se modificar livremente a formatação.

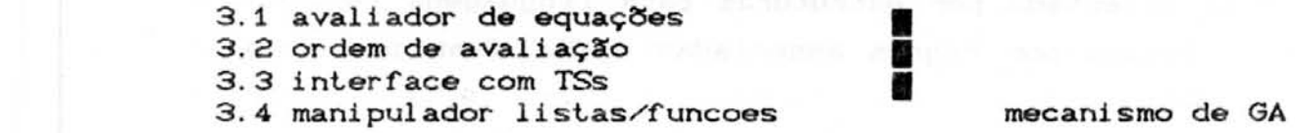
Para as notações diagramáticas que não possuem formatação livre pode-se especificar na LDE a formatação, como foi exemplificado com o diagrama de N-S no capítulo 4.

> Nas linguagens diagramáticas a formatação é executada por um editor de elementos léxicos, que pode consultar os valores dos atributos calculados pela GA.

7.6 O mecanismo de GA

O mecanismo de GA é responsável pela execução da análise semântica, bem como pela avaliação de atributos que são utilizados na formatação.

Este módulo, **mecanismo de GA**, é decomposto em três submódulos, que são brevemente descritos abaixo:



3.1 avaliador de equações: implementa um algoritmo que executa as equações semânticas;

3.2 ordem de avaliação: um algoritmo que define a ordem de execução das equações;

Uma descrição detalhada do avaliador de equações e do algoritmo que define a ordem de avaliação é encontrada nos trabalhos [KOL 88] e [ESP 89b]. Para implementação da interface com as TSs a bibliografia sugerida é [HOR 86], [REP 87], [HOO 87].

3.3 interface com TSs: define a interface entre o avaliador das equações semânticas e as tabelas relacionais, permitindo ao avaliador manter constantemente atualizados os atributos representados nas tabelas.

3.4 manipulador de listas/funções

Este módulo implementa uma linguagem do tipo funcional para manipular as listas que representam as TSs. No capítulo dois ilustrou-se o uso de uma linguagem funcional para a descrição da TS para a minilinguagem de expressões.

> Na implementação atual da LDE estas funções são codificas em Pascal e ligadas (compiladas) junto com o editor. A forma como se faz a codificação é comentada em [ESP 89b]

7.7 **Executor de álgebra relacional**

O executor de álgebra relacional é decomposto em um executor de consultas, um módulo gerenciador do catálogo de tabelas e um módulo que faz a interface com o mecanismo de GA.

O.1 executa consultas
O.2 mantêm o catálogo
O.3 interface com GA

álgebra
relacional

1.0 álgebra relacional: Este módulo implementa um sistema relacional acoplado ao mecanismo de GA. As linguagens de definição de dados (DDL) e de manipulação de dados (DML) foram apresentadas nos capítulos 3-6. A notação da DML é baseada em [DAT 84]. Nas especificações do capítulos 3-6 mostrou-se que é possivel juntar expressões descritas na DML com as equações da GA.

Este módulo foi decomposto em três submódulos:

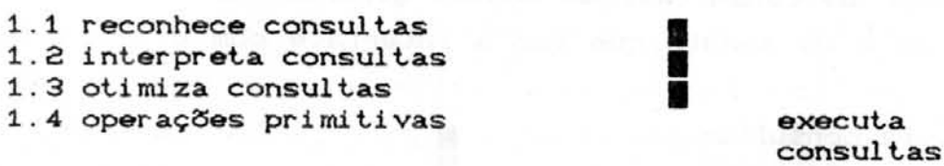
1.1 executa consultas: executa consultas sobre as informações das TSs. Este módulo é detalhado abaixo.

1.2 mantém o catálogo: gerencia a criação e remoção de tabelas; alocação de áreas, etc.

1.3 interface com GA: trata dos problemas relacionados com a atualização de atributos em tabelas.

Executor de consultas

O executor de consultas compreende um compilador para reconhecer consultas e definições de tabelas. Uma definição de uma tabela gera uma entrada no catálogo das tabelas. A partir de uma consulta é gerado um código intermediário, o qual é processado por um interpretador. O interpretador executa o código chamando funções primitivas da álgebra relacional.



A extensão do mecanismo de GA por tabelas relacionais foi sugerida por [HOR 86]. Na arquitetura para o ED o conceito de tabelas tem duas funções principais:

i) executar as verificações internas ao formalismo de GA;

ii) executar as verificações externas ao formalismo de GA;

A construção do executor de álgebra relacional está em andamento no Projeto ADS. Esta versão inicial do executor de álgebra relacional opera somente em memória, com rotinas para salvar as tabelas em disco. As TSs são usualmente pequenas (algumas dezenas de Kbytes) o que permite a sua representação na memória durante o uso.

Numa segunda versão deve-se integrar este executor de álgebra relacional com um BD, onde estarão armazenadas todos as informações de um ADS.

7.8 O Banco de dados

Este módulo dará suporte ao gerenciamento de todas as informações produzidas pelo sistema: árvores sintáticas, tabelas de símbolos, representação das gramáticas, etc.

No compartilhamento de informações entre diferentes editores devem ser providas operações que permitam manipular as tabelas de símbolos representadas como funções/listas ou tabelas relacionais, como foi exemplificado no capítulo 2.

As facilidades para implementação de TSs são fortemente ligadas ao mecanismo avaliador de GA, pois são as ações implícitas do avaliador que mantêm os valores de

atributos constantemente atualizados, através das equações semânticas.

O estudo e implementação do BD é um tema de pesquisa em aberto.

8 OPERAÇÕES ESTRUTURAIS

A idéia deste capítulo, bem como do próximo, é de documentar a implementação do protótipo do GED. Esta documentação visa dar suporte ao desenvolvimento futuro do GED; por exemplo, para acrescentar novas facilidades basta anexar as novas operações às operações descritas aqui.

Os documentos são representados internamente na árvore sintática decorada com atributos. Parte da informação é representada na estrutura sintática (informação estrutural) e parte como atributos associados à estrutura sintática (informação não estrutural).

Na edição do documento (atualização da representação interna) são considerados dois tipos de operações: as estruturais e as não estruturais. As operações estruturais modificam a estrutura sintática do documento. E as não estruturais (capítulo 9) modificam os atributos associados à estrutura sintática, que descrevem os aspectos semânticos.

As operações estruturais, por serem dirigidas pela sintaxe (GLC estendida pelo conceito de nodos compartilhados), preservam a consistência da AS. Estas operações são guiadas pelas regras abaixo (baseadas na classificacão das produções, capítulo 5):

- para estruturas tipo seleção os elementos a serem exibidos no menu são as possíveis estruturas alternativas;
- uma estrutura tipo opcional pode ser incluída ou excluída livremente;
- numa estrutura tipo lista pode-se incluir ou excluir livremente (em qualquer ponto da lista) qualquer número de elementos;
- no tipo agregação, se um de seus elementos é excluído

então toda a estrutura é excluída; na criação de uma nova ocorrência deste tipo todos os elementos constituintes devem ser expandidos;

- quando uma estrutura do tipo elementar compartilhável é removida, remove-se juntamente todos os nodos usuários.

As operações estruturais são codificadas a partir das operações de manipulação da AS que são implementadas pelos tipos abstratos de dados **TpASA** e **TpCursor**. Estes tipos foram originalmente definidos para o GET [ESP 89a] e são listados no apêndice A, a fim de possibilitar uma leitura rápida dos algoritmos descritos para o GED.

A descrição dos algoritmos segue o método VDM [JON 80], [JON 86]; as noções básicas necessárias para a leitura dos algoritmos são, também, apresentadas no apêndice A.

8.1 Operações estruturais

Nos próximos ítens é detalhado o processo de geração de menus. São ilustrados os conceitos relacionados com os diversos tipos de ítens de menus (obrigatórios / opcionais, de criação / de compartilhamento) associados ao estado da árvore sintática que representa o documento sendo editado.

Os conceitos sobre a geração de menus foram agrupados nos ítens que seguem:

i) menu de elementos para inclusão
   - elementos opcionais
   - elementos obrigatórios
     . só uma opção (expansão automática)

- elementos para compartilhamento

ii) menu de elementos para remoção

```
 1. Dfd --> Sp
 2. Sp  --> !Ng tL dL pL fL
 3. tL  --> !Tg tL
 4. tL  --> ^
 5. dL  --> !Dg dL
 6. dL  --> ^
 7. pL  --> P  pL
 8. pL  --> ^
 9. fL  --> F fL
10. fL  --> ^
11. F   --> !Ag Nd Nd
12. Nd  --> ?Tg
13. Nd  --> ?Pg
14. Nd  --> ?Dg
15. P   --> !Pg Osp
16. Osp --> Sp
17. Osp --> ^
```

figura 8.1: gramática do DFD utilizada nos exemplos deste capítulo.

Os exemplos apresentados neste capítulo são baseados na gramática do DFD introduzida no capítulo 5, figura 8.1.

8.2 Menu de elementos para inclusão

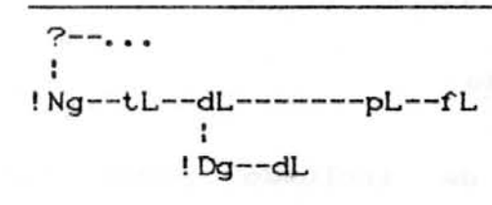
Os itens de menus de inclusão podem ser classificados em opcionais (tipo lista ou opcional) ou obrigatórios (tipo agregação); por exemplo a inclusão de um depósito (!Dg) é opcional (figura 8.1, produções 5-6). No entanto, após a criação do nome de um fluxo (!Ag) se torna obrigatório (produções 11-14) o apontamento de dois elementos (Nd).

Caso existirem ítens obrigatórios, somente estes serão exibidos. Caso contrário são exibidos os ítens opcionais. A prioridade para a edição de elementos obrigatórios visa manter a consistência sintática da estrutura de árvore em relação à gramática. A árvore sintática está num estado consistente se o menu gerado é composto apenas de ítens opcionais.

Se o usuário cancela uma operação de inclusão de ítens obrigatórios o sistema desfaz a(s) última(s) inclusões retornando ao estado consistente do último menu de ítens opcionais.

8.2.1 Inclusão de elementos opcionais

Na figura 8.2 é mostrado o menu associado ao estado da árvore. Todos os ítens deste menu são opcionais. Selecionando-se a linha 3 para incluir um Processo (!Pg), tem-se a árvore da figura 8.3.



Inclusão

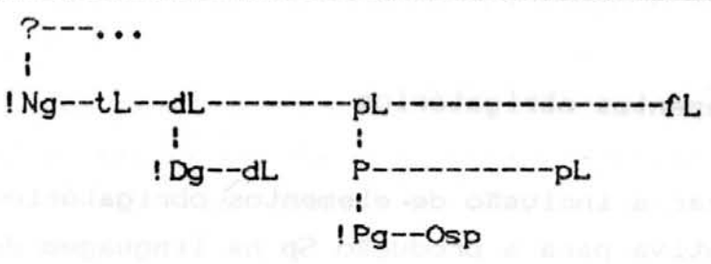
	cursor	produção	terminal
1	IN	tL	!Tg
2	INNIN	dL	!Dg
3	INNN	pL	!Pg
4	INNNN	fL	!Ag

figura 8.2: menu de elementos opcionais

O cursor representado por INNIN (<in, next, next, in, next>) deve ser lido como <desce, próximo, próximo,

próximo, desce, próximo>. Onde "desce" significa descer de um nodo pai para o filho mais à esquerda e "próximo" significa vai para o irmão que está à direita.

A passagem do estado da figura 8.2 para o estado da figura 8.3 ilustra a inclusão de um elemento opcional.



Inclusão

	cursor	produção	terminal
1	IN	tL	!Tg
2	INNIN	dL	!Dg
3	INNNIIN	Osp	!Ng
4	INNNIN	pL	!Pg
5	INNNN	fL	!Ag

figura 8.3: inclusão de um processo

A explosão de um processo é opcional, i.e. a produção Osp é do tipo opcional. Para a explosão do processo seleciona-se a linha 3 do menu, resultando na árvore da figura 8.4.

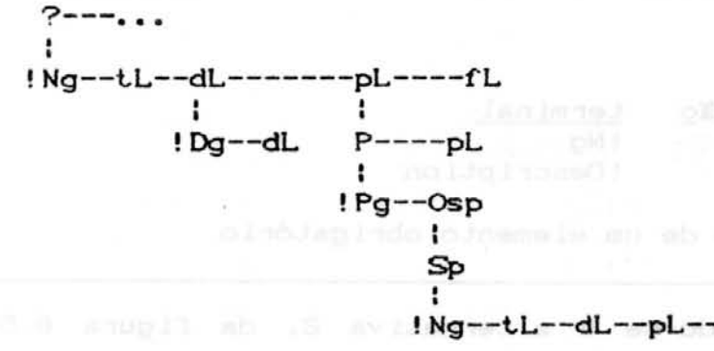


figura 8.4: explosão de um processo

Externamente, a nível de edição, a estrutura acima (figura 8.4) corresponde a duas páginas de um DFD, a segunda página é o refinamento de um processo da primeira. As operações para tratamento de páginas são apresentadas no capítulo 10.

8.2.2 Inclusão de elementos obrigatórios

Para ilustrar a inclusão de elementos obrigatórios define-se uma alternativa para a produção Sp na linguagem do DFD, como segue:

```
2.   Sp --> !Ng tL dL pL fL
2.1  Sp --> !Description
```

Leia-se: um subprocesso (Sp) é um diagrama (produção 1) ou uma descrição textual (produção 2). A figura 8.5, abaixo, ilustra a árvore associada ao menu gerado pela produção Sp.

```
?---...
:
Sp

Inclusão
    cursor      produção    terminal
1       I       Sp          !Ng
2       I       Sp          !Description
```

figura 8.5: inclusão de um elemento obrigatório

Selecionando-se a alternativa 2, da figura 8.5, acima, obtém-se a figura 8.6, abaixo.

```
?
!
Sp
!
!Description
```

figura 8.6: inclusão de uma alternativa

8.2.3 Expansão automática da estrutura da árvore

A expansão automática da estrutura da árvore ocorre quando se tem apenas uma opção no menu de ítens obrigatórios.

A gramática inicial do DFD, figura 8.1, possue apenas uma alternativa para a produção Sp. A figura 8.7 mostra o estado da árvore associada ao menu de opções de inclusão, onde se tem apenas uma opção. Neste caso o sistema automaticamente inclui a opção única na árvore, figura 8.8.

```
?---...
!
Sp

Inclusão
    cursor     produção   terminal
1      I       Sp         !Ng
```

figura 8.7: menu com apenas uma opção

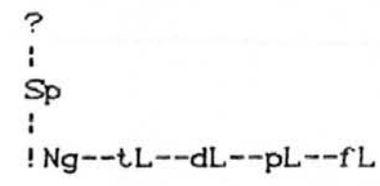


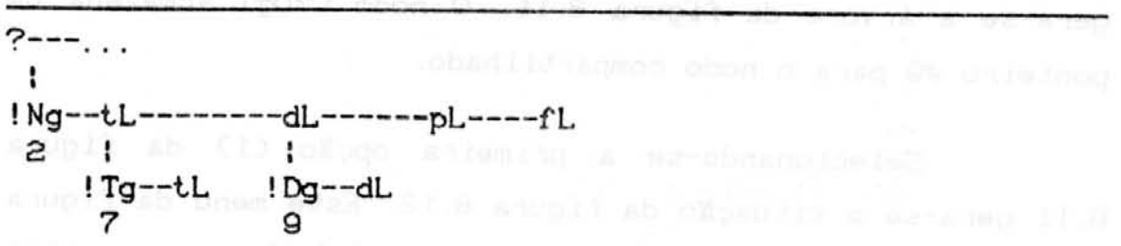
figura 8.8: inclusão automática

8.2.4 Menu de elementos de apontamento

Nas figuras utilizadas para ilustrar este item, são mostrados os identificadores internos (Id) dos nodos folha do tipo terminal (!) e os "ponteiros" dos nodos donatários (?); os ponteiros são prefixados pelo caracter #, ver figuras 8.10 e 8.12.

Nos exemplos que seguem é ilustrada a operação que compartilha nodos - são criadas estruturas com terminais dos tipos (?). Estes exemplos são mostrados sobre a estrutura de uma página parcialmente editada, com uma Entidade (!Tg) e um Depósito (!Dg), figura 8.9. Selecionando-se o item 4 da figura 8.9, a inclusão do fluxo, a árvore toma a forma da figura 8.10.

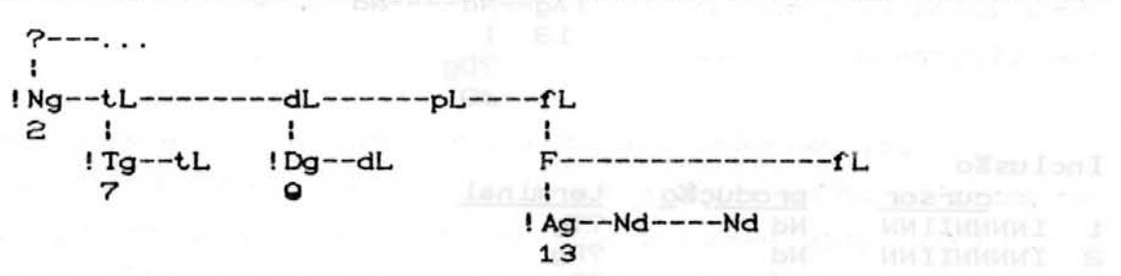
Este menu, figura 8.10, possui elementos gramaticalmente obrigatórios - os nodos **Nd.** Caso o usuário cancele a inclusão, o sistema automaticamente desfaz esta última inclusão retornando ao estado consistente da figura 8.9.



Inclusão

	cursor	produção	terminal
1	IN	tL	!Tg
2	INNIN	dL	!Dg
3	INNN	pL	!Pg
4	INNNN	fL	!Ag

figura 8.9: menu associado a árvore



Inclusão

	cursor	produção	terminal
1	INNNNIIN	Nd	?Tg
2	INNNNIIN	Nd	?Pg
3	INNNNIIN	Nd	?Dg
4	INNNNIINN	Nd	?Tg
5	INNNNIINN	Nd	?Pg
6	INNNNIINN	Nd	?Dg

Apontamento

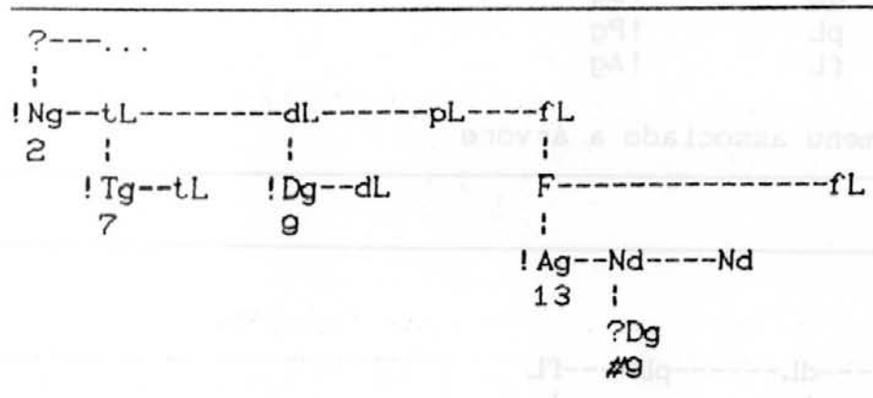
	cursor	terminal	Id
1	INI	!Tg	7
2	INNI	!Dg	9

figura 8.10: menu com ítens de apontamento

Na tela são exibidas somente as possibilidades de apontamento, para os nodos terminais com **Id=7** e **Id=9**. A seleção da linha 2 do menu de apontamento da figura 8.10

gera-se a árvore da figura 8.11. O nodo (?Dg) armazena um ponteiro #9 para o nodo compartilhado.

Selecionando-se a primeira opção (1) da figura 8.11 gera-se a situação da figura 8.12. Este menu da figura 8.12 voltou a ser o mesmo da figura 8.9 (com as mesmas opções).



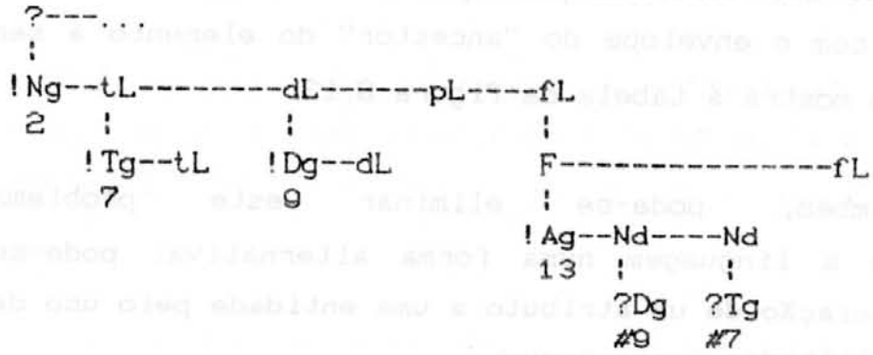
Inclusão

	cursor	produção	terminal
1	INNNNIINN	Nd	?Tg
2	INNNNIINN	Nd	?Pg
3	INNNNIINN	Nd	?Dg

Apontamento

	cursor	terminal-Id	
1	INI	!Tg	7
2	INNI	!Dg	9

figura 8.11: menu de apontamento e árvore com ligações



Inclusão

	cursor	produção	terminal	ocorrências
1	IN	tL	!Tg	1
2	INNIN	dL	!Dg	1
3	INNN	pL	!Pg	1
4	INNNN	fL	!Ag	1

figura 8.12: menu opcional e árvore com ligações

8.2.5 Problema da inclusão com ambigüidade

Em algumas linguagens pode ocorrer ambigüidade na definição do nodo, na AS, onde o elemento deve ser incluído.

Supondo uma página de um diagrama entidade relacionamento (E-R) parcialmente editada, com mais de uma entidade. Em cada uma das entidades pode ser incluído um atributo. No entanto no menu de opções aparece o terminal !Attr como a única opção de inclusão.

O problema está em determinar em qual das entidades o usuário deseja incluir o atributo selecionado. Este problema está ilustrado na figura 8.13.

Quando é detectado este tipo de ambigüidade solicita-se ao usuário um apontamento para selecionar a

entidade na qual o atributo deve ser associado. A posição de inclusão pode ser definida pela proximidade da posição do apontamento com o envelope do "ancestor" do elemento a ser criado, como mostra a tabela da figura 8.13.

Também, pode-se eliminar este problema reescrevendo a linguagem numa forma alternativa: pode-se fazer a associação de um atributo a uma entidade pelo uso de nodos compartilhados, como segue:

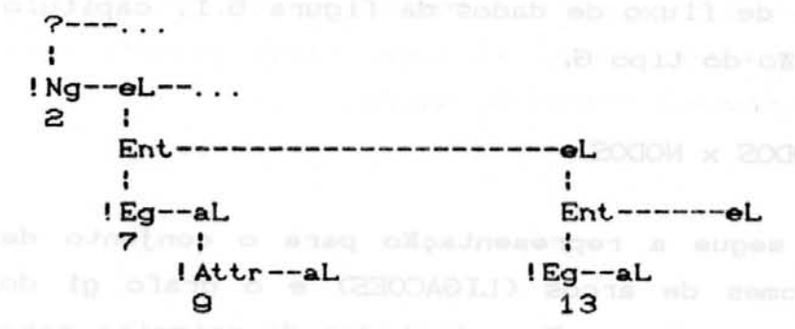
```
aL          --> Attribute  aL
aL          --> ^
Attribute   --> ?Ent !Attr
```

Nesta descrição um atributo é associado a uma entidade; na edição é necessário apontar para a entidade na qual o atributo será associado. Se a entidade compartilhada for removida o atributo é removido também.

gramática

```
      % abreviações
      Eg  : entity-graphic
      Ent : entity
      Attr: attribute
      aL  : attribute list
      Ng  : name-graphic
      eL  : entities list%

      ER   --> !Ng eL
      eL   --> Ent eL
      eL   --> ^
      Ent  --> !Eg  aL
      aL   --> !Attr  aL
      aL   --> ^
             ...
```



Inclusão

	cursor	produção	terminal	ancestor-Id
1	INIININ	aL	!Attr	!Attr 9
2	ININIIN	aL	!Attr	!Eg 13
3	...			

figura 8.13: inclusão com ambiguidade

8.2.6 Algoritmos para a estrutura de grafo e geração de menus

Neste item, inicialmente, é apresentada uma extensão do tipo TpASA descrito no Apêndice A. É necessário ler este Apêndice para compreeder a representação de

estruturas de grafo. Em seguida são apresentados os algoritmos de geração de menus.

Conceito de grafo

Um grafo qualquer, de nodos do tipo inteiro, pode ser representado por uma relação G, onde

G ⊆ **Nat0 x Nat0**
g ≡ { <i,j> ¦ igj }

Como exemplo de grafo pode-se ter

g1 = { <1,2> <1,3> <3,4> <4,1> }

Uma representação mais detalhada pode incorporar um nome para cada arco. Por exemplo, pode-se representar o grafo do diagrama de fluxo de dados da figura 5.1, capítulo 5, como uma relação do tipo G.

G ⊆ LIGACOES x NODOS x NODOS

Abaixo segue a representação para o conjunto de nodos (NODOS), nomes de arcos (LIGACOES) e o grafo g1 do tipo G. Assume-se que arcos são orientados do primeiro para o segundo nodo.

LIGACOES = <a, c, d, e, f, g, nil>
NODOS = <e1,e2,p1,p2,p3,p4,d1,d2,...>

g1 = { <a,e1,p2>, <c,p2,p1>, <d,p1,e2>, <d1,nil, nil>,
<g,e1,e2>, <e,p3,p1>, <f,p3,p4> }

Para o editor diagramático esta representação é modelada sobre a estrutura da árvore sintática. Na árvore sintática são necessários 3 nodos para representar <a, e1, p2>.

<mk-Node(a,!Ag,_),mk-Node(_,?Tg,#e1), mk-Node(_,?Dg,#p2)>

onde existem os nodos mk-Node(e1,!Tg,_)
mk-Node(p2,!Dg,_)

Esta representação pode ser vista na figura 8.12, onde o nodo !Ag é pai dos nodos ?Tg e ?Dg. Nesta figura, ao invés dos nomes "a","e1","p2" aparecem os números 13,9,7, como ilustrado abaixo.

<mk-Node(13,!Ag,_),mk-Node(_,?Tg,#9), mk-Node(_,?Dg,#7)>
mk-Node(7,!Tg,_)
mk-Node(9,!Dg,_)

Domínio de dados para grafos

Na definição do domínio de dados é necessário introduzir um invariante com as seguintes restrições: 1) toda a ligação deve ocorrer entre símbolos compatíveis, por exemplo uma ligação entre ?Ag e !Ag é válida, porém entre ?Tg e !Ag não; 2) Não se pode criar ponteiros para nodos inexistentes.

```
Symbs : NonTer U Ter U {^}
TpASA = Node ! emptyASA

Node :: SYMB : Symbs
        IN   : TpASA
        NEXT : TpASA
        ID   : Integer
        PTR  : Integer

Sys :: A : TpASA
       Ci: TpCursor
       Cf: TpCursor
     LAST : Integer;

 with inv-S(mk-Sys(a,c1,c2,l) ≡
      isvalid(c1,a) ∧ isvalid(c2,a)
    ∧ (∀i,j∈TpCursor)(i≠j → ID(asA(i,a))≠ID(asA(j,a)))
    ∧ isvalidGraph(a)

/* a relação entre os cursores e a árvore é válida, e
   não existem dois nodos com o mesmo identificador, e
   o grafo é válido
*/
```

```
isvalidGraph(a) ≡
   (∀i∈TpCursor)( (∃j,Ptr(asA(i,a))=ID(asA(j,a))
                  ∨ (Ptr(asA(i,a))=Nulo) )
    ∧ (∀i,j∈TpCursor)(Ptr(asA(i,a))=ID(asA(j,a)
                  ⇒ str(asA(i,a))=str(asA(j,a)))

/* não existem ponteiros com valor inválido
   e a ligação ocorre entre terminais de mesmo nome
*/
```

Na descrição dos algoritmos são utilizadas três funções auxiliares **Symb, Id, Ptr,** que retornam respectivamente, o símbolo, o identificador do nodo e o ponteiro do nodo.

```
type Symb: TpCursor x TpASA --> Symbs
type Id  : TpCursor x TpASA --> Integer
type Ptr : TpCursor x TpASA --> Integer

Symb(c,a) ≡ SYMB(asA(c,a)) /* letra maiúscula denota */
Ptr (c,a) ≡ PTR (asA(c,a)) /* seletor de campo  */
Id  (c,a) ≡ ID  (asA(c,a))
```

Identificador único

Para satisfazer o novo invariante é necessário estender a operação que cria nodos na estrutura da árvore sintática (mk-Node()). Para cada nodo é atribuído um valor como identificador único; isto é feito por uma função geradora de identificadores únicos **getId.**

```
initLast --> LAST
pos-initlast((),i) = i=0

gerId: LAST  --> LAST x Integer
pos-gerId(l,lf,i) = (lf=l+1,i=lf)
```

Na implementação real, o identificador único pode ser o endereço de memória onde é alocado o nodo, se não existe sobreposição de nodos. Se isto não for possível a solução mais simples consiste em redefinir a operação

mk-Node por uma mk1-Node, chamando-se a função **genId** para gerar o identificar do nodo.

mk1-Node(s,i,n,id,ptr) ≡ mk-Node(s,i,n,getId(Last),ptr)

Inserção do ponteiro (Ptr)

A inclusão de um nodo do tipo grafo é feita pela atribuição do valor do identificador do nodo a ser apontado no campo do nodo que aponta (SetPtr).

```
type SetPtr: TpCursor x TpASA x Integer --> TpASA
     SetPtr(c,a,i) ≡ μ(asA(c,a),[PTR --> i])

/* a função μ substitui um campo na strutura do nodo */
```

A inclusão dos nodos de criação é executada pela operação que inclui nodos na árvore sintática (InsertProd).

Gramática para grafos

Para estender a gramática livre de contexto com o conceito de grafo, introduziu-se uma classificação para os símbolos terminais e não-terminais. Esta classificação é exemplificada a partir da gramática da figura 8.1.

```
NonTer   = < Dfd, Sp, tL, dL, fL, F, Nd, P, Osp >
Ter      = < ?Tg, ?Pg, ?Dg, !Pg, !Tg, !Dg, ^ >
First    = < <tL,<!Tg>>, <Nd,<?Pg,?Dg,?Tg>, ...>
```

O conjunto dos terminais é subdividido em dois outros conjuntos. O conjunto dos símbolos que compartilham (Shareds) nodos criáveis e o conjunto dos nodos criáveis (Creates).

Shareds = ⟨ ?Tg, ?Pg, ?Dg ⟩

Creates = ⟨ !Pg, !Tg, !Dg ⟩

A partir dos não-terminais também são criados outros dois sub-conjuntos. O conjunto dos símbolos opcionais (Optionals) e o conjunto dos símbolos do tipo lista (List).

Optionals = ⟨ tL, dL, pL, fL, Osp ⟩

List = ⟨ tL, dL, pL, fL ⟩

Com estas informações pode-se estender o invariante para a estrutura da gramática, como está esboçado abaixo.

```
isValidGr1() ≡
      NonTer ∩ Ter = ⟨⟩ ∧
      isLL1() ∧
      Shareds ⊆ Creates ⊆ Ter ∧ Shareds ≠ ⟨⟩
      Lists ⊆ Optionals ⊆ NonTer ∧ Optionals ≠ ⟨⟩

/* um não-terminal não pode ser também terminal, e
   a gramática é do tipo LL(1), e
   para cada terminal compartilhável deve existir um
   correspondente de criação; obs: o símbolo ⊆ significa que
   os nomes sem o prefixo (!,?) são os mesmos;
   além disso deve existir algum símbolo do tipo
   compartilhado;
   os não-terminais do tipo lista estão contidos nos
   não-terminais do tipo opcional; além disso deve existir
   algum símbolo opcional. */
```

Com base na gramática exemplificada acima define-se as seguintes operações:

```
type _________: TpCursor x TpASA --> Boolean
     isShared  (c,a) ≡ Symb(c,a) ∈ Shareds
     isCreate  (c,a) ≡ Symb(c,a) ∈ Creates
     isOptional(c,a) ≡ Symb(c,a) ∈ Optional
     isList    (c,a) ≡ Symb(c,a) ∈ List
```

```
isNonTer  (c,a) ≡ Symb(c,a) ∈ NonTer
isTer     (c,a) ≡ Symb(c,a) ∈ Ter
```

Nestas definições fica implícita a consulta aos valores dos conjuntos globais (Shares, Creates, etc.).

A geração dos menus

Os menus são criadas a partir de diversos conjuntos de tuplas <cursor e símbolo> que são exibidos na tela para seleção. A informação do cursor é usada para se fazer a inclusão da opção selecionada na posição adequada (que gerou a opção) da AS.

```
Item    : TpCursor x Symbs
BagMenu : Item-set

type _________: TpASA --> BagMenu
BagLeafs (a)  ≡ { <c,symb(c,a)> | c∈TpCursor, isLeaf(c,a)  }
BagShareds(a) ≡ { <c,symb(c,a)> | c∈TpCursor,isShared(c,a) }
BagCreates(a) ≡ { <c,symb(c,a)> | c∈TpCursor, isCrate(c,a) }

BagOptionals(a) ≡
        { <c,symb(c,a)> | c∈TpCursor, isOptional(c,a) }

BagObligates(a) ≡ BagLeafs(a) - BagOptionals(a)
```

Estas operações implementam os menus gerados nas situações exemplificadas nos itens 8.2.1-8.2.5:

BagOptionals: figuras 8.2, 8.3, 8.9,...

BagObligates: figuras 8.5, 8.7, 8.10, ...

BagShareds: de apontamento, figuras 8.10, 8.11, ...

Os itens dos menus são classificados em dois grupos: elementos compartilháveis e de criação. Em cada

grupo podem existir elementos opcionais e obrigatórios. Na inserção, os ítens obrigatórios têm prioridade sobre os opcionais, como ilustra o esquema da função **exibe.**

```
type Exibe: BagMenu x BagMenu --> Item
     Exibe(obl, opt) ≡
        if obl≠() ∧ card(obl)=1
           then  /* insercao automatica x ∈ obl */
   else if obl≠()
           then /* exibe menu obl */
   else   /* senão exibe menu de itens opcionais */
```

Consistência da árvore sintática

Para a ativação do mecanismo da gramática de atributos é necessário verificar se a árvore está consistente. Para isto usa-se a função:

```
type TpASA --> Boolean
IsOkASA(a) ≡ (∀i ∈ Leafs(a))(symb(i,a) ∈ TerU(^))
```

Outra forma de verificar se a árvore está consistente é testar se não é gerado nenhum item obrigatório nos menus (BagObligates=()).

Inserção para grafos

Após a seleção de um item de menu a estrutura correspondente é inserida na árvore. No caso do item selecionado ser do tipo de criação usa-se a função **InsertProd** (apêndice A), que insere a estrutura de uma produção na árvore sintática.

No caso de ser um item do tipo compartilhado, utiliza-se a função **InsertShared.** Antes da inserção é necessário testar se a opção selecionada é válida, para isto utiliza-se a função **IsValidShared.** Uma ligação é válida se e

somente se a classe do first (f) selecionado for do tipo Shared e se o nome do nodo apontado (c) for o mesmo da produção, por exemplo ?Tg e !Tg.

```
type IsValidShared: Symb x TpCursor x TpASA --> TpBoolean
     IsValdShared(s,c,a) ≡ s=symb(c,t) ∧ s ∈ Shareds
```

A função InsertSHARED cria um nodo do tipo ligação em duas etapas: na primeira, insere a produção (InsertProd) e na segunda, cria a ligação entre a produção inserida e o nodo referenciado (SetPtr).

```
type insertSHARED :
     TpCursor x Symb x Symb x TpCursor x TpASA      --> TpASA
pre-InsertShared(ci,f,p,cf,a) ≡ isvalidShared(f,ci,a)
InsertSHARED( ci, f, p, cf, a) ≡
     Let a1 = InsertProd(ci,a,Select(f,p))
     in  SetPtr( In(ci),a1,Id(cf,a))

/* ligacao de ci (f=first, p=producao) para cf */
```

Na tela são listados todos os "first" (primeiros terminais) de cada terminal gerado no menu. O usuário seleciona um destes "first", e a função select busca a alternativa de uma produção a partir do "first" selecionado.

A função SetPtr cria uma ligação entre o nodo donatário e o nodo compartilhado (figura 8.11, #9).

8.3 Remoção de elementos estruturais

Na remoção, também são considerados os tipos de estruturas, conforme classificação no capítulo 2.

Para manter a consistência da estrutura sintática estendida pelo conceito de nodos compartilhados, as

operações de remoção seguem as regras abaixo:

i) uma estrutura **tipo opcional** pode ser excluída livremente.

ii) numa estrutura **tipo lista** pode-se excluir livremente (em qualquer ponto da lista) qualquer número de elementos.

iii) no **tipo agregação**, se um de seus elementos é excluído então toda a estrutura é excluída.

iv) quando for removido um nodo **compartilhado**, remove-se juntamente todos os seus donatários.

No próximo item, 8.3.1, são apresentados alguns exemplos de remoção, e no seguinte, 8.3.2, é apresentado o algoritmo que executa a remoção.

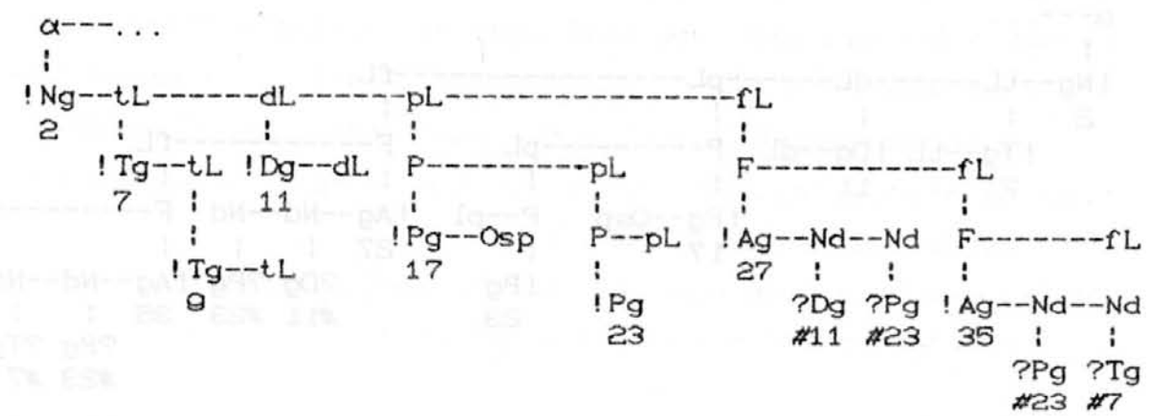
8.3.1 Exemplos de remoções

A figura 8.14 ilustra a relação entre as opções do menu e o estado da árvore sintática. Vale observar:

a) As linhas 1 e 2 em ligações, figura 8.14, representam um fluxo: ligação (arco) de um Depósito para um Processo.

b) Pode-se ver que os terminais do tipo donatário (?) não são mostrados na lista de possibilidades para remoção.

A partir da figura 8.14, com a remoção da entidade !Tg (com Id=9) da terceira linha das opções do menu de remoção, obtém-se o novo estado ilustrado na figura 8.15.



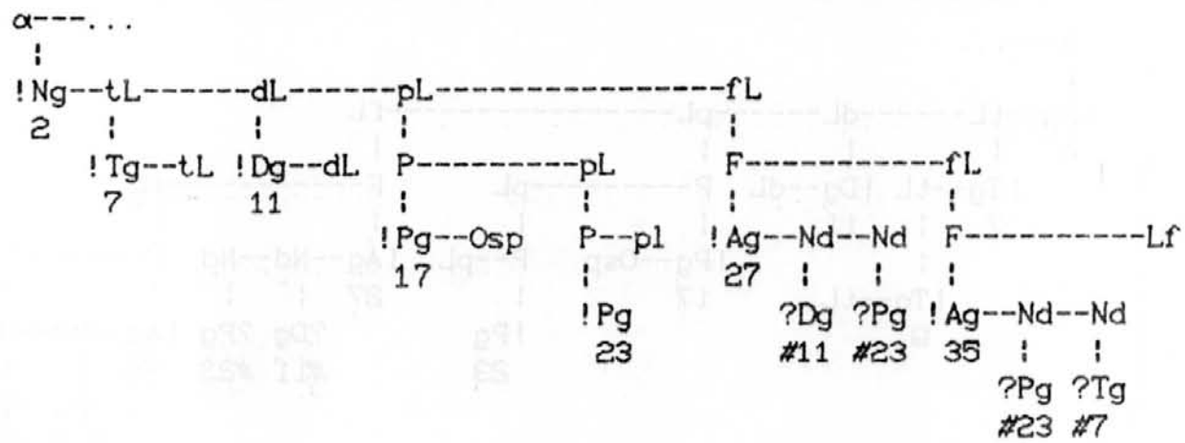
Ligações

	donatário	produção	terminal-referenciado	
1	INNNNIINI	Nd	?Dg	#11
2	INNNNIINNI	Nd	?Pg	#23
3	INNNNINIINI	Nd	?Tg	#7
4	INNNNIINNI	Nd	?Pg	#23

Exclusão

	cursor	produção	terminal-Id	
1	I	α(raiz)	!Ng	2
2	INI	tL	!Tg	7
3	ININI	tL	!Tg	9
4	INNI	dL	!Dg	11
5	INNNII	P	!Pg	17
6	INNNINII	P	!Pg	23
7	INNNNII	F	!Ag	27
8	INNNNINII	F	!Ag	35

figura 8.14: menu com as possibilidades de remoção



Ligações

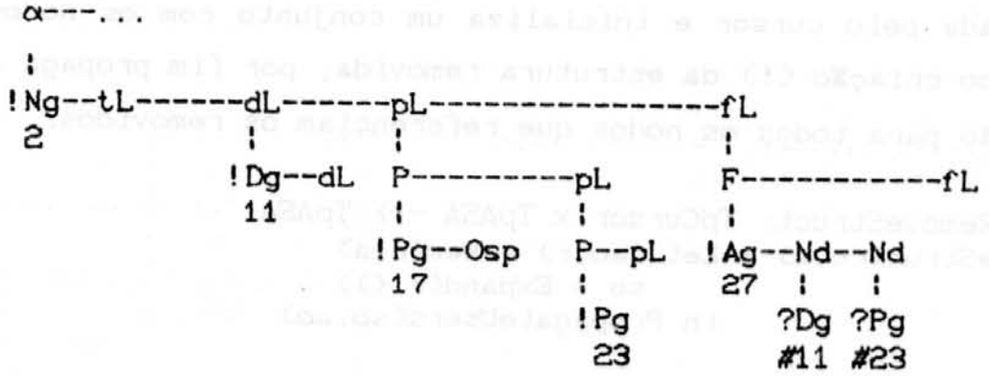
	donatário	produção	terminal-referenciado
1	INNNNIINI	Nd	?Dg #11
2	INNNNIINNI	Nd	?Pg #23
3	INNNNINIINI	Nd	?Tg #7
4	INNNNIINNI	Nd	?Pg #23

Exclusão

	cursor	produção	terminal-Id
1	I	α(raiz)	!Ng 2
2	INI	eL	!Tg 7
3	INNI	dL	!Dg 11
4	INNNII	P	!Pg 17
5	INNNINII	P	!Pg 23
6	INNNNII	F	!Ag 27
7	INNNNINII	F	!Ag 35

figura 8.15: remoção de um elemento tipo lista

Remove-se, figura 8.15, novamente uma entidade (segunda linha das opções de remoção). Esta entidade (id=7) está sendo referenciada por um fluxo; isto implica na remoção da estrutura que representa o fluxo (Id=35). Este exemplo ilustrou a remoção de um elemento compartilhado. (figuras 8.15-8.16).



Ligações

	donatário	produção	terminal-referenciado	
1	INNNNIINI	Nd	?Dg	#11
2	INNNNIINNI	Nd	?Pg	#23

Exclusão

	cursor	produção	terminal-Id	
1	I	α(raiz)	!Ng	2
2	INNI	dL	!Dg	11
3	INNNII	P	!Pg	17
4	INNNINII	P	!Pg	23
5	INNNNII	F	!Ag	27

figura 8.16: remoção de um elemento compartilhado

Neste mesmo exemplo ilustrou-se a remoção de um elemento do tipo agregação. A remoção de um subcomponente obrigatório da produção F --> !Ag Nd Nd, implica na remoção de toda a estrutura de F.

8.3.2 Os algoritmos de remoção

A função **RemoveStruct** executa a remoção da estrutura indicada pelo parâmetro cursor. Esta operação garante a consistência do conceito de nodos compartilhados, i.e. na remoção de uma estrutura deve-se remover juntamente os terminais que referenciam a estrutura removida.

Esta função (**RemoveStruct**) remove a estrutura indicada pelo cursor e inicializa um conjunto com os nodos do tipo criação (!) da estrutura removida; por fim propaga a remoção para todos os nodos que referenciam os removidos.

```
type RemoveStruct: TpCursor x TpASA --> TpASA
RemoveStruct(c,a) ≡ Let (ao,r) = Rem(c,a)
                        so = Expand(r,())
                    in PropagateUsers(so,ao)
```

```
type Rem: TpCursor x TpASA --> TpASA x TpASA
Rem(c,a) ≡
if pre-NearOpt(c,a) then
    if isList(c,a)
       then let b  = CutIn   ( next(in(c)), a)
                ao = RemoveIn( next(in(c)), a)
                r  = CutIn    (c,ao)
                a1 = RemoveIn(c,ao)
            in (InsertIn(c,a1,b), r)
      else ( RemoveIn(c,a), CutIn(c,a))
 else (InitASA(), a)
```

Toda remoção (**Rem**) acontece sobre um não-terminal que tem uma alternativa vazia (tipo opcional ou tipo lista). Este nodo ancestor é selecionado pela função **NearOpt**. Se o nodo for do tipo lista é necessário retirar apenas um componente da lista, o que se faz em três etapas: primeiro remove-se toda a parte da lista excluindo o nodo a ser removido, em seguida remove-se o nodo a ser excluído e por fim inclui-se a parte da lista removida na árvore. As operações **CutIn** e **RemoveIn** são sempre chamadas aos pares porque a primeira apenas devolve a estrutura cortada, enquanto que a segunda devolve a árvore sem a estrutura cortada.

Se o nodo é do tipo opcional (else do isList) simplesmente devolve-se a estrutura cortada e a árvore sem a estrutura cortada.

A função **NearOpt** escolhe o nodo ancestor mais próximo, do tipo opcional ou lista, onde será executada a remoção.

```
pre-NearOpt(c,a) ≡ (∃i∈TpCursor) (i≪c ∨ i=c)
                        ∧ symb(i,a)∈Optionals
pos-NearOpt(c,a,r) ≡ (r≪c ∨ r=c).symb(r,a)∈Optionals
                        ∧ not(∃i∈TpCursor, c≪i≪r)
```

A função **Expand** recebe como entrada uma estrutura removida e retorna um conjunto contendo todos os nodos do tipo criação (! - **isCreate**) que foram removidos.

```
type Expand: TpAsa x Id-set --> Id-set
     pos-Expand(a,s,r) ≡
     let so = { ID(c,a) ! c∈TpCursor, isCreate(c,a) }
     in  r = s∪so
```

A função **PropagateUsers** propaga o efeito da remoção para os nodos que referenciam um nodo removido. Pega-se um elemento do conjunto dos nodos removidos e remove-se todos os nodos que o referenciam (**RemUsers**). Os nodos removidos do tipo criação são incluídos no conjunto. Este processo termina quando o conjunto dos removidos está vazio.

```
type PropagateUsers: TpASA x Id-set --> TpASA
pos-PropagateUsers(a,s,r)≡
                        (∀c∈TpCursor).not(Ptr(c,r)∈s)
PropagateUsers(a,s)≡
   if s = {} then a
     else let so ∈ s,
               (ao, s1) = RemUsers(so,a,s)
               sr   = s1 - so
          in  PropagateUsers( ao, sr)
```

RemUsers remove todos os nodos que referenciam um particular nodo removido.

```
type RemUsers: Id x Idset x TpASA --> Id-set x TpASA
pos-RemUsers(i,s,a)≡ (∀c∈TpCursor).Ptr(c,a)≠i)
    RemUsers(i,s,a)≡
    if not pos-RemUsers(i,s,a) then
        let c∈TpCursor ¦ Ptr(c,a)=i
            (ao,R) = Rem(c,a)
            so = Expand(r, s)
        in  RemoveUsers(i,so,ao)
    else (so, ao)
```

O que garante que a estrutura sintática se mantém consistente (o invariante) é a remoção sobre os nodos do tipo lista ou opcional. Para a estrutura de grafo a consistência é verificada pela propagação do efeito da remoção sobre os nodos que foram afetados.

9 OPERAÇÕES NÃO ESTRUTURAIS (EDIÇÃO DE LÉXICOS)

Cada terminal (elemento léxico), representado por uma classe, é definido por um conjunto de atributos. As operações **não estruturais** permitem a modificação dos valores destes atributos - são as operações de edição de classes.

Como foi visto no capítulo 2, as classes são definidas por dois tipos de atributos: (1) atributos que definem o tamanho e a posição dos objetos geométricos e (2) os atributos do tipo decoração, por exemplo, os atributos textuais. Em função dos tipos de atributos os editores diagramáticos apresentam dois modos de operação: **modo texto e modo gráfico** (ver [MEL 89]). No modo texto são editados os atributos textuais e no modo gráfico as classes são editadas a nível gráfico.

Antes de se executar uma operação de edição é necessário identificar o elemento terminal na tela. Esta operação de identificação é chamada de **apontamento.**

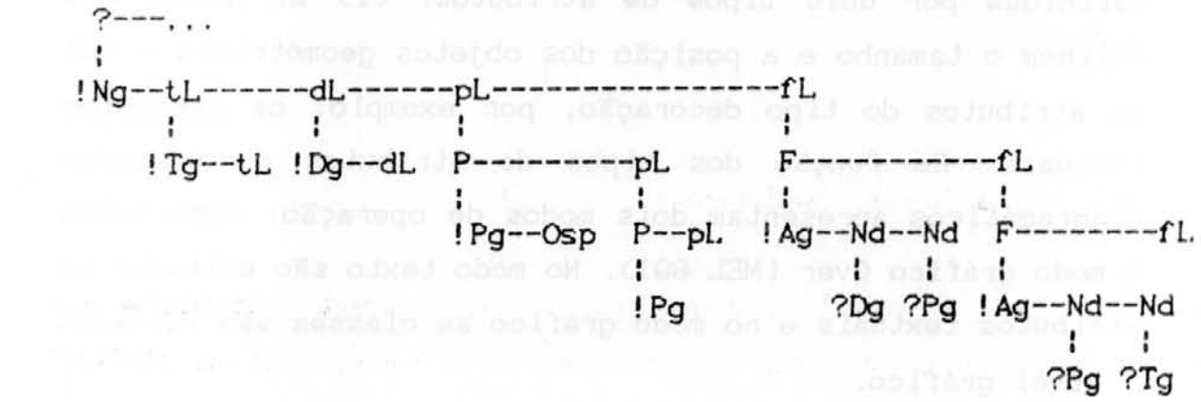
A seguir descreve-se o processo de formatação e a partir das informações geradas pela formatação mostra-se como é executada a operação de apontamento.

9.1 Formatação

Para linguagens diagramáticas cada símbolo terminal (folha da árvore sintática) está associada a um elemento léxico. Os elementos léxicos do tipo criação (!) possuem uma representação concreta como objeto gráfico. Ao contrário, os donatários (?) não possuem uma representação

concreta na tela, são apenas ligações que guardam algum tipo de informação estrutural, por exemplo o conceito de arco para o caso de grafos.

O formatador exibe na tela todos os objetos gráficos associados aos elementos terminais do tipo (!). Os objetos exibidos na tela são inseridos na tabela dos **terminais exibidos**, como mostra a figura 9.1.



Terminais exibidos

	cursor	produção	terminal	classe
1	I	?	!Ng	#1
2	INI	tL	!Tg	#2
3	INNI	dL	!Dg	#3
4	INNNII	P	!Pg	#4
5	INNNINII	P	!Pg	#5
6	INNNNII	F	!Ag	#6
7	INNNNINII	F	!Ag	#7

Tabela de classes

#Id	Name	Envelope	Atributos
#1	Terminator	(10,10.0)-(15,20.0)	name="entidade"...
#2	Deposit	(30,10.0)-(35,10.0)	name="deposito"...
#3	Process	(30,40.0)-(35,40.0)	name="processo"...
#4	DataFlow	(15,12.5)-(45,32.5)	name="ligacao"...
...			

figura 9.1: classes exibidas na tela

A figura 9.1 mostra numa forma simplificada a tabela de classes. As informações do envelope são utilizadas para se executar a operação de seleção (apontamento) de um

objeto gráfico associado a uma classe.

A identificação da classe para a edição de atributos é feita pelo apontamento. O apontamento é usado também para identificar o nodo da estrutura sintática para as operações estruturais. Por exemplo, numa remoção aponta-se para a classe a ser removida; a remoção da classe associada a um terminal implica na remoção dos símbolos não-terminais que foram afetados (como foi exemplificado no capítulo 8).

9.2 Operação de apontamento

A nível operacional o apontamento consiste em selecionar um dos objetos mostrados na tela, com um "CLICK" do "mouse".

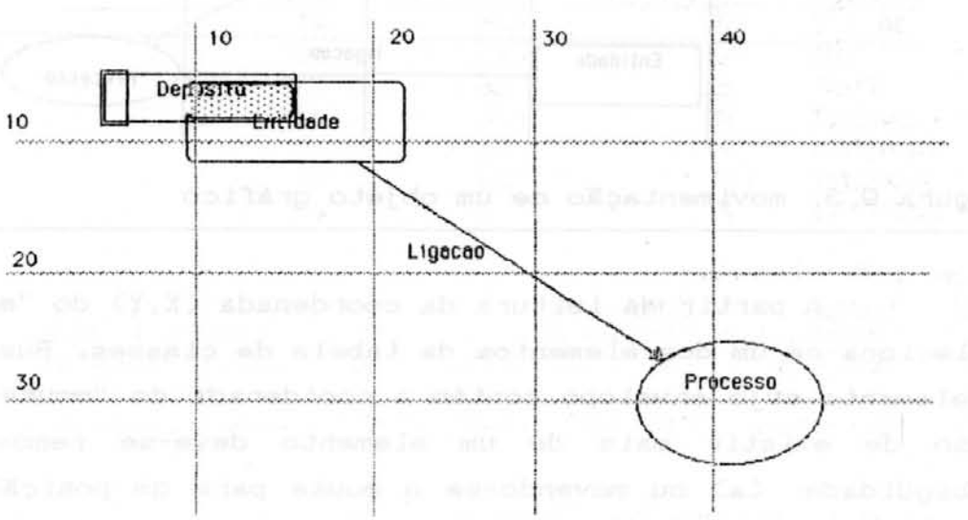


figura 9.2: ambiguidade na seleção de um elemento léxico

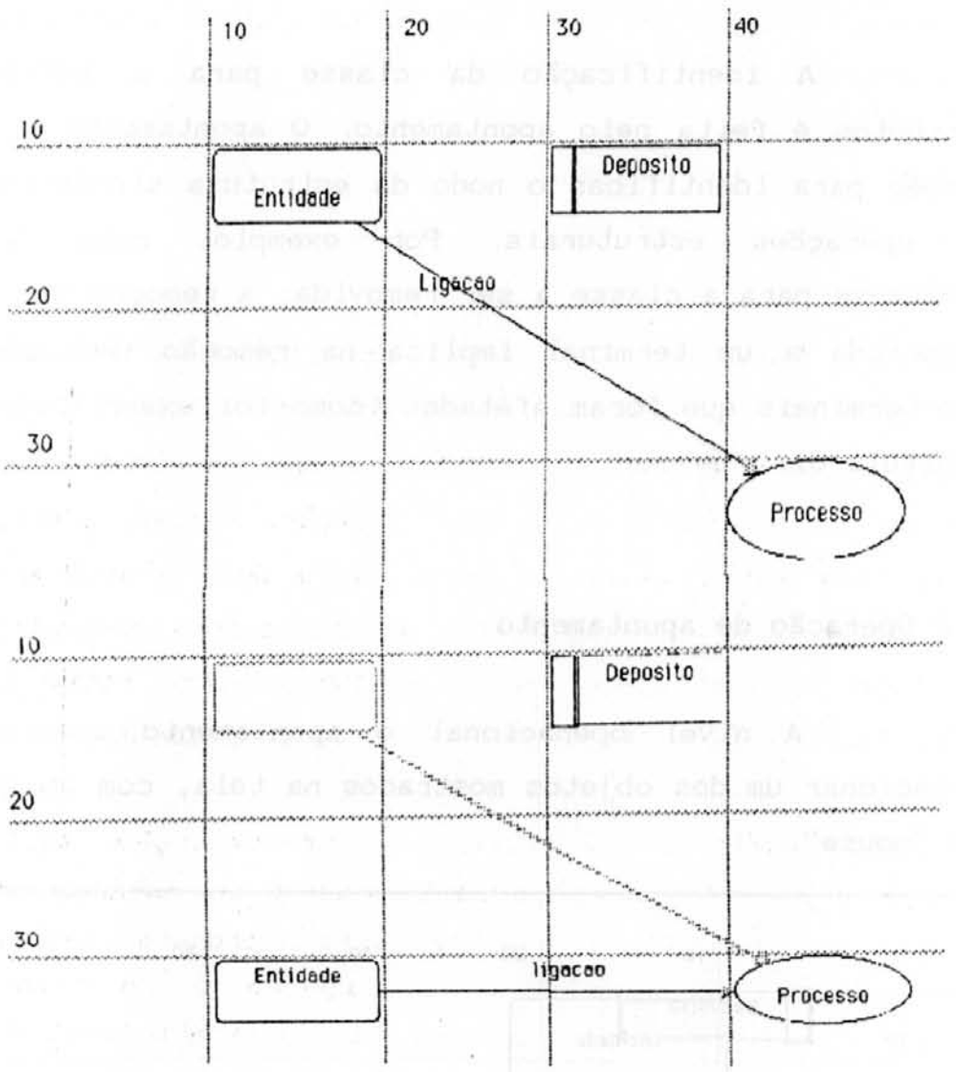


figura 9.3: movimentação de um objeto gráfico

A partir da leitura da coordenada (X,Y) do "mouse" seleciona-se um dos elementos da **tabela de classes.** Busca-se o elemento cujo envelope contém a coordenada do "mouse". No caso de existir mais de um elemento deve-se remover a ambigüidade: (a) ou movendo-se o mouse para um posição não ambígüa, (b) ou pela seleção de um dos objetos que se sobrepõe: o sistema exibe um a um os objetos sobrepostos e solicita uma confirmação.

A figura 9.2 ilustra dois objetos sobrepostos, se o "mouse" está posicionado na área pontilhada tem-se um caso de ambigüidade.

9.3 Movimentação de objetos gráficos

A movimentação apenas modifica os atributos Point, que guardam os valores da posição do objeto dentro da página. A movimentação é feita pela seleção (apontamento) e arrasto do objeto para uma nova posição. A nível operacional a movimentação implica na reavaliação dos valores afetados da classe e subclasses. Após a movimentação alguns atributos do tipo externo podem ser afetados também, neste caso o avaliador da GA atua fazendo a devida propagação.

A figura 9.3 ilustra uma movimentação. As figuras 9.4 e 9.5 representam respectivamente os estados antes e após a movimentação. O envelope para o processo também é um retângulo. O envelope para a ligação é do tipo reta, representado por três pegadores, um em cada extremo e um no meio da "seta".

Tabela de classes

#Id	Name	Envelope	Atributos
#1	Terminator	(10,10.0)-(15,20.0)	name="entidade"...
#2	Deposit	(30,10.0)-(35,10.0)	name="deposito"...
#3	Process	(30,40.0)-(35,40.0)	name="processo"...
#4	DataFlow	(15,12.5)-(45,32.5)	name="ligacao"...

...

figura 9.4: lista de objetos gráficos antes da movimentação

Tabela de classes

#Id	Name	Envelope	Atributos
#1	Terminator	(30,10.0)-(35,20.0)	name="entidade"...
#2	Deposit	(30,10.0)-(35,10.0)	name="deposito"...
#3	Process	(30,40.0)-(35,40.0)	name="processo"...
#4	DataFlow	(15,32.5)-(45,32.5)	name="ligacao"...

...

figura 9.5: lista dos objetos gráficos após a movimentação

9.4 Edição de atributos

Para editar um atributo seleciona-se um objeto, e mostra-se os atributos editáveis do objeto (os do tipo texto). Esta operação ativa o modo de **edição textual.** Estes valores são modificados com a utilização de um editor de texto.

10 PARTICIONAMENTO DO DOCUMENTO EM PÁGINAS

Uma página é uma unidade de trabalho de um documento. As operações são executadas na página corrente que corresponde à porção conceitual do documento que está sendo editado, por um determinado período de tempo.

Durante a edição, a representação concreta da página é mostrada em uma janela na tela. Os elementos terminais da descrição sintática (fronteira da árvore) são graficamente exibidos na tela, pelo processo de formatação. Após cada operacão de edição o processo de formatação é executado, atualizando a representação concreta da página.

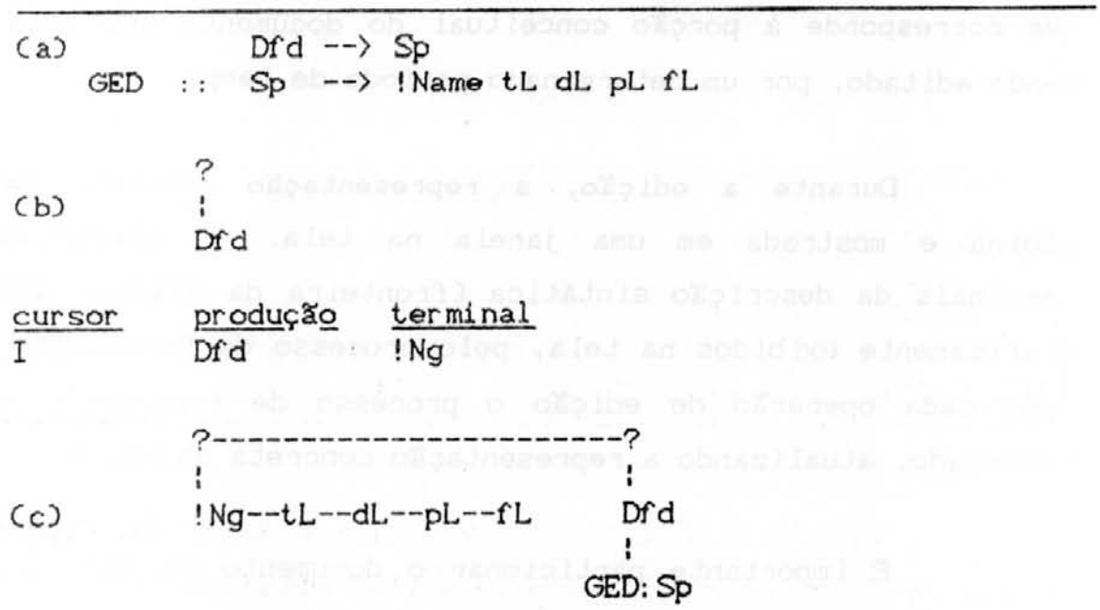
É importante particionar o documento em páginas, pois pode-se trazer para a memória somente as páginas que estão sendo alteradas.

10.1 Subárvores associdas a páginas

Como foi comentado no capítulo 5, item 5.6, cada produção rotulada por "ED", figura 10.1(a), está associada a uma página. Todas as páginas de um documento são organizadas na floresta (FOREST) de páginas do sistema. Assim, cada ocorrência de um nodo "Sp" implica na criação de subárvore, associada a nova página. Esta nova página deve ser incluída na FOREST.

A inclusão de uma página a partir da figura 10.1(b) produz a estrutura da figura 10.1(c). Esta figura ilustra a FOREST com duas páginas, na segunda o nodo GED: Sp indica que é um nodo tipo página. A remoção da subárvore do nodo Sp e a inclusão desta na FOREST é feita automaticamente

pelo sistema. Neste momento assume-se como página corrente de edição a página que está mais à esquerda (? --> !Ng) na figura 10.1(c).

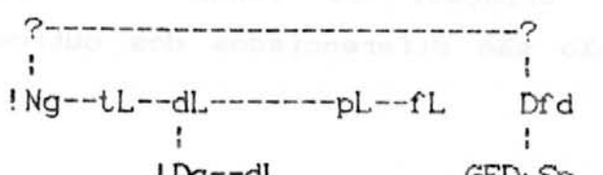


	cursor	produção	terminal
1	IN	tL	!Tg
2	INNIN	dL	!Dg
3	INNN	pL	!Pg
4	INNNN	fL	!Ag

figura 10.1: inclusão de um nodo associado a uma página

10.2 A inclusão de uma página

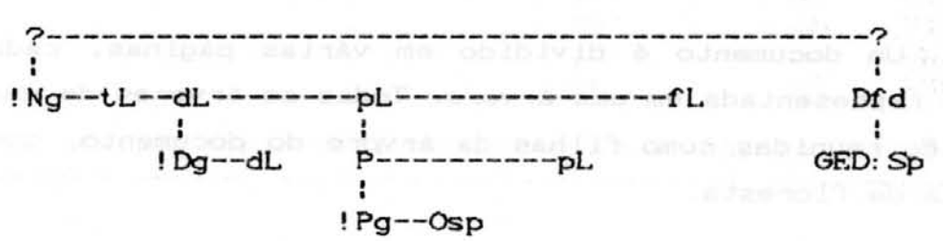
Na figura 10.1 é mostrado o menu associado ao estado da árvore. Selecionando-se a linha 2 para inclusão tem-se a árvore da figura abaixo:



	cursor	produção	terminal
1	IN	tL	!Tg
2	INNIN	dL	!Dg
3	INNN	pL	!Pg
4	INNNN	fL	!Ag

figura 10.2: menu de elementos opcionais

Esta nova árvore, mantém o mesmo menu, da figura acima. Selecionando-se a terceira linha, obtém-se:



	cursor	produção	terminal
1	IN	tL	!Tg
2	INNIN	dL	!Dg
3	INNNIIN	Osp	!Ng
4	INNNIN	pL	!Pg
5	INNNN	fL	!Ag

figura 10.3: inclusão de um processo

A explosão de um processo é opcional, i.e., a produção Osp é do tipo opcional. Para a explosão do processo seleciona-se a linha 3 do menu, a qual gera uma nova página.

Para efeito de criação, os ítens de menu associados a uma página não são diferenciados dos outros ítens de menu.

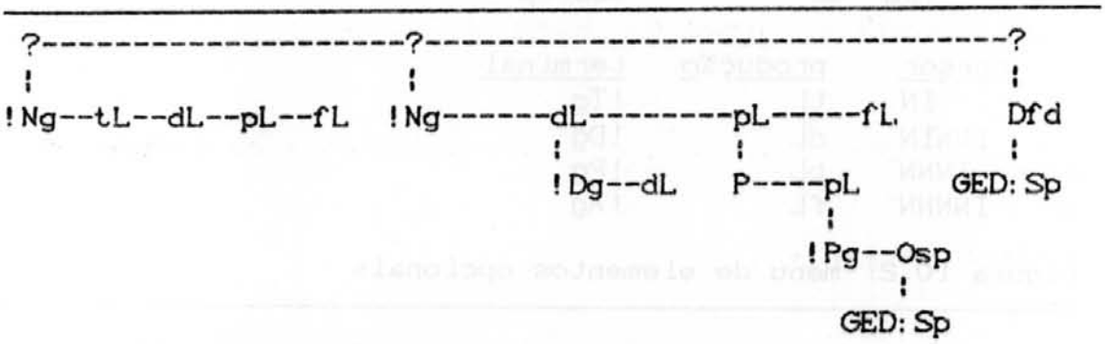


figura 10.4: explosão de um processo

10.3 Operações para a manipulação da floresta de árvores

Um documento é dividido em várias páginas, cada página é representada em uma árvore. Todas as árvores de uma página são reunidas como filhas da árvore do documento, que é chamada de floresta.

Esta solução de representar as páginas agrupadas numa árvore de um documento (como se fosse um arquivo) tem como vantagem a possibilidade de se manipular a representação do documento como um todo utilizando-se as (mesmas) primitivas de manipulação de nodos de uma árvore.

Para o conceito de página a gramática foi estendida permitindo um rótulo que indica que é do tipo página. Na especificação dos algoritmos é feita referência ao conjunto da páginas **Pages**. Para o fragmento de gramática abaixo

```
        Dfd --> Sp
GED :: Sp  --> !Name tl
```

o conjunto das páginas é: Pages = (Sp).

O domínio de dados para a floresta de árvores é:

```
TpForest = TpASA
onde
     Node :: SYMB: Symbs U Nullsymb
             PAGE: Integer U Nullpage
             ... /* definições do tipo TpASA */
```

Segue uma especificação das operações para a manipulação das árvores (páginas) na floresta.

A função **InitForest** cria um arquivo vazio; é uma árvore sintática vazia.

```
type InitForest:  --> TpForest
     InitForest()  ≡ initASA( NullSymb )
```

A função **CreateASA** inclui uma página em uma floresta, retornando a nova floresta.

```
type CreateASA : TpForest x TpASA  --> TpForest
     CreateASA(f,a)≡
      Let fo = mk-Node(_, emptyASA, f)
      in  insertIn(emptyCursor, fo, a)
```

A função **SetPage** associa uma página a um nodo folha do tipo página. Esta função é utilizada no momento de criação de uma nova página. A função **GetPage** retorna a página associada ao nodo. Esta função é utilizada para se fazer acesso a uma página existente.

```
type SetPage   : TpCursor x TpASA x TpPage --> TpASA
     SetPage(c,a,p) = μ(asA(c,a),[PAGE -->p])

type GetPage   : TpCursor x TpASA          --> TpASA
     GetPage(c,a)   = PAGE(asA(c,a))  /*PAGE é seletor de campo */
```

A função **IsPage** retorna **True** se o nodo indicado é do tipo página. A função **ExistPage** retorna true se um nodo do tipo página já possui uma página associada.

```
type IsPage    : TpCursor x TpASA  --> TpBoolean
     IsPage(c,t) = Symb(c,t) ∈ Pages

type ExistPage : TpCursor x TpASA  --> TpBoolean
     ExistPage(C,T) = (getpage(c,t)≠Nullpage)
```

A função **InsertPage** é chamada quando, na criação de nodos numa árvore, identifica-se que o nodo a ser criado é do tipo página. Neste caso, remove-se a subárvore associada à página e com o uso da função **CreateASA** insere-se a página na floresta. Após isto associa-se a nova página ao nodo folha (**SetPage**).

```
type InsertPage: TpCursor x TpASA x Ptrprod x TpForest
                          --> TpASA x TpForest

InsertPage(c,t,p) ≡
    let to = InsertProd(c,t,p)
    in  let t1 = CutIn     (c,to)
            t2 = removeIn  (c,to)
        in let fo =  CreateASA(f, t1 )
           in (SetPage(c,t2,Id(emptyCursor, fo)), fo)
```

As funções **ChangePage** e **ChangePageNo** são utilizadas para mudar a página corrente de edição. A primeira é utilizada quando o argumento de busca é o **Id** da página, enquanto que a segunda é utilizada quando o argumento de busca é um número inteiro. O valor deste inteiro deve estar situado no intervalo entre 1 e o número máximo de páginas, que é obtido pela função **NoMaxPages**.

```
type ChangePage    : TpASA x TpCursor x TpForest --> TpCursor
pos-ChangePage(a,c,f,co) ≡ Page(c,a)=Id(f,co)
ChangePage(f,c) ≡ ChangeNext(emptyCursor, f ,GetPage(c,f))

type ChangeNext: TpASA x TpCursor x TpForest --> TpCursor
ChangeNext(c,f,a) ≡  if a≠Id(c,f) and ExistNext(c,f)
                     then ChangeNext(next(c),f,a)
                     else c
```

A operação **ChangePage** é válida se a página foi criada pela operação **InsertPage**, que associa uma árvore a floresta. Para a **ChangePageNo** é definida uma pré-condição para garantir a validade da operação.

```
type ChangePageNo : TpForest x Integer    --> TpCursor
     pre-ChangePageNo(f,i) ≡ 1<i<NoMaxpages(f)
     pos-ChangePageNo(f,i,co) ≡ lengthNext(co,f)=i
     ChangePageNo(f,1) ≡ ChangeNoNext(emptyCursor,f,1)

type ChangeNoNext: TpCursor x TpForest x TpASA --> TpCursor
ChangeNoNext(c,1) ≡ if 1<LengthNext(c,f) and ExistNext(c,f)
                    then ChangeNoNext(next(c),f,1)
                    else c

type NoMaxPages    : TpForest --> TpInteger
NoMaxPages(f) = lengthNext(emptyCursor,f)
```

11 TRABALHOS RELACIONADOS

Este capítulo comenta alguns tópicos de trabalhos relacionados com o tema desta dissertação.

11.1 A linguagem do tipo "constraint-based"

No capítulo 3 exemplificou-se a especificação dos elementos léxicos numa linguagem do tipo "constraint-based" [BAR 87]. Nesta linguagem, baseada em regras de formação, as variáveis que definem os objetos gráficos são explicitadas. Assim sendo, é fácil relacionar os atributos dos símbolos terminais da GA com as variáveis. O modo de especificação interativa (também apresentado no capítulo 3) dificulta o mapeamento dos atributos dos símbolos terminais para as variáveis que definem os objetos gráficos.

Na especificação dos léxicos interativamente não é possível definir alguns tipos de restrições. Por exemplo na implementação do EDG [MEL 89] são possíveis duas formas de restrição: ou os componentes do objeto gráfico são fixos sobre o envelope (não podem mudar de posição dentro do envelope do objeto gráfico) ou são totalmente livres (podem mudar de posição). Com a especificação das restrições por regras definidas em função dos valores das variáveis que definem as figuras geométricas podem ser definidos outros tipos de restrições.

11.2 Gramática de grafos

A sintaxe de uma linguagem de programação é descrita por uma GLC. O capítulo 3 apresenta uma extensão da GLC para permitir a definição de linguagens do tipo diagramático baseadas em grafos (nodos compartilhados).

Na literatura relacionada com a construção de editores orientados por estrutura existe um formalismo chamado gramática de grafos [ENG 88] [NAG 88] [GOT 82] [GOT 88]. Este formalismo é usado no ambiente IPSEN para a construção de ferramentas textuais e/ou diagramáticas [NAG 88].

Já existem trabalhos no sentido de estender o formalismo gramática de grafos com equações semânticas que podem ser avaliadas incrementalmente de forma similar a uma GA [KAP 87].

Nas fases iniciais dos desenvolvimento deste trabalho pensou-se na hipótese de utilizar o formalismo gramática de grafo por parecer ser mais natural para especificar as notações diagramáticas baseadas em grafo. Porém, após uma análise da bibliografia, optou-se por extender a LDE para expressar o conceito de grafo, por três razões:

i) primeiro, o formalismo gramática de grafos é mais geral que a GLC estendida pelos nodos compartilhados. No entanto este grau de generalidade é de uma maior complexidade, tanto a nível de especificação das notações como a nível de implementação do formalismo.

ii) segundo, o formalismo GA está mais desenvolvido, maduro, i.e. existe uma boa bibliografia, algoritmos eficientes,

etc. No capítulo 2 são referenciados vários trabalhos relacionados com o tema GA.
iii) terceiro, no projeto ADS existe muito trabalho que pode ser reusado (diretamente ou com adaptações). Por exemplo algoritmos de construção de árvores sintáticas, avaliador de GA, etc.

11.3 A metalinguagem devida a Hekmatpour

Hekmatpour [HEK 87] apresenta uma metalinguagem para especificação de linguagens diagramáticas para a geração de editores orientados por estrutura. O formalismo proposto por Hekmatpour apresenta três níveis: léxico, sintático e semântico.

O nível léxico é especificado de forma interativa, como em [MEL 89].

No nível sintático os símbolos léxicos são combinados através de relações. As contruções introduzidas, no capítulo 5, para especificação da sintaxe de linguagens do tipo grafo são similares (possuem o mesmo poder de expressão) que as construções do tipo relação do trabalho citado.

O nível de semântica é definido por um conjunto de regras. As regras são especificadas numa notação funcional que combina: operadores lógicos, operadores de conjuntos, operadores de seleção das relações definidas no nível de sintaxe e contruções de linguagens funcionais.

Esta notação é menos geral que a metalinguagem LDE. Por exemplo, o nível de "semântica" da linguagem não é utilizado para se definir atributos que poderiam ser utilizados no nível léxico; assim, todos os aspectos relacionados com formatação devem ser definidos no nível léxico.

Esta notação é simples se comparada à gramática de grafo ou a LDE, porém é menos geral.

No trabalho citado [HEK 87] é apresentada uma especificação de um diagrama de fluxo de dados (DFD). No capítulo 5 também descreve-se um DFD. Estas especificações podem ser comparadas.

11.4 O synthesizer generator

O Synthesizer Generator [REP 87] foi o sistema que introduziu a avaliação incremental de GAs para editores orientados por estrutura.

O ganho em estender a LDE, baseada em GA, com nodos compartilhados (para as estruturas do tipo grafo) é a utilização de algoritmos para a avaliação de GAs e de facilidades associadas (por exemplo na implementação de tabelas de símbolos). Existe um trabalho de mais de 10 anos nesta área [REP 87].

Os trabalhos relacionados com GA foram comentados no capítulo 2.

12 CONCLUSÃO

Este trabalho descreve uma proposta para a implementação de um editor orientado por estrutura para notações diagramáticas (GED). O GED aceita como entrada descrições das notações na metalinguagem LDE, baseada em gramática de atributos.

Na parte inicial do trabalho exemplificou-se a especificação de notações diagramáticas em LDE enfatizando os níveis léxico, sintático e semântico. Nos capítulos finais discutiu-se os diversos aspectos relacionados com a implementação do GED.

A metalinguagem LDE

Na LDE (a nível de GLC) foi introduzido o conceito de nodos compartilhados que permitem especificar notações diagramáticas com estrutura de grafo. Mostrou-se como implementar este conceito num editor orientado por estruturas: gerar menus para as operações de edição.

A nível de semântica buscou-se, na medida do possível, manter a notação de GA tradicional da LDE, usada na especificação de linguagens textuais para o gerador de editores textuais (GET). Isto possibilitou a definição de uma arquitetura comum para o GET e GED (ambos em desenvolvimento no projeto ADS-UFRGS), que permite integração entre ferramentas textuais e diagramátricas.

Para a descrição dos elementos léxicos foi introduzida uma notação baseada em linguagens do tipo "constraint-based". Os elementos gráficos descritos nesta

notação são mapeados para os símbolos terminais da GA.

A principal contribuição

Este trabalho constitui-se numa das primeiras tentativas de utilização do formalismo GA para descrever e implementar ferramentas diagramáticas. Optou-se por GA devido ao largo uso deste formalismo na descrição de semântica para linguagens textuais (para compiladores, editores orientados por estrutura, etc.) e pela existência de algoritmos de avaliação incremental.

Um dos maiores resultados deste trabalho é a possibilidade de integração de ferramentas (textuais e/ou diagramáticas) através das informações estruturais (AS) e das tabelas de símbolos em um BD. Esta integração de ferramentas permite a construção de sofisticados ambientes de desenvolvimento de software (ou outros ambientes de edição).

Trabalhos futuros

Diversas atividades podem ser desenvolvidas a partir do protótipo do GED:

a) Construção de editores diagramáticos específicos (por ex. DFD, diagrama E-R, Diagrama estruturado) fazendo uso do GED.

b) Construção de editores textuais específicos, para Dicionário de dados (DD), português estruturado, já visando a integração pelas informações em TSs.

c) A implementação do BD integrado aos editores ED/ET, como foi apresentado no capítulo 7.

No projeto ADS, está em andamento a construção do módulo que implementa as facilidades de tabelas de símbolos. Este módulo está sendo construído de forma independente de um sistema gerenciador de BD. Trabalhos futuros deverão buscar a integração das tabelas de símbolos em um banco de dados.

d) Reescrita de módulos do ED/ET buscando eficiência. Os algoritmos existentes atualmente no ET para a avaliação de GA não são eficientes; novos trabalhos deverão substituir estes algoritmos por algoritmos mais eficientes.

e) A implementação do sistema de edição dos componentes léxicos baseado na metalinguagem apresentada no capítulo 3.

f) Experimentações com a integração de diferentes ferramentas como um ADS: problemas de interface; problemas relacionados com o compartilhamento de informações à nível estrutural (As) e à nível de TSs; problemas das notações não desenvolvidas para serem automatizadas; etc.

g) Estudo de metodologias que fazem uso das ferramentas geradas.

APÊNDICE A

Este capítulo é dividido em duas partes. Na primeira são introduzidas as noções básicas do método VDM, tendo como base os livros [JON 80], [JON 86] e na segunda são apresentados os tipos de dados que implementam a estrutura da árvore sintática.

A.1 Uma introdução ao Método VDM

A linguagem de definição Meta-IV, usada no método VDM, fornece uma visão orientada a modelos do sistema de software a ser desenvolvido. São definidos explicitamente os objetos matemáticos e operações do modelo do sistema de software. Os modelos básicos da Meta-IV são o conjunto, objeto composto, lista, mapeamento e o tipo função.

Obrigações de prova

A especificação dos domínios de dados associados às funções de manipulação, bem como o refinamento de um tipo de dados, dão origem às **obrigações de prova.** Estas provas são fórmulas que têm que ser verdadeiras para que o passo na especificação do projeto estabeleça ou preserve a "correção". As provas requeridas em cada caso devem ser realizadas no nível apropriado de formalidade podendo então a especificação ser classificada como formal, rigorosa ou sistemática.

Uma especificação, ou projeto, e o desenvolvimento (etc.) são ditos formais se todos os documentos e suas obrigações de prova são formalmente expressados e formalmente provados. Uma especificação é dita rigorosa

(quase formal) se todos os documentos e suas obrigações de prova cruciais são formalmente expressas, mas nem todas as provas são formalmente verificadas. Uma especificação é dita sistemática (quase rigorosa) se os documentos são formalmente expressados, mas nem todas as obrigações de prova cruciais são formalmente expressadas [BJO 88].

Existem duas maneiras de se introduzir as obrigações de prova: formal ou informal. Para uma especificação ser formal é necessário desenvolver formalmente todas as obrigações de prova. Numa prova formal o argumento é construído através de manipulações simbólicas de acordo com um conjunto de regras de inferência. Uma prova informal contém apenas os passos mais importantes do argumento, evitando, assim, uma detalhada manipulação simbólica. Porém, tendo-se as funções e os domínios expressados formalmente, pode-se desenvolver formalmente as obrigações de prova, sempre que se achar conveniente; por exemplo, se a prova informal não é convincente.

A decomposição de uma especificação em VDM, visando a sua implementação, ocorre em dois níveis: modelagem de dados e decomposição de operações. Os passos na modelagem de dados transformam objetos abstratos em objetos (menos abstratos) representáveis na linguagem de implementação escolhida; estes passos dão origem às obrigações de prova baseadas nas funções de retorno "retrieve functions". A decomposição de operações é o processo no qual as especificações implícitas (pré, pós-codições, funções recursivas) são decompostas em operações de mais baixo nível, até em termos de operações de uma linguagem de programação (comandos iterativos). Estes

passos de decomposição dão origem a dois tipos de obrigações de prova: as provas do tipo "satifaz a especificação" que verificam se a definição explícita da função satisfaz a definição implícita; e as provas que verificam a validade da combinação das sub-funções e/ou operações; estas provas validam a composição dos subcomponentes através das construções da linguagem (por ex. comandos sequenciais, iterativos, etc.)

As provas de obrigação orientam o trabalho de desenvolvimento, interrelacionando todos os componentes do sistema. Se todas as provas são verificadas significa que o sistema é consistente com as pré/pós-condições do primeiro nível. Como resultado, aumenta a confiança do projetista sobre a consistência da especificação. Porém, não é possível provar formalmente que a especificação satisfaz os requisitos informais do usuário. O máximo que se pode fazer é postular e provar teoremas sobre a especificação, podendo-se assim descobrir a existência de propriedades indesejáveis no sistema [JON 86].

O objetivo do trabalho é obter uma especificação funcional (sem considerar requisitos não funcionais tais como espaço e tempo) dos principais algoritmos utilizados na implementação do editor. Na medida do possível segue-se a abordagem sugerida por [JON 80], [JON 86]. Uma breve introdução a domínio de dados, funções e operações do método VDM é apresentada no próximo item.

Nos algoritmos não são realizadas provas formais. Apesar disso, esta introdução ao VDM apresenta as noções que o especificador deve ter em mente para validar cada passo

executado na construção da especificação. Conforme a discussão acima, a forma em que os algoritmos são apresentados é "quase" sitemática, uma vez que alguns tipos de dados, como o de representação de gramáticas, não é expressado formalmente; apenas as operações são enunciadas.

Na especificação dos algoritmos não é apresentado nenhum refinamento, no sentido de aproximar a especificação a uma linguagem de implementação. Portanto, muitas das noções citadas acima, como as funções de retorno, não são comentadas nesta breve introdução ao VDM.

A.1.1 Domínios de dados

As funções ou operações são descritas para atuarem sobre um ou mais domínios de dados. Um domínio de dado é descrito a partir dos tipos de dados pré-definidos do VDM: conjunto, lista, mapeamento, objetos compostos. Uma descrição detalhada destes tipos de dados é encontrada em [JON 86]. Seguem alguns exemplos de definição de domínios de dados.

```
ex: Date  :: day:⟨1,...,366⟩          /* tipo composto */
           year:⟨1583,...,2599⟩
    S : Date-set     /* conjunto de elementos do tipo data */
    L : Date-list    /* lista de elementos do tipo data    */
```

As vezes é necessário definir uma restrição sobre os valores de um domínio de dados. No exemplo abaixo, o tipo **Date1** contém um subconjunto do domínio **Date** tal que a função **inv-Date**, chamada de invariante, é verdadeira.

```
Date  :: day:⟨1,...,366⟩ year:⟨1583,...,2599⟩
```

```
Date1 :: Date with inv-Date(mk-Date1(d,y))
                    ≡ anobiss(y) ∨ y ≤ 365
assim: (∀d ∈ Date1) → (d ∈ Date ∧ inv-Date(d))
```

O invariante do tipo de dados é um predicado que vale para todas as instâncias de estados que podem ser criadas a partir das operações sobre o tipo. Somente os estados que satisfazem o invariante são ditos válidos (no item A.1.5 é exemplificado o uso do invariante).

A.1.2 Definição de funções

Uma função (noção de função matemática) na Meta-IV pode ser definida de forma direta ou implícita. Uma definição direta enuncia como o valor deve ser computado, enquanto que a definição implícita indica o que deve ser computado (pré e pós-condições).

Por exemplo, uma definição implícita seria:

```
type max: ℕ x ℕ --> ℕ
     pre-max(i,j)  ≡true
     pos-max(i,j,r)≡(r=i ∨ r=j) ∧ i≤r ∧ j≤r
```

Uma definição direta:

```
type max: ℕ x ℕ --> ℕ
     max(i,j)       ≡ if i≤j then j else i
```

A noção de função para especificação de sistemas é apresentada através de um exemplo. Os domínios MARRIED e UNMARRIED representam respectivamente o conjunto de pessoas casadas e não casadas. Sobre estes tipos de dados são definidas as funções:

Register : inclui uma pessoa no conjunto das não casadas

Marry : inclui um casal no conjunto das casadas

isMarried : consulta se uma pessoa está casada

Init : inicializa os conjuntos

Na especificação destas funções segue-se a convenção de uso de letras maiúsculas para os domínios de dados considerados principais; letras minúsculas com pelo menos uma maiúscula para os outros domínios de dados e para nomes de funções. As palavras reservadas da meta linguagem são sublinhadas.

```
UNMARRIED : Person-set
MARRIED   : Person-set
Person    : /* qualquer representação adequada */

type Register : UNMARRIED x MARRIED x Person --> UNMARRIED
     pre-Register(u,m,p)   ≡ p ∉ u ∧ p ∉ m ∧ m∩u=()
     pos-Register(u,m,p,r) ≡ r = u U (p)   ^ m∩u=()

type Marry : UNMARRIED x MARRIED x Person x Person
                                  --> UNMARRIED x MARRIED
     pre-Marry(u,m,h,w)        ≡ h ∈ u ∧ w ∈ u ^ m∩u=()
     pos-Marry(u,m,h,w,ur,mr) ≡ (ur = u - (h, w)) ∧
                                  (mr = m U (h, w)) ^ m∩u=()

type isMarried :UNMARRIED x MARRIED x Person --> Boolean
     pre-isMarried(u,m,p)    ≡ true ^ m∩u=()
     pos-isMarried(u,m,p,b) ≡ (b = p ∈ m) ^ m∩u=()

type Init:  --> UNMARRIED x MARRIED
     pre-Init()     ≡ true
     pos-Init((),u,m) ≡ u=() ∧ m=() ^ m∩u=()
```

O predicado m∩u=() anexado a todas as pré e pós-condições (com excessão da função Init) garante que uma pessoa não pode ser ao mesmo tempo casada e descasada.

A.1.3 Obrigações de prova para funções

A especificação de uma função, bem como o relacionamento entre a definição implícita e direta dão origem a obrigações de prova.

A.1.3.1 Implementabilidade(pi)

É possível escrever uma definição que nunca pode ser satisfeita. Por exemplo, na função **sqrt** abaixo, sabe-se que não existe um número inteiro **y** tal que **y*y=2**. A obrigação de prova de **implementabilidade**, se satisfeita, previne casos como este.

```
type sqrt : ℤ  --> ℤ
     pre-sqrt(x) ≡ x ≥ 0
     pos-sqrt(x,y) ≡ y*y = x
```

A obrigação de prova de **implementabilidade** (pi) exige que para cada valor do domínio de uma função/operação exista algum resultado no contra-domínio.

```
Dada         f: D      --> R,
          pre-f: D      --> Boolean,
          pos-f: D x R  --> Boolean,

onde      pre-f(d)    ≡ ...
          pos-f(d,r)  ≡ ...
              f(d)    ≡ ...
```

deve-se mostrar que

$pi.\ \forall d \in D.pre\text{-}f(d) \rightarrow \exists i \in R.pos\text{-}f(d,i)$

Por exemplo, dada a função **max**,

```
type max: ℕ x ℕ --> ℕ
     pre-max(i,j)   ≡true
     pos-max(i,j,r)≡(r=i ∨ r=j) ∧ i≤r ∧ j≤r
     max(i,j)       ≡ if i≤j then j else i
```

pi: (∀i,j ∈ ℕ).pre-max(i,j) ⇒ (∃r ∈ ℕ) ∧ pos-max(i,j,r)
≡ true ⇒ (∀i,j ∈ ℕ)(∃r ∈ ℕ) ∧ pos-max(i,j,r)
≡ (∀i,j ∈ ℕ)(∃r ∈ ℕ) ∧ pos-max(i,j,r)
≡ (∀i,j ∈ ℕ)(∃r ∈ ℕ) ∧ (r=i ∨ r=j) ∧ i≤r ∧ j≤r
≡ true

Para uma prova informal basta verificar que max(i,j) ou é i ou é j, i e j pertencem a ℕ; logo satisfaz a condição de existir um resultado no contra-domínio.

A.1.3.2 Satisfaz a especificação (ps)

A definição explícita de uma função satisfaz a especificação (definição implícita) se para cada valor do domínio que satisfaz a pré-condição a aplicação da função resulta em um valor que pertence a imagem e que ao mesmo tempo satisfaz a pós-condição. Nada se pode afirmar sobre os valores que não satisfazem a pré-condição (usualmente retorna-se undefined, na definição direta de função).

ps. (∀d ∈ D).pre-f(d) ⇒ f(d) ∈ R ∧ pos-f(d,f(d)))

Por exemplo

type max: ℕ x ℕ --> ℕ
pre-max(i,j) ≡true
pos-max(i,j,r)≡(r=i ∨ r=j) ∧ i≤r ∧ j≤r
max(i,j) ≡ if i≤j then j else i

ps: ∀i,j ∈ ℕ, pre-max(i,j) ⇒ max(i,j) ∈ ℕ ∧ pos-max(i,j,max(i,j))
≡ true ⇒ ∀i,j ∈ ℕ, max(i,j) ∈ ℕ ∧ pos-max(i,j,max(i,j))
≡ ∀i,j ∈ ℕ, max(i,j) ∈ ℕ ∧ pos-max(i,j,max(i,j))
temos, max(i,j)= if i≤j then j else i
deve-se considerar dois casos
i) com i≤j: max(i,j)=j
ii) com i>j: max(i,j)=i
i) (∀i,j ∈ ℕ) i≤j: max(i,j) ∈ ℕ ∧ pos-max(i,j,j)

≡ (∀i,j ∈ ℕ) i≤j: j ∈ ℕ ^ (j=i ∨ j=j) ∧ i≤j ∧ j≤j
≡ true
ii) (∀i,j ∈ ℕ) i>j: max(i,j) ∈ ℕ ∧ pos-max(i,j,j)
≡ (∀i,j ∈ ℕ) i<j: i ∈ ℕ ^ (i=i ∨ i=j) ∧ i≤i ∧ j≤i
≡ true

A.1.4 Decomposição de funções

A decomposição de uma função em termos de outras funções de mais baixo nível é ilustrada através de um exemplo. Supõe-se que se deseja obter uma função fatorial (**fatInt**) que receba na entrada tanto números positivos como números negativos. Esta função pode ser decomposta em duas: **fatNat** e **abs**, que respectivamente calculam o fatorial de um número positivo e o valor absoluto de um número.

```
type abs: Inteiro --> Inteiro
     pre-abs(x) ≡ true
     pos-abs(x,y) ≡ (x<0 ∧ y=-x) ∨ (x≥0 ∧ y=x)

type fatNat: Inteiro --> Natural
     pre-fatNat(x) ≡ x≥0
     pos-fatNat(x,y) ≡ y=x!

type fatInt: Inteiro --> Natural
     pre-fatInt(x) ≡ true
     pos-fatInt(x,y) ≡ (x<0 ∧ y=(-x)!) ∨ (x≥0 ∧ y=x!)
     fatInt(x,y) ≡ fatnat(abs(x))
```

A definição direta da função introduz além da obrigação de prova "satisfaz a especificação" já comentada, as obrigações de prova relacionadas com a combinação dos subcomponentes (fatNat(abs(x)). Esta combinação de componentes pode ser vista como uma ativação seqüencial, onde primeiro é ativada a função **abs** e a partir do resultado desta é ativada a função **fatNat**.

```
fatInt(x,y) ≡ a := abs(x);
              y := fatNat(a)
```

Para a combinação ser válida a pré-condição da função **fatNat** deve ser satisfeita para o valor da variável **a**. Isto pode ser verificado a partir da combinação das pré e pós-condições, como ilustrado acima. A pré-condição para a ativação da função **abs** é satisfeita pois é true.

pre-fatInt true, x, a, y ∈ Inteiro

pre-abs true
a := abs(x)
pos-abs(x,a) ≡ (x<0 ∧ a=-x) ∨ (x≥0 ∧ a=x)

pre-fatnat(a) ≡ a≥0
y := fatnat(a)
pos-fatNat y=a!

pos-fatInt (x,y) ≡ (x<0 ∧ y=(-x)!) ∨ (x≥0 ∧ y=x!)

Para mostrar que pré-condição da ativação da função **fatNat** é satisfeita mostra-se que

pos-abs ⇒ pre fatNat
≡ ((x<0 ∧ a=-x) ∨ (x≥0 ∧ a=x)) ⇒ a≥0.
≡ true

Por fim, para que a definição explícita (pela decomposição) satisfaça a especificação implícita deve-se mostrar que

(pre-abs ∧ pos-abs ∧ pre-fatnat ∧ pos-fatNat) ⇒ pos-fatInt

≡ ((x<0 ∧ a=-x) ∨ (x≥0 ∧ a=x)) ∧ a≥0 ∧ y=a! ⇒
(x<0 ∧ y=(-x)!) ∨ (x≥0 ∧ y=x!)
≡ true

O objetivo deste exemplo foi apresentar noções de como surgem as obrigações de prova na decomposição de uma função. Em [JON 86] é apresentada uma forma sistemática para

se desenvolver todo o tipo de obrigações de provas que surgem na decomposição de funções/operações.

A.1.5 Definição de estados

As noções de estado e de operações são utilizadas para facilitar especificações que usam estruturas de dados complexas. Um estado pode ser visto como uma coleção de variáveis com tipo determinado, que representa a "memória" do sistema. A noção de operação é introduzida como uma função que poder ler e mudar os valores das variáveis que representam os estados do sistema.

No exemplo anterior as variáveis candidatas a representarem os estados do sistema são UNMARRIED e MARRIED. No item anterior estas variáveis foram grafadas em letra maiúscula por serem consideradas principais.

Outra característica de um tipo de dados representado como estados e operações é a existência, não obrigatória, de uma operação de inicialização do domínio (no exemplo é a operação **Init**).

Podem ser definidas também operações de consulta às variáveis de estado; no exemplo é a operação **isMarried**.

Buscando dar uma clara noção do papel do invariante, descreve-se o problema do item anterior visto agora como operações que atuam sobre estados. Definiu-se um **invariante** ($m \cap u = \{\}$) sobre o domínio que representa os estados do sistema. No item anterior este predicado ($m \cap u = \{\}$)

fazia parte de cada pré/pós-condição, a excessão da pré-condição da operação **Init.** O invariante sobre os estados pode ser considerado como uma forma global (ou "meta") de pré e pós-condições [JON 86].

A descrição do exemplo segue a sintaxe utilizada em [JON 80]. No item que segue é apresentado o mesmo exemplo com a sintaxe utilizada em [JON 86], que parece ser mais orientada a implementação. As especificações dos algoritmos deste capítulo seguem a notação usada em [JON 80].

```
State0 :: U: Person-set  /* Unmarried */
          M: Person-set  /* Married   */

type inv-State: State0 --> Boolean
     pos-inv-State(mk-State0(u,m)) ≡ (u∩m)={}
```

Segue um exemplo de definição de operações sobre o estado St.

```
St :: State0 with  (∀s ∈ State0) inv-State(s)

type Register : STATE x Person --> STATE
     pre-Register(mk-st(u,m),p) ≡ p ∉ u ∧ p ∉ m
     pos-Register(mk-st(u,m),p,mk-st(uo,m)) ≡ uo = u U {p}

type Marry : STATE x Person x Person --> STATE
     pre-Marry(mk-st(u,m),h,w)     ≡ h ∈ u ∧ w ∈ u
     pos-Marry(mk-st(u,m),h,w,mk-st(uo,mo)) ≡
          (uo = u - {h, w}) ∧ (mo = m U {h, w})

type isMarried : STATE x Person --> Boolean
     pre-isMarried(s,p) ≡ true
     pos-isMarried(mk-st(u,m),p,b) ≡ (b = p ∈ m)

type Init:  --> STATE
     pre-Init()   ≡ true
     pos-Init((),mk-st(u,m)) ≡ u={} ∧ m={}
```

A.1.6 Obrigações de prova para estados

A obrigação de prova de implementabilidade para funções deve ser reecrita para operações, como é mostrado abaixo.

pi. $\forall d \in D.\ pre\text{-}f(d) \Rightarrow \exists r \in R.\ pos\text{-}f(d,r)$

deve ser reescrita como

pio. $(\forall \alpha \in \sum.\ inv\text{-}\sum(\alpha).\ pre\text{-}OP(\alpha) \Rightarrow$
$(\exists \alpha' \in \sum.\ inv\text{-}\sum(\alpha').\ pos\text{-}OP(\alpha, \alpha')$

pio'. $(\forall \alpha \in \sum.\ pre\text{-}OP(\alpha) \Rightarrow$
$(\exists \alpha' \in \sum.\ pos\text{-}OP(\alpha, \alpha')$

Deve-se provar (pio) para todas as operações que modificam os valores de estado do tipo de dado com excessão da operação de inicialização. A fórmula (pio) é para ser usada com um tipo de dado onde foi definido um invariante e a (pio') para um tipo de dado sem invariante. O invariante do tipo de dado deve ser verificado após a inicialização; a partir do predicado escreve-se a obrigação de prova para a operação de inicialização (pii) como:

pii. $(\forall \alpha \in \sum.\ pre\text{-}INIT(\alpha) \Rightarrow$
$(\exists \alpha' \in \sum.\ inv\text{-}\sum(\alpha').\ pos\text{-}OP(\alpha, \alpha')$

A verificação de pio para Register consiste em

$(\forall s \in STATE,\ p \in Person).\ inv\text{-}State(s).\ pre\text{-}Register(s,p)$
$\Rightarrow inv\text{-}State(so).\ pos\text{-}Register(s,so)$

$\equiv (\forall s \in STATE,\ p \in Person).$
$(\text{let } mk\text{-}State(m,u)=s \text{ in } u \cap m=\{\}).(p \notin u \wedge p \notin m)$
$\Rightarrow (\text{let } mk\text{-}State(mo,uo)=so \text{ in } (uo \cap mo=\{\}).(uo = u \cup \{p\}))$

$\equiv$ true

Informalmente: Se $(u \cap m = \{\}).(p \notin u \wedge p \notin m)$

então $(u \cup \{p\}) \cap m = \{\}$ é imediato.

A obrigação de prova para a operação de inicialização Init também é imediata.

true $\Rightarrow (u = \{\} \wedge m = \{\}).(u \cap m = \{\})$

A.1.7 Definição de estados usando sintaxe [JON 86]

É comum se encontrar uma sintaxe alternativa para a especificação de estados e operações em VDM. Segue a descrição do exemplo do item 1.2 nesta forma alternativa. Pode-se ver a convenção adotada na representação de nomes e nas palavras reservadas que indicam se a operação lê (rd) ou lê/escreve (wr) as variáveis que representam os estados.

```
State :: UNMARRIED : Person-set
         MARRIED   : Person-set
         Person    : /* qualquer representação adequada */

REGISTER(p: Person)
ext  wr u: UNMARRIED
ext  rd m: MARRIED

pre p ∉ u ∧ p ∉ m
pos uo = u U {p}


MARRY (h: Person x w: Person)
ext  wr u: UNMARRIED
ext  wr m: MARRIED

pre h ∈ u ∧ w ∈ u
pos uo= u - {h,w}  ∧  mo = m U {h,w}
```

```
ISMARRIED (p:Person ) r:Boolean
ext  rd m: MARRIED

pre true
pos (r = p ∈ m)


INIT()
ext  wr u: UNMARRIED
ext  wr m: MARRIED

pre  ≡ true
pos  ≡ uo=() ∧ mo=()
```

Usou-se o sufixo (o) para denotar as variáveis que representam os estados finais. Na notação [JON 86] as variáveis são sobrelinhadas.

A.1.8 Relação entre Funções e Operações

A partir do exemplo apresentado no item A.1.2 será exemplificado o uso das operações de um tipo de dados por outro de mais alto nível (nem sempre a abordagem adotada para o desenvolvimento de um sistema é "top-down").

É comum focalizar o desenvolvimento de um sistema nos pontos cruciais. Por exemplo, o desenvolvimento dos algoritmos para o editor concentra-se na estrutura de dados e nas operações abstratas sobre a estrutura de dados, deixando de lado os aspectos com a interface. O exemplo que segue ilustra como poderia se organizar um nível superior a partir do conjunto de operações especificadas.

Define-se um novo tipo de dado que agrupa o conjunto de operações que atuam sobre o domínio de dados comum. Nos passos de desenvolvimento devem ser verificadas

as obrigações de prova nos moldes do que foi exposto acima.

O objetivo do exemplo é ilustrar como pode ser desenvolvido o primeiro nível do editor. A estrutura do algoritmo é a mesma.

```
STATE :: STATE0 with  (∀s ∈ State0) inv-State(s)

MSG        :  "yet_married" | "yet_registered" | ""
OP         :  OpRegister | OpMarry | OpInit | OpIsMarried
PARAMETERS :: P:Person | C:(Person x Person)

OUT        :: S: STATE |
              (E: MSG, P:PARAMETERS) |
              B: Boolean

type Execute: OP x PARAMETERS x STATE --> OUT
     pre-Execute(o,p,s) ≡ inv-State(s)
     pos-Execute(o,p,s) ≡ inv-State(s)

execute(o,p,s) ≡
   case o :
   (OpRegister   --> if pre-Register(P(p),s)
                     then  (Register(P(p),s), "")
                     else  (s,("yet_registered", P(p)))
    OpMarry      --> if pre-Marry(C(p),s)
                     then  (Marry(C(p),s),"")
                     else  (s,("yet_Married",C(p)))
    OpIsMarried  --> IsMarried(C(p),s))

type Init:  --> STATE
     pre-Init()   ≡ true
     pos-Init((),mk-st(u,m)) ≡ u=() ∧ m=()

type System : (OP x PARAMETERS)-list x STATE --> STATE
     pre-System(o,s)    ≡ inv-State(s)
     pos-System(o,s,sr) ≡ inv-State(sr)
     System(o,s) ≡ if hdo ≠ <> then s
                   else System( tlo, S(Execute( hdo, s)))

type  M...  /* mais alto nível */
      pre-M... ≡ true
      pos-M... ≡
      M... ≡ ... System(o,init())
```

A.2 Manipulação da árvore sintática (AS)

A manipulação da árvore sintática (AS) é implementada a partir dos tipos de dados TpCursor, Árvore binária (TpASA), tipo que manipula a gramática, etc. Estes tipos são sequencialmente introduzidos; a definição de um tipo posterior faz uso dos que ocorrem antes.

A.2.1 O Cursor

O cursor é representado por uma lista que indica o caminho a ser seguido, a partir da raiz, para se alcançar um determinado nodo. Um valor **"in"** denota o caminhamento de um nodo pai para o filho mais à esquerda e o valor **"next"** denota o caminhamento de um nodo para o seu irmão à direita.

Exemplos baseados na figura A.1:

O cursor vazio (<>) indica o nodo raiz '?'
O cursor < "in" > indica o nodo '!Ng'
O cursor < "in", "next" > indica o nodo 'tL'
O cursor < "in", "next", "next", "in" > indica o nodo '!Dg'

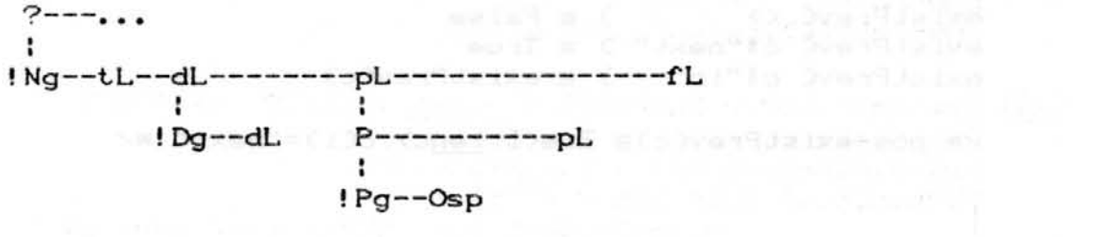


figura A.1: um exemplo de AS

O cursor é modificado com o uso de 4 operações:

In : de um nodo para o seu nodo filho mais à esquerda
Out : de qualquer nodo filho para o seu nodo pai
Next : de um nodo para seu irmão à direita
Prev : de um nodo para seu irmão à esquerda

Não se deve confundir os nomes de operações In e Next com os valores constantes "in" e "next" usadas como conteúdo de uma variável do tipo cursor. O operador infixo ‖ concatena duas listas.

```
type  In   : TpCursor  --> TpCursor
      In(c) ≡ c‖"in"

type Next : TpCursor  --> TpCursor
     Next( c )   ≡ c‖"next"

type Out  : TpCursor  --> TpCursor
     pre-Out(c)    ≡ existOut(c)
     pos-Out(c,o) ≡ c=o‖"in"‖r ∧ not existOut(r)
     Out( c‖"in" )    ≡ c
     Out( c‖"next" ) ≡ Out( c )

type Prev : TpCursor  --> TpCursor
     pre-Prev(c)    ≡ existPrev(c)
     pos-Prev(c,o) ≡ c=o‖"prev"‖r ∧ not existPrev(r)
     Prev( c‖"in" )  ≡ Prev(c)
     Prev( c‖"next") ≡ c

type existPrev : TpCursor  --> Boolean
     pos-existPrev(c) ≡ c≠<> ∧ c=α‖"next"‖β
     existPrev( <>         ) ≡ False
     existPrev( c‖"next" ) ≡ True
     existPrev( c‖"in"   ) ≡ existPrev(c)

     /* pos-existPrev(c)≡ ∃i∈(1:lenc).c(i)="next" */
```

```
type  existOut  : TpCursor  --> Boolean
      pos-existOut (c) ≡ c≠<> ∧ c=α‖"in"‖β
      existOut( <>          ) ≡ False
      existOut( c‖"in"      ) ≡ True
      existOut( c‖"next"    ) ≡ ExistOut (c)

    /* pos-ExistOut(c)≡ ∃i∈(1:lenc).c(i)="in" */
```

Outras operações

As operações **LengthIn** e **LengthNext** retornam, respectivamente, o número de "in" e o número de "next" que existe numa variável do tipo cursor. Visto sobre a estrutura da árvore significa, respectivamente, profundidade da árvore e o número de nodos irmãos à direita em relação ao nodo indicado pelo cursor.

```
type LengthIn : TpCursor  -->  Integer
     pos-LengthIn(c,i) ≡ r=card({i|i∈(1:lenc),c(i)="in"})
     LengthIn (c) ≡   if c=<>      then 0
                 else if hdc="in" then 1 + LengthIn(tlc)
                 else LengthIn(tlc)

type LengthNext:  TpCursor  -->  Integer
     pos-LengthNext(c,i) ≡ r=card({i |i∈(1:lenc),c(i)="nex"})
     LengthNext(c) ≡   if c=<>      then 0
                  else if hdc="next" then 1 + LengthNext(tlc)
                  else LengthNext(tlc)
```

A operação **less** introduz uma relação de ordem parcial sobre a estrutura da árvore sintática. Pode-se pensar como uma enumeração dos nodos da árvore em ordem crescente, partindo de cima para baixo, da esquerda para a direita.

```
_,«,_:less: TpCursor x TpCursor --> Boolean
      pos-less(c1,c2) ≡ (c1=α‖"in"‖β ∧ c2=α‖"next"‖δ)∨
                        (c1=α ∧ c2=α‖β)
```

```
_,=,_: equal: TpCursor x TpCursor --> Boolean
      pos-equal(c1,c2) ≡ (c1=c2)

/*
pos-less(a,b)≡
 (∀i:(0:lena),j:(1:lenb),i<j)
    (i>0 → a(i)=b(i)) ∧ (a(j)="in" ∧ b(j)="next"))∨
 (∀i:(0:lena)) (lenb > lena ) ∧ (i>0 → a(i)=b(i))))

 pos-equal(a,b) ≡ (lena=lenb ∧ ∀i∈(1:lena) a(i)=b(i)
*/
```

Seja a, b, c ∈ TpCursor, então:

a≪a	(reflexiva)
a≪b ∧ b≪c → a≪c	(transitiva)
a≪b ∧ b≪a → a=b	(antisimétrica)

Por exemplo para os nodos da figura A.1, página anterior, vale ?≪!Ng≪tL≪dL≪!Dg≪dl.≪pL≪P≪!Pg≪Osp≪pL≪fL.

Esta operação foi introduzida para se comparar dois cursores, nas operações de caminhamento sobre a estrutura da árvore (apresentadas no próximo item), porém pode também ser usada em provas de indução sobre a estrutura da árvore. Usa-se a seguinte regra:

```
(Seja n,m ∈ TpCursor)(n≪m → p(n) ⊢ p(m))
-----------------------------------
então ∀n ∈ TpCursor ⊢ P(n)
```

onde $\alpha \vdash \beta$ denota a partir de α inferir β

A partir da definição do operação less, se torna útil a abreviação b≫a, onde not(a≪b) ≡ b≫a.

A.2.2 Árvore binária

Inicialmente apresenta-se uma estrutura de árvore binária sobre a qual será modelada a árvore sintática. Com o objetivo de simplificar as operações de inserção e remoção, define-se uma condição no invariante para não permitir que ocorra o caso de árvore vazia (a≠emptyASA). Juntamente, para associar um cursor a uma árvore é necessário definir outra condição para não permitir que um cursor aponte para um nodo inexistente.

```
TpASA =  emptyASA | Node
Node  :: SYMB: /* qualquer definição adequada */
         IN  : TpASA
         NEXT: TpASA

TpSystem :: C: TpCursor
            A: TpASA

Sys = TpSystem with

      inv-Sys(mk-Sys(c, a))≡ isValid(c,a)


type isValid : TpCursor x TpASA --> Boolean
     isValid(c,a) ≡ c||α ∈ ( x | IN(asA(x,a))=emptyASA )
                    ∧ a ≠ emptyASA

      /* Obs: se α=<> então o cursor aponta para folha
                  senão é nodo interior a árvore  */
```

Cria-se uma operação para a inicialização da árvore; após a inicialização o invariante é satisfeito.

```
type InitASA : Symbs --> TpASA  /* Symbs: a ser definido */
      pos-InitASA(s,a)≡ a≠emptyASA
      InitASA(s) ≡ mk-Node(s, emptyASA, amptyASA)
```

Define-se também uma operação que retorna a árvore apontada por um cursor: **asA.**

```
type asA : TpCursor x TpASA --> TpASA
     asA(c,a) ≡ if c = <>          then a
                else if c="in"||r  then asA(r,IN(a))
                else if c="next"||r then asA(r,NEXT(a))
```

IN e **NEXT** são seletores de campos do objeto composto **Node. IN** e **NEXT** denotam, respectivamente, a subárvore filha e a lista de árvores irmãs da árvore passada como parâmetro.

Em função do invariante é necessário redefinir as operações **In** e **Next** que atuam sobre o cursor.

```
type ExistIn  : TpCursor x TpASA  --> Boolean
     ExistIn  (c,a) ≡ IN(asA(c,a))≠emptyASA

type ExistNext : TpCursor x TpASA  --> Boolean
     ExistNext(c,a) ≡ NEXT(asA(c,a))≠emptyASA


type In   : TpCursor x TpASA--> TpCursor
     pre-In(c) ≡ existIn(c,a)
     In( c )   = c||"in"

type Next : TpCursor x TpASA --> TpCursor
     pre-Next(c) ≡ existNext(c,a)
     Next( c )   ≡ c||"next"
```

Se estas operações forem utilizadas onde o domínio TpASA não possui um invariante com a restrição **a≠emptyASA**, esta restrição deve ser juntada às pré e pós-condições. O mesmo é válido para as operações de remoção e inserção, discutidas a seguir.

Existem três tipos de operações para alteração da estrutura da árvore: as operações de inserções de subárvores **InsertIn** e **InserNext**, as operações de remoções de subárvores **RemoveIn** e **RemoveNext**, as operações de corte de subárvores **CutIn** e **CutNext**.

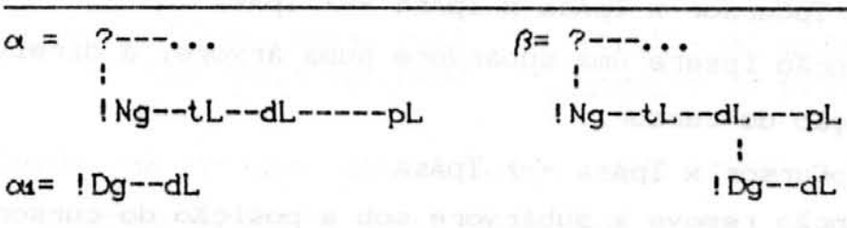


figura A.2: ex: **InsertIn**(<"in", "next", "next">, α, α1) = β

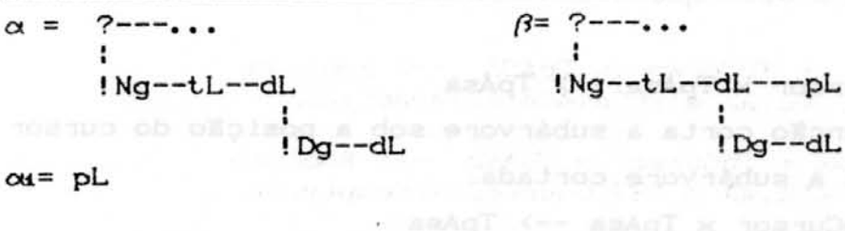


figura A.3: ex: **InsertNext**(<"in","next","next">, α, α1) = β

A figura A.2 ilustra a operação **InsertIn**; na mesma figura: **RemoveIn**(<"in", "next", "next">, β) = α
e **CutIn** (<"in", "next", "next">, β) = α1

A figura A.3 ilustra a operação **InsertNext**; na mesma figura: **RemoveNext**(<"in", "next", "next">, β) = α
e **InsertNext**(<"in", "next", "next">, β) = α1

Segue a descrição informal destas operações:

InsertIn : TpCursor x TpAsa x TpAsa --> TpAsa

Esta função insere uma subárvore numa árvore, sob a posição do cursor.

InsertNext : TpCursor x TpAsa x TpAsa --> TpAsa

Esta função insere uma subárvore numa árvore, à direita da posição do cursor.

RemoveIn : TpCursor x TpAsa --> TpAsa

Esta função remove a subárvore sob a posição do cursor; retornando a árvore original, sem a subárvore.

RemoveNext : TpCursor x Tpasa --> TpAsa

Esta função remove todas as subárvores irmãs, à direita, da subárvore sob a posição do cursor.

CutIn : TpCursor x TpAsa --> TpAsa

Esta função corta a subárvore sob a posição do cursor e retorna a subárvore cortada.

CutNext : TpCursor x TpAsa --> TpAsa

Esta função corta a lista de subárvores irmãs, à direita, da subárvore sob a posição do cursor e retorna a lista de subárvores cortada.

Os sufixos "In" e "Next" dos identificadores das operações indicam se a subestrutura a ser manipulada é, respectivamente, a subárvore filha ou as subárvores à direita do nodo apontado pelo cursor.

Inserções

```
type InsertIn   : TpCursor x TpAsa x TpAsa  -->  TpAsa
     pre-InsertIn(c,a1,a2)   ≡ IN(asA(c,a1)) = emptyASA
     pos-InsertIn(c,a1,a2,o) ≡ IN(asA(c,o))  = a2
     InsetIn(c,a1,a2) ≡ μ(asA(c,a1),[IN --> a2])

type InsertNext : TpCursor x TpAsa x TpAsa  -->  TpAsa
     pre-InsertNext(c,a1,a2) ≡ NEXT(asA(c,a1)) = emptyASA
     pos-InsertNext(c,a1,a2) ≡ NEXT(asA(c,o))  = a2
     InsertNext(c,a1,a2) ≡ μ(asA(c,a1),[NEXT --> a2])
```

/* a operação μ substitui um componente de uma estrutura composta */

Remoções

```
type CutIn      : TpCursor x TpAsa   -->  TpAsa
     pos-CutIn(c,a,o) ≡ IN(asA(c,a))=o

type CutNext    : TpCursor x TpAsa   -->  TpAsa
     pos-CutNext(c,a,o) ≡ IN(asA(c,a))=o

type RemoveIn   : TpCursor x TpAsa   -->  TpAsa
     pos-RemoveIn(c,a,o) ≡ IN(asA(c,a))=emptyASA
     RemoveIn(c,a) ≡ μ(asA(c,a),[IN --> emptyASA])

type RemoveNext : TpCursor x TpAsa   -->  TpAsa
     pos-RemoveNext(c,a,o) ≡ NEXT(asA(c,a))=emptyASA
     RemoveNext(c,a) ≡ μ(asA(c,a),[NEXT --> emptyASA])
```

Folhas

Para o caminhamento sobre as folhas da árvore é definido um tipo de dados folha (TpLeaf) com diversas operações:

isLeaf: retorna verdadeiro se o nodo apontado pelo cursor é uma folha.

inLeaf: a partir de uma nodo no meio da árvore se posiciona na folha (mais próxima, descendo).

nextLeaf: o cursor posiciona-se na próxima folha.

prevLeaf: posiciona-se na folha anterior.

right: posiciona-se na folha mais à direita.

left: posiciona-se na folha mais à esquerda.

TpLeaf :: TpCursor

with inv-TpLeaf(c,a) ≡ isLeaf(c,a) ∧ a≠emptyASA

(TpLeaf é o domínio que contém apenas os cursores da fronteira da árvore)

type isLeaf: TpCursor x TpASA --> Boolean
pre-isLeaf(c,a) ≡ a≠emptyASA
isLeaf(c,a) ≡ IN(asaA(c,a)) = emptyASA

type inLeaf: TpCursor x TpASA --> TpLeaf
pre-inLeaf(c,a) ≡ existIn(c,a)
pos-inLeaf(c,a,o) ≡ IN(asA(o,a))=emptyASA

type nextLeaf: TpLeaf x TpASA --> TpLeaf
pre-nextLeaf(c,a) ≡ (∃i ∈ TpLeaf, i»c)
pos-nextLeaf(c,a,o) ≡ o»c ∧ not (∃i ∈ TpLeaf, o»i»c)

type prevLeaf: TpLeaf x TpASA --> TpLeaf
pre-prevLeaf(c,a) ≡ (∃i ∈ TpLeaf, i«c)
pos-prevLeaf(c,a,o) ≡ o«c ∧ not (∃i ∈ TpLeaf, o«i«c)

type right: TpASA --> TpCursor
pre-right(c,a) ≡ (∃i ∈ TpLeaf, i»c)
pos-right(a,o) ≡ (∀i ∈ TpCursor, o»i)

type left: TpASA --> TpCursor
pre-leaf(c,a) ≡ (∃i ∈ TpLeaf, i«c)
pos-left(a,o) ≡ (∀i ∈ TpCursor, o«i)

É fácil mostrar que as operações prevLeaf e nextLeaf são inversas.

pre-nextLeaf(c,a) → prevLeaf(nextLeaf(c,a),a)=c
pre-prevleaf(c,a) → nextLeaf(prevLeaf(c,a),a)=c

A.2.3 O tipo gramática

Para se introduzir a árvore sintática é necessário fazer referência a algumas operações de um tipo de dado que descreve uma gramática. Este tipo não é desenvolvido aqui; mais detalhes são encontrados em [ESP 89a].

Do tipo de dado que implementa a gramática são utilizadas duas funções: **Select** - esta função seleciona uma alternativa a partir do símbolo não terminal e do "primeiro" terminal de uma de suas alternativas; **First** - retorna uma lista com os símbolos terminais "primeiros" de um não terminal. Estas operações são exemplificadas abaixo.

```
ex:  P1 --> a B | b A    [P1 --> <<a B>, <b A>>]
     P2 --> X y | ^      [P2 --> << x y>>, <^>]

Select( a, P1 ) = <a, B>
Select( ^, P2 ) = <^>

First(P1) = < a, b >
First(P2) = < ^ > U First(X)
```

Segue uma definição para o domínio de dados que representa a gramática.

```
G = N --> L
L = R-list
R = Symbs-list
Terminals = /* terminais da gramática */
NonTer = /* não terminais */
Symbs  = Ter U NonTer
Ter    = Terminals U <^>

inv-G(g) ≡ isValidG() ∧ isLL1(g)
```

```
isValidG() ≡ Ter ∩ NonTer = { } ∧
             (∀r ∈ R, len(r) ≥ 1) ∧
             (∀l ∈ L, len(l) ≥ 1)
/*
um símbolo ou é apenas terminal ou apenas não terminal;
e cada nome de produção é associado a pelo menos uma
alternativa;
e cada alternativa contém pelo menos um símbolo ;
*/

isLL1(g) ≡ Let N dom g, L rng g in
           (∀n∈N, l∈L, [n --> l])
            (∀i ∈ {1:len(l)})
             (∀p, q ∈ {1:len(l(i))})
               p ≠ q ⇒ first(l(i)(p))∩first(l(i)(q))={}
/*
e a gramática deve ser do tipo LL(1) - os primeiros
símbolos terminais de duas alternativas distintas devem ser
distintos)
*/
```

As operações Select, First não são desenvolvidas. Segue apenas a definição dos domínios. Na realidade estas operações deviam ter no domínio também a gramática; no seu uso nos algoritmos que seguem, fica implícito este parâmetro.

```
type Select: Ter x NonTer --> Symbs-list
type First:  NonTer       --> Ter-set
```

A.2.4 O tipo árvore sintática

A representação da árvore sintática é mapeada sobre a estrutura da árvore binária apresentada anteriormente. A relação entre a árvore sintática e a gramática é representada pelos invariantes inv-G2 e inv-G3, esboçados abaixo.

```
inv-G2(a,g) ≡ (∀i∈TpCursor)
                let s=sons(i,a)
                in (Select(hds,symb(i,a)) = s)

/* qualquer relação pai-filhos da árvore deve ter sido
gerada por uma produção da gramática: compara-se a lista de
filhos com a lista da produção */

type sons: TpCursor x TpASA --> Symbs-list
pos-sons(c,a,o) ≡
      Let s=(<symb(i,a),i> | c≪i ∧ out(i)=c) ∧
          elems(l) = s ∧
          l(k), l(j), k<j ⇒ l(k)=<_,y>, l(j)=<_,y> e x≪y
      in (∀i(1:lens), l(i)=<o(i),_>

sons(c,a) ≡ sonsNext(in(c,a))
sonsNext(c,a) ≡ if existNext(c,a)
                 then Symb(c,a)‖sonsNext(next(c,a),a)
                 else <>

/* sons: cria uma lista dos símbolos filhos de um nodo da
árvore */

inv-G3(a,g) ≡ (∀i∈TpLeaf)(symb(i,a) ∈ TerU(^)

/* qualquer folha deve ser do tipo terminal ou vazio */
```

Estes invariantes garantem que a estrutura da árvore é preservada em relação às regras gramaticais.

Criação de nodos na árvore sintática

Para criar nodos sobre a árvore binária que satisfazem as regras gramaticais define-se a operação **InsertProd**. Para se verificar que as pré-condições das suboperações são satisfeitas é suficiente tomar **a≠emptyASA** (parte do invariante) junto com a pré-condição da operação.

```
type InsertProd: TpCursor x TpASA x Ter --> TpASA
     pre-InsertProd(c,a,t) ≡ isLeaf(c,a,t)
     InsertProd(c,a,t) ≡
       Let p =Select(Symb(c,a),t)
           ao=InsertIn(c,a, mk-Node(hdp,emptyAsa,emptyASA))
      in InsNextSymb(in(c),ao,tlp)


type InsNextSymb: TpCursor x TpASA x Symb-list --> TpASA
     InsNextSymb(c,a,p) ≡
     if s = <> then a else
     let ao=InsertNext(c,a,mk-Node(hdp,emptyAsa,emptyASA))
     in InsNextSymb(next(c),ao,tlp)
```

BIBLIOGRAFIA

[AHO 86] AHO, A.V.; SETHI, R.; ULLMAN, J. D. **Compilers: Principles, Techniques, and Tools**. Reading, Addison-Wesley, 1986.

[BAR 87] BARFORD, L. A.; ZANDEN, B. T. V. **Attribute grammars constraint-based graphics systems**. Ithaca, Cornell University, June 1987.

[BJO 88] BJORNER, D. **Software architectures and programming system design**. Denmark, Department of Computer Science/Technical University of Denmark, 1988.

[DAT 84] DATE, C.J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro, Campus, 1986.

[DER 88] DERANSART, P.; JOURDAIN, M.; LORHO B. **Attribute grammars: definitions, systems and bibliography**. Berlin, Springer Verlag, 1988. Lecture Notes in Computer Science 323.

[ENG 88] ENGELS, G; LEWERENTZ, C.; SCHAFER, W. Grap grammar engineering: a software specification method. INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3. Warrenton Dec. 2-6, 1986. Proceedings. Berlin, Springer-Verlag, 1987. p.186-201 Lecture Notes in Computer Science 291.

[ESP 89a] ESPERANÇA, L.; FAVERO, E. L.; PRICE, R. T. **Um algoritmo de reconhecimento incremental**. Porto Alegre, CPGCC-UFRGS. set.1989. RI 137.

[ESP 89b] ESPERANÇA, L.; **Um Interpretador de gramáticas de atributos**. Porto Alegre, CIC/UFRGS, Dezembro, 1989. (Trabalho de diplomação)

[FAV 87] FAVERO, E. L. **Uma implementação de um núcleo gerador de editores dirigidos por sintaxe em microcomputador**. Porto Alegre, CIC/UFRGS, Julho, 1987. (Trabalho de Diplomação)

[FAV 88] FAVERO, E.L.; AGUSTINI, A.; AZEREDO, P.A.; PRICE, R.T. Os comandos de navegação em um editor dirigido por sintaxe. In: REUNIÃO DE TRABALHO E COLETANEA DE RESULTADOS DE PESQUISA 4. out. 1988. Projeto Estra. São Paulo, SID-Informática, 1988. v.1 p. 307-317.

[GOT 82] GOTTLER, H. Attribute graph grammars for graphics In: INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE 2; Hous Ohrbeck, Oct. 4-8, 1982. **Procedings.** Berlin, Springer-Verlag, 1982. p. 130-142. Lecture Notes in Computer Science 153.

[GOT 88] GOTTLER, H. Graph grammars and diagram editing. INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3o, Warrenton Dec. 2-6, 1986. p. 216-231. **Proceeding.** Berlin, Springer-Verlag, 1987. p. 216-231. Lecture Notes in Computer Science 291.

[HAL 88] HALEWOOD, K.; WOODWARD, M. R. NSEDIT: A syntax-directed editor and testing tool based on Nassi-Shneiderman Charts. **Software Practice and Experience**, Sussex, 18(10):987-998, Oct.1988.

[HEK 87] HEKMATPOUR, S.; WOODMAN, M. Formal specification of graphical notations and graphical software tools. In: EUROPEAN SOTFWARE ENGINEERING CONFERENCE, Strasbourg, Sept. 9-11, 1987. **Procedings.** Berlin, Springer-Verlag, 1987. p. 297-305. Lecture Notes in Computer Science 289.

[HOO 87] HOOVER, R. **Incremental graph evaluation.** Ithaca. Cornell University, 1987. (Phd Thesis)

[HOR 86] HORWITZ, S.; TEITELBAUM, T. Generating editing environments based on relations and attributes. **Acm transactions on programming languages and systems.** New York, 8(4):577-608, Oct. 1986.

[JON 80] JONES, C. B. **Software development: a rigorous Aproach.** London, Prentice-Hall International. 1980.

[JON 86] JONES, C. B. **Systematic software development using VDM.** London, Prentice-Hall International. 1986.

[KAP 87] KAPLAN, SIMON M. **Incremental Attribute Evaluation on Node-Label Controlled Graphs.** Urbana, University of Illinois, May, 1987. Technical Report no. UIUCDCS-K-87-1309.

[KAS 82] KASTENS.; U.; HUTT, B.; ZIMERMANN, E. **GAG: a practical compiler generator**. Berlin, Springer-Verlag, 1982. Lecture Notes in Computer Science 141.

[KOL 88] KOLIVER, C. **Um mecanismo de avaliação incremental de gramáticas de Atributos**. Porto Alegre, CIC/UFRGS 1988. (Trabalho de Diplomação)

[MAR 87] MARTIN, J. **Recommended diagramming standards for analysts and programmers: a basis for automation**. Englewood Cliffs, Prentice-Hall, 1987.

[MEL 89] MELO, W.L.M. **Uma proposta de um editor diagramático generalizado**. Porto Alegre, CPGCC da UFRGS, Março, 1989. (Dissertação de Mestrado)

[MRA 89] MRACK, F. R. **Protótipo de um dicionário de dados para um editor diagramático generalizado**. Porto Alegre, CIC/UFRGS, 1989. (Trabalho de Diplomação)

[NAG 88] NAGL, L. A software development environment based on graph technology. In: INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3., Warrenton Dec. 2-8, 1986. **Proceedings**. Berlin, Springer-Verlag, 1987. p. 458-477. Lecture Notes in Computer Science 291.

[PET 87] PETERS, L. **Advanced structured analisys and design**. New Jersey, Prentice-Hall, 1987.

[PRI 84] PRICE, R.T. **A Prototype syntax driven language editor**. Brighton, University of Sussex, 1984. (Tese de Doutorado).

[SIL 89] SILVA, M.S. **Um formatador de diagramas**. Porto Alegre, CIC/UFRGS, 1989. (Trabalho de Diplomação).

[SNE 85] SNELTING, G. Experience with the PSG - Programming System Generator. In: INTERNATIONAL JOIN CONFERENCE ON THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT (TAPSOFT). Berlin, March, 25-29, 1985. **Proceeding**. Berlin, Springer- Verlag, 1985. p. 148-162. Lecture Notes in Computer Science 186, v. 2: Formal Methods and Software Development.

[REP 83] REPS, T.; TEITELBAUM, T.; DEMERS, A. Incremental Contex-dependent Analysis for Language-based Editors. **Acm transactions on Programming Languages and Systems**, New York, 5(3): 449-477, July 1983.

[REP 87] **REPS, T.; TEITELBAUM, T. The synthesizer generator: reference manual.** New York, Departament of Computer Science/Cornell University, July, 1987.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um editor orientado por estrutura para
linguagens diagramáticas

Dissertação apresentada aos Srs.

Prof. Dr. Carlos Alberto Heuser

Prof. Dr. Dalton José Nunes

Prof. Dr. Roberto Tom Price

Visto e permitida a impressão

Porto Alegre, 19/09/90

Prof. Dr. Roberto Tom Price
Orientador

Prof. Dr. Ricardo Augusto da L. Reis
Coordenador do Curso de Pós-Graduação
em Ciência da Computação