Tabelas de Decisão e
Implementação do Gerador I-M-E

Satoshi Nagayama

DISSERTAÇÃO APRESENTADA AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA DA
UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE

MESTRE EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Valdemar W. Setzer**

—São Paulo, Dezembro de 1990—

Agradecimentos

Ao Prof. Dr. Valdemar W. Setzer por sua orientação, estímulo, dedicação, idéias e sugestões.

Ao Banco do Brasil pelo incentivo.

Conteúdo

Prefácio	v
1 Tabelas de Decisão	1
1.1 Introdução	1
1.2 Histórico	2
1.3 Estrutura da Tabela de Decisão	4
1.3.1 Interpretação da Tabela de Decisão	4
1.3.2 Condições e Ações	5
1.3.3 Regras de Decisão	6
1.3.4 Linhas de Condição Limitadas e Estendidas	7
1.3.5 Entradas de Condição e Ação	8
1.3.6 Linhas de Ação estendidas	8
1.3.7 Tabelas Limitadas, Estendidas e Mistas	9
1.3.8 Condições Dependentes e Independentes	10
1.3.9 Entradas de Condição Implícitas	11
1.4 Desdobramento de “_”	12
1.5 Combinação ou Intercalação de Regras	13
1.6 Função-“And”	14
1.7 Regras e Colunas	16
1.8 Regras Ambíguas	17
1.9 Ambigüidade Aparente	19
1.10 Eliminação de Ambigüidades	20
1.11 Ambigüidade em Tabelas Mistas	22
1.12 Conversão de Tabelas Mistas em Limitadas	24
1.13 Seleção de Regras	26
1.14 Divisão de uma Tabela de Decisão por uma Condição	27
1.15 Fatoração de Tabelas de Decisão por uma Linha de Condição	31
1.16 Completude de uma Tabela de Decisão	33
1.16.1 Completude em Tabelas com condições dependentes	36
1.17 “Else Rule”	37
1.17.1 Estratégia para a “else rule”	38
1.18 Linguagem de Tabela de Decisão	41

2	Tradução de Tabelas de Decisão	43
2.1	“Switch method”	44
2.2	Método de Teste Sequencial	51
2.3	Método de Máscara	51
2.3.1	Máscara de Regra	52
2.3.2	Máscara de Regra com Interrupção	55
2.3.3	Máscara de Condição	71
2.4	Método de Fatorações Sucessivas	73
2.4.1	Número de STP’s	79
2.4.2	Estrutura de Dados para o Algoritmo	83
2.4.3	Linhas de Condição irrelevantes	83
2.4.4	Descrição do Algoritmo	87
2.4.5	Escolha da Condição para Teste	92
2.4.6	Método trivial	96
2.4.7	Método I de Pollack	96
2.4.8	Método II de Pollack	98
2.4.9	Método Esquerda para Direita	99
2.4.10	Heurísticas baseadas na Teoria da Informação	102
2.4.11	Heurísticas com “Look-Ahead”	103
2.4.12	Outras heurísticas	104
2.4.13	Outros Aspectos do Algoritmo de Fatorações Sucessivas	105
2.5	Algoritmos Ótimos	106
2.5.1	Método de Busca Exaustiva Parcial	106
2.5.2	Método de Programação Dinâmica	111
3	Mais poder de Notação às Tabelas de Decisão	118
3.1	Conjuntos de Valores	118
3.2	Ações Comuns de Preparação	123
4	Um Pré-Processador de Tabelas de Decisão	124
4.1	Funcionamento	124
4.2	Linguagem e Limites de um Grupo de Decisão	125
4.3	Especificações de uma Tabela de Decisão	126
4.3.1	Nome da Tabela de Decisão	127
4.3.2	Ações Comuns de Preparação (opcional)	127
4.3.3	Definição de Conjuntos de Valores (opcional)	128
4.3.4	Condições	129
4.3.5	Ações	130
4.3.6	Fim da Tabela	132
4.4	Especificações das Transformações	134
4.5	Código Gerado e Comentários	135
4.6	Um exemplo simples	136

5	O Gerador I-M-E	139
5.1	Algumas Palavras sobre o Método I-M-E	139
5.1.1	Partes ou Etapas	140
5.1.2	Fases	141
5.1.3	Sub-processos de Agrupamento	143
5.2	Declarações	144
5.2.1	Arquivos	145
5.2.2	Tabelas	148
5.2.3	Telas	150
5.2.4	Sequências	155
5.2.5	Variáveis de Sub-processos de agrupamento	156
5.2.6	Variáveis simples e auxiliares	157
5.2.7	Sinônimos	158
5.3	Comandos de Seleção	159
5.3.1	Ítems do Comando de Seleção	160
5.3.2	Condição de Seleção	160
5.3.3	Condição de Acesso	161
5.3.4	Encerramento da Etapa Principal	161
5.3.5	Código para Comando de Seleção	162
5.3.6	Código para Seleção de Arquivos	164
5.3.7	Código para Seleção de Tabelas	165
5.3.8	Código para Seleção de Sequências	166
5.3.9	Código para Seleção de Telas	167
5.3.10	Código para seleção de "strings"	171
5.3.11	Agrupamentos	171
5.3.12	Encaixamento de Comandos de Seleção	175
5.3.13	Seleção através de Sub-processos	176
5.4	Tabelas de Decisão na Fase de Escolha	177
5.4.1	Entradas de Condição	177
5.4.2	Descrição das Ações	178
5.4.3	Ações Comuns de Preparação	178
5.5	Comandos de Transformação	179
5.5.1	Comando de Inicialização	180
5.5.2	Comando de Troca	181
5.5.3	Comando de Atribuição	181
5.5.4	Outros Comandos	185
5.5.5	Expressões	186
5.6	Tabela de Identificadores do Gerador	186
5.7	Pilha Auxiliar	189
6	Exemplos	192
6.1	Exemplo 1	192
6.2	Exemplo 2	198
6.3	Exemplo 3	200
6.4	Exemplo 4	205

Conteúdo	iv
6.5 Exemplo 5	209
Comentários Finais	216
Apêndice	219
Grafo Sintático	220
Bibliografia	231

Prefácio

Este trabalho é composto de uma parte teórica e uma parte prática. A parte prática trata da implementação do Gerador de programas I-M-E, e a parte teórica trata de tabelas de decisão que são utilizadas como um dos mecanismos de especificação ao gerador. Este tratamento teórico é feito para subsidiar o algoritmo de conversão de tabelas que é empregado na parte prática, tornar o texto auto-contido e resumir a literatura sobre tabelas de decisão. Embora existam outros resumos [Po-Hi-Ha 71, Po 74, Mo 82], em especial o excelente trabalho de H. Takaoka [Ta 76], o nosso é mais atual e completo. Neste texto apresentamos um elemento introduzido em [Se-Ma 88] (definição de conjuntos de valores), que não encontramos na literatura, e também procuramos enfatizar a apresentação de elementos que dão sustentação aos algoritmos. Para isto, são apresentados alguns teoremas dentre os quais, os de número 1, 2 e 11 podem ser encontrados em [Po-Hi-Ha 71], sendo que a demonstração do teorema 2 é análoga à apresentada na referência, a do teorema 1 é ligeiramente diferente e a do teorema 11 é totalmente diferente. Os demais teoremas, embora simples, não foram encontrados na literatura consultada a menos de uma breve referência a um deles (teorema 22) [Ki-Jo 75].

O capítulo 1 contém a descrição de tabelas de decisão usuais. O capítulo 2 contém um resumo sobre os algoritmos de conversão existentes. O capítulo 3 apresenta basicamente a definição de conjuntos de valores e o modo como são incorporados à descrição da tabela de decisão. O capítulo 4 descreve a implementação de um pré-processador de tabelas de decisão cujo algoritmo básico de conversão por fatorações sucessivas é utilizado no gerador I-M-E. O capítulo 5 descreve a implementação desse gerador. O capítulo 6 apresenta exemplos de especificação e de código produzido pelo nosso gerador. Finalmente, o apêndice contém o grafo sintático da linguagem aceita pelo gerador.

As especificações do gerador I-M-E foram fornecidas por V. W. Setzer; o princípio de geração de código com retornos empilhados foi também de sua autoria. O restante foi projetado por nós.

Capítulo 1

Tabelas de Decisão

1.1 Introdução

Existem diversas maneiras de apresentar um algoritmo. Uma delas é descrever em linguagem informal as suas operações. É claro que esta forma é propícia para uma comunicação entre pessoas, mas inadequada para uma comunicação entre uma pessoa e uma máquina. Para esta finalidade seria desejável um mecanismo mais conciso de comunicação; ele deverá necessariamente ser formal, pois o computador é máquina algorítmica. Ao mesmo tempo, gostaríamos que a descrição do algoritmo não prejudicasse a sua compreensão.

As linguagens de programação existentes são formas de comunicação entre pessoas e computadores, umas melhores que outras em termos de legibilidade. Elas oferecem diversos mecanismos para que os usuários possam executar as suas tarefas, ou seja, elas possuem um repertório de comandos que tentam suprir as suas necessidades. Parte desses mecanismos tratam de decisões. Uma decisão é uma escolha entre alternativas, isto é, dado um conjunto de ações, precisamos escolher quais delas devem ser executadas num certo instante. Esta escolha é subsidiada por um conjunto de condições que propiciam um conjunto de situações possíveis. As linguagens de programação de alto nível efetivam as decisões através dos comandos *if* e *case*. São comandos simples, quando se apresentam isoladamente, mas podem contribuir para uma perda significativa da legibilidade quando vários encadeamentos são necessários. O mecanismo aqui apresentado para as decisões é a Tabela de Decisão, que consiste basicamente numa tabela de formato especial, codificando um conjunto de situações, com base num conjunto de condições. Cada situação, que é uma possível combinação das condições dentro do universo do problema, especifica um conjunto de ações que deve ser providenciado.

[A grande virtude da Tabela de Decisão é permitir a visualização do quadro de decisões de forma simplificada e mais imediata [Va 85].] Este mecanismo de especificação de decisões é largamente utilizado no ambiente administrativo pela sua facilidade e concisão. Trazendo-o para a área de desenvolvimento de programas traz de pronto a perspectiva de uma maior aproximação entre os usuários dos sistemas de processamento de dados e as pessoas responsáveis pela sua implementação, diante da utilização de ferramentas comuns na especificação dos problemas. Consideramos que o fato de uma pessoa desconhecer a linguagem da outra é uma sensível dificuldade no relacionamento entre elas. [Além disso, as tabelas de decisão propiciam o encorajamento do desenvolvimento de programas modulares e mais estruturados, pois as ações especificadas

em cada regra se prestam a refletir conjuntos de comandos logicamente interligados.

Em razão da clareza proporcionada pelas tabelas de decisão, elas constituem uma ferramenta importante no desenvolvimento de sistemas, desde o projeto, passando pela implementação e testes, até a documentação, para qualquer tipo de aplicação e não apenas para a área comercial [Le 83]. A parte prática deste trabalho inclui uma proposta de incorporação desta ferramenta no trabalho de programação.

Quando o problema impõe a consideração simultânea de um número razoável de condições, pode-se observar, na prática, a dificuldade em descrever as situações possíveis [Le 82a] por encadeamentos de comandos condicionais fornecidos pelas linguagens de programação usuais. Esses encadeamentos normalmente se misturam a outros comandos, prejudicando ainda mais o entendimento do código. A construção de uma tabela de decisão, ao contrário, é um processo sistemático de enumeração separada de todas condições e ações envolvidas num determinado problema [Mi 77]. As condições são enumeradas em linhas e as situações proporcionadas por estas condições são dispostas em colunas. Assim, é uma estrutura fácil de construir e modificar e que não apresenta problemas para o seu entendimento. As colunas permitem uma visualização clara das ações que são executadas e da sequência em que isto ocorre. Além disso, as tabelas propiciam uma separação física das condições que atuam no problema das ações que devem ser executadas em cada uma das situações, [introduzindo uma estruturação de algoritmos em nível mais alto do que o da programação estruturada.]

1.2 Histórico

início

As tabelas de decisão são usadas largamente fora do ambiente de processamento de dados. O uso desta técnica nesta área teve início com o *Projeto de Sistemas Integrados* da General Electric em novembro de 1957 [Po-Hi-Ha 71] que tinha como objetivo acompanhar e controlar o processo de produção de um produto desde a recepção do pedido até o acabamento final. Devido a dificuldade em expressar a lógica complexa do problema através de mecanismos até então utilizados, o grupo responsável desenvolveu as tabelas de decisão e um processo computacional para automatizar a tradução da tabela. A estrutura da tabela introduzida recebeu o nome de TABSOL [St 71], que era uma linguagem de programação por tabelas de decisão, implementada por interpretadores [Me-Ba 77]. Independente e quase simultaneamente a Sutherland Company desenvolveu uma tabela de decisão diferente no formato, mas idêntico no conceito e que tinha como objetivo apenas a constituição de uma ferramenta para o desenvolvimento de sistemas e de documentação, sem propor um método automático de tradução, cabendo ao programador a tarefa de converter manualmente a tabela de decisão em comandos de uma linguagem computacional [Po-Hi-Ha 71].

Com a divulgação dos trabalhos da General Electric, houve uma sequência enorme de trabalhos no desenvolvimento de processadores de tabelas de decisão. Estes são nomes genéricos de programas que automatizam o tratamento e tradução de tabelas de decisão. A IBM desenvolveu isoladamente um processador para o IBM-1401 e, em conjunto com a Boeing Company, um outro para o IBM-7080 e mais outro para o IBM-7090, o FORTAB, em conjunto com a Randy Corporation. Todos produziam fontes em FORTRAN. A Insurance Company of North America produziu um sistema de tabelas de Decisão chamado LOBOC que recebia como entrada as expressões condicionais representando as diversas situações possíveis de um problema e

1983 15 1983-05

como saída, um programa em linguagem “assembler” e uma imagem da tabela de decisão como documentação, construída a partir das entradas que foram submetidas [Me-Ba 77].

Em 1962 [Me-Ba 77, Mc 68], a CODASYL organização que desenvolveu o COBOL, trabalhou sobre tabelas de decisão e acabou propondo uma linguagem para elas, chamada DETAB-X (Decision Table, Experimental), cujo formato tornou-se o padrão.

Em 1965, um grupo da Association for Computing Machinery desenvolveu, a partir das experiências obtidas com o DETAB-X, um pré-processador¹ para tabelas de decisão chamado DETAB/65 que converte tabelas de decisão em código fonte COBOL. Uma explicação detalhada de sua implementação e do código produzido pode ser encontrado no artigo de Callahan & Chapman [Ca-Ch 67] e de Silberg [Si 71]. Embora implementado em diversas máquinas e distribuído livremente, sua utilização decresceu até desaparecer completamente em razão do algoritmo de conversão ineficiente e falta de manutenção [Po-Hi-Ha 71]. Depois deste, surgiram alguns outros com a mesma filosofia para o FORTRAN. Resumindo, a história das tabelas de decisão tem os seus primeiros trabalhos entre 1957 a 1960; em 1961 e 1962, surgiu um grande número de trabalhos no desenvolvimento de processadores de Tabelas de Decisão, seguidos por dois anos de relativa inatividade. Iniciado pelo DETAB/65, a partir de 1966, chegou a vez dos pré-processadores. Um dos mais recentes, o sistema *Logic Gem* [Dl 89, Mc 89], pode traduzir para diversas linguagens, tais como Pascal, C, BASIC, FORTRAN e dBASE.

Uma questão interessante é tentar saber porque, já que apresentam vantagens, as tabelas de decisão não se tornaram ferramentas de uso popular na comunidade de programadores e analistas. Vamos citar algumas possíveis causas enumeradas na bibliografia [Po-Hi-Ha 71, Ch 83, Le 83, Mc 85, Mc 86]:

1) As abundantes publicações sobre a matéria não se destinaram aos programadores e analistas comerciais. Os artigos veiculados em revistas técnicas, tratando de discussões altamente especializadas sobre métodos de conversão, exigiam pré-requisitos por parte dos leitores. Todos os fabricantes de computadores incluem em seus cursos de programação, os fluxogramas como método de desenvolvimento de sistemas, mas nunca uma introdução às tabelas de decisão. Experiências dão conta de que melhores resultados são obtidos se as tabelas de decisão são ensinadas no início do aprendizado dos analistas e programadores.

2) Enquanto o uso de fluxograma conduz a um modelo de tomada de decisão, onde os testes são seguidos de ações e estes seguidos de outros testes, o uso de tabelas de decisão requer a enumeração em separado de todas as condições presentes num problema e a análise das possibilidades sobre elas. Isto leva a uma necessidade de se pensar no problema antes de se começar a codificar a sua solução. Esta distinção requer uma postura diferente diante dos problemas e é natural que pessoas acostumadas a um tipo de visão, resistam a outras maneiras de enfrentar as questões.

3) Segundo a bibliografia acima, faltaram tradutores de tabelas de decisão para a comunidade de processamento de dados. Os processadores desenvolvidos antes do DETAB/65 foram distribuídos de forma limitada, fazendo com que muitas tabelas de decisão fossem traduzidas manualmente para uma linguagem de programação. É razoável que a ausência de mecanismos automáticos de tradução tenha desinteressado os programadores pelas tabelas de decisão.

4) Falsa impressão de que as tabelas de decisão não são suficientemente poderosas aliada à ausência de um método bem desenvolvido de programação que incorpore este mecanismo de

¹Classe de processadores que traduzem tabelas de decisão para linguagem de alto nível.

especificação.

5) Falta de instrumentos de apoio à construção de tabelas de decisão, como editores voltados para esta finalidade que automatizem várias etapas do processo, livrando o usuário de algumas tarefas tediosas e mecânicas na elaboração de tabelas. De um modo geral, não existe um elenco de ferramentas disponíveis para auxiliar na construção de tabelas de decisão comparáveis às existentes para linguagens algorítmicas.

Podemos verificar que existem sistemas complexos desenvolvidos com a utilização de tabelas de decisão, como sistemas de controle de processos [Ra 87, Ku-Ch 83, Ni-Mc 86], detecção de falhas em equipamentos nucleares e diagnósticos médicos, mas não podemos afirmar que as tabelas tenham se tornado populares entre os profissionais de processamento de dados.

1.3 Estrutura da Tabela de Decisão

Como visto, o uso de tabelas de decisão teve início em 1957. Deste então, surgiram várias formas para a sua apresentação. Algumas apresentam formato complexo [Mu 84, Mi-Ch 89], na tentativa de capturar propriedades que ultrapassam os aspectos relacionados com decisões. A despeito desta diversidade, consagrou-se como quase unânime, a estrutura sugerida pelo exemplo da figura abaixo [Po 74, Me-Ba 77, Po-Hi-Ha 71], com pequenas variações.

$X > 0$	Y	N	Y	-
$Y = 10$	N	Y	Y	-
A	1	2	>3	<0
Ação1	1		2	1
Ação2		1	3	
Ação3	2		1	

1.3.1 Interpretação da Tabela de Decisão

A tabela acima é interpretada, da seguinte forma:

- Se $X > 0$, $Y \neq 10$ e $A = 1$, então serão executadas as ações denominadas Ação1 e Ação3, nesta ordem;
- Se $X \leq 0$, $Y = 10$ e $A = 2$, então será executada a ação denominada Ação2;
- Se $X > 0$, $Y = 10$ e $A > 3$, então serão executadas as ações denominadas Ação3, Ação1 e Ação2, nesta ordem;
- Se A tem valor negativo, então será executada a ação denominada Ação1.

A interpretação de uma tabela de decisão é feita por colunas. Simbolicamente, poderíamos entender como se em cada coluna na primeira linha colocássemos a palavra "if" e entre cada par de entradas de condição, o conectivo "and" e na linha horizontal dupla a palavra "then" [Po 74].

	<i>if</i>	<i>if</i>	<i>if</i>	<i>if</i>
C1	<i>and</i>	<i>and</i>	<i>and</i>	<i>and</i>
C2	<i>and</i>	<i>and</i>	<i>and</i>	<i>and</i>
C3	<i>then</i>	<i>then</i>	<i>then</i>	<i>then</i>

Segundo V. W. Setzer a interpretação mais adequada seria imaginar que a partir da segunda coluna em lugar de “if” colocaríamos “else if” no topo da primeira linha o que daria uma ordenação da esquerda para a direita. Schneider [Sc 85] propôs uma ordenação das colunas segundo um peso que é atribuído a cada uma delas pelo usuário. Uma ordenação deste tipo seria útil, visto que eliminaria problemas relacionados com a ambigüidade (vide 1.8), mas esta interpretação, que não é adotada neste trabalho, limitaria o uso de técnicas de tradução existentes.

Outra forma menos usual de se escrever uma tabela de decisão é colocar as condições e ações em colunas e as regras em linhas, mas neste trabalho adotaremos o modelo do exemplo anterior.

1.3.2 Condições e Ações

A tabela de decisão é formada por um conjunto de *linhas de condição*, e um conjunto não vazio de *linhas de ação*, separadas entre si, na figura, por uma linha horizontal dupla. As linhas de condição são separadas entre si por linhas horizontais simples, assim como as linhas de ação. Observa-se que não se faz restrição teórica sobre tabelas de decisão com conjunto vazio de linhas de condição, embora do ponto de vista prático isto possa parecer inútil.

Descrição das Condições	Entradas de Condição	Linhas de Condição
Descrição das Ações	Entradas de Ação	Linhas de Ação

Cada linha de condição é dividida em duas partes separadas por uma linha vertical dupla: A *descrição da condição* (“condition stub”) e a parte de *entradas de condição* (“condition entry”). As entradas de condição para cada linha de condição são separadas entre si por linhas verticais simples. Cada linha de ação é dividida em duas partes separadas por uma linha vertical dupla: A *descrição da ação* (“action stub”) e a parte das *entradas de ação* (“action entry”). As entradas de ação para cada linha de ação são separadas entre si por linhas verticais simples. A *i*-ésima entrada de condição de todas as linhas de condição formam uma *coluna de entradas de condição*. Analogamente, a *i*-ésima entrada de ação de todas as linhas de ação formam uma

coluna de entradas de ação. A cada coluna de entradas de condição corresponde uma única coluna de entradas de ação e vice-versa. Todas as linhas de condição devem ter uma entrada de condição em cada coluna. Diferentemente, as linhas de ação podem ter ou não entradas de ação, embora uma tal linha sem entradas seja totalmente inútil do ponto de vista prático, pois seria uma ação presente na tabela que nunca seria executada.

Na prática, não tem sentido duas linhas diferentes de condição com a mesma descrição da condição. Assim, de um modo geral, a descrição da condição determina uma única linha de condição. Em vista disso, denotaremos uma linha de condição pela sua descrição. Portanto, uma linha de condição, que tenha uma descrição C_i , será chamada de linha de condição C_i . Chamaremos também de condição e ação às partes de descrição de condição e ação, respectivamente.

Doravante, sempre que para um propósito específico, a parte das linhas de ação de uma tabela de decisão não for importante, será mostrada nas figuras apenas a parte das linhas de condição da tabela, sem que esta ressalva seja feita. Deste modo, mesmo figurando apenas parte de uma tabela, ela será referida sempre como uma tabela de decisão.

1.3.3 Regras de Decisão

Uma regra de decisão é uma sentença da forma:

“se C então A ”

onde C é uma expressão booleana qualquer e A é o nome de uma ação ou de um conjunto delas. A interpretação da regra acima é que “ A ” deve ser executada se “ C ” é verdadeira.

Cada *coluna* da tabela de decisão é formada por uma coluna de entradas de condição e da correspondente coluna de entradas de ação. Pode-se ver claramente que a interpretação associada a cada coluna é uma regra de decisão. Assim, uma tabela de decisão especifica um conjunto não vazio de regras de decisão. Normalmente, para facilidade de leitura, denominamos a regra determinada pela i -ésima coluna da tabela de decisão de regra R_i . Em algumas situações, nos desenhos de tabelas, colocaremos os nomes das regras de decisão numa linha inicial da tabela.

As entradas de ação numa regra dá a ordem em que a respectiva ação será executada na regra correspondente.

Alguns sistemas de tabelas de decisão impõem que toda regra deve especificar pelo menos uma ação. Mas é perfeitamente possível que numa determinada situação não se queira executar absolutamente nada. Isto caracteriza uma *regra vazia*.

	R_1	R_2	R_3
C_1	Y	Y	N
C_2	N	Y	N
A_1	1		
A_2			1

Na tabela acima, a regra R_2 é vazia. Em casos de tradutores que exigem uma ação em toda regra, basta definir uma “ação vazia” do tipo “NOP”.

Chamaremos uma regra de decisão simplesmente por *regra*.

Da forma como apresentamos, a tabela de decisão pode ser definida formalmente das formas abaixo, entre outras [Ca 74]:

- Como uma dupla de conjuntos de linhas de condição e de linhas de ação;
- Como um terno de conjuntos de condições, ações e regras de decisão;
- Como uma quádrupla de conjuntos de condições, ações, entradas de condição e entradas de ação [Ku-Ch 83].

Existem também definições funcionais para tabelas de decisão que podem ser encontrados em diversas fontes [Ca 74, Mi 85, Re-Sa-Ya 87, Mi 89].

1.3.4 Linhas de Condição Limitadas e Estendidas

As linhas de condição podem ser limitadas e estendidas, dependendo do conteúdo da descrição da condição. Se a descrição contém uma expressão lógica com valor “verdadeiro” (“Y”) ou “falso” (“N”), então a linha de condição se diz *limitada*. Caso contrário, a descrição da condição é uma expressão que deve ser complementada com o conteúdo das entradas de condição para se tornar uma expressão lógica e se diz linha de condição *estendida*.

Para simplicidade, chamaremos a cadeia contida na parte da descrição da condição de *condição*. Isto é um abuso, pois no caso de linhas estendidas, a cadeia não é uma expressão lógica. Analogamente, a cadeia contida na parte de descrição de ação será chamada simplesmente de *ação*.

O caso degenerado do conjunto de condições ser vazio tem a interpretação de que o conjunto de ações deve ser sempre executado, independentemente de qualquer teste sobre as variáveis do problema. Na verdade, uma tabela deste tipo, representa a situação em que se diria:

incondicionalmente, as ações devem ser executadas.

É claro que neste caso, temos somente uma regra estabelecendo a ordem de execução das ações, pois caso contrário, não teríamos como escolher qual delas executar. Tal tabela é dita *trivial*. Ela não tem utilidade prática e nem é permitida como entrada válida pelos processadores de tabelas de decisão, como aliás seria esperado, mas esta liberalidade pode ajudar na descrição de alguns algoritmos. Uma figura esquemática de uma tal tabela seria como a seguinte.

Ação1	1
Ação2	3
Ação3	2

Qualquer estrutura de comandos seqüenciais é, na verdade, uma tabela trivial, onde a ordem de execução é dada pela seqüência em que eles aparecem.

Num caso extremo de excepcionalidade, poderemos ter uma tabela de decisão trivial com uma regra vazia, que não tem utilidade prática e até deve ser impugnada como entrada para um processador de tabelas, mas ela tem utilidade na manipulação de tabelas por parte dos algoritmos.

1.3.5 Entradas de Condição e Ação

Denotaremos por C_{ij} a entrada de condição correspondente à regra R_j e condição C_i . Analogamente, A_{ij} , denota a entrada de ação correspondente à regra R_j e ação A_i .

As entradas de condição podem ser “Y”(Yes), “N”(No) ou “-” (Indiferente) para as linhas de condição limitada. Comportam ainda os símbolos “*” e “\$” que serão explicados mais adiante. Os símbolos “Y” e “N” numa regra indicam que a condição deve ser verdadeira ou falsa, respectivamente, para que tal regra seja selecionada.

O símbolo “-” na entrada de condição significa que, para a regra em questão, a condição é indiferente, podendo ser “Y” ou “N”. Isto indica que os algoritmos de tradução de tabelas podem ignorar estas entradas nos testes de seleção.

Para as linhas de condição estendidas, as entradas de condição são valores, que as expressões contidas na parte de descrição da linha de condição podem assumir.

Por exemplo, na figura abaixo, estão exemplos de linhas de condição estendida e limitada, respectivamente.

X	1	2	>10
---	---	---	-----

X = 10	Y	N	N	-
--------	---	---	---	---

Neste trabalho, as entradas de ação podem conter um número indicando a ordem de sua execução na regra. Se contiverem um símbolo vazio, ou “-”, ou zero, indicam que a ação não deve ser executada.

1.3.6 Linhas de Ação estendidas

Podemos estender a idéia de linhas de condição estendida para a parte de ações, de forma imediata, desde que a ordem de execução das ações seja irrelevante, ou se imponha um custo adicional de repetir a descrição de uma ação, sempre que for necessária, para a boa ordenação das ações.

As linhas de ação, que possuem na sua parte de descrição uma ação completa, são as encontradas normalmente em tabelas de decisão. Por analogia com as condições, são chamadas de *linhas de ação limitadas*. Aquelas linhas que, na parte de descrição, não descrevem totalmente a ação a ser executada, deixando para ser completada nas entradas de ação são chamadas de *linhas de ação estendidas*.

Nas tabelas que comportam estes dois tipos de linhas de ação, as entradas de ação conteriam somente um símbolo (por exemplo, “X”) assinalando que uma certa ação limitada deve ser executada ou conteria o complemento da descrição da ação estendida que deve ser executada.

Este tipo de tabela não é freqüente e este trabalho só tratará com linhas de ação limitadas, com as entradas de ação recebendo a ordem em que as ações devem ser executadas. Assim, não se fará menção a tipos de linhas de ação. Todas são limitadas. Deste modo, os termos *limitada* e *estendida* somente tem a ver com as linhas de condição.

A título de ilustração, para fechar este assunto, apresentamos exemplos.

A figura abaixo mostra como um chefe de vendas calcula a comissão a ser paga a cada um de seus vendedores. A ordem de execução das ações deve ser definida, por exemplo, de cima para baixo.

Tempo de casa	<2	<2	≥ 2	≥ 2
Venda > Média	N	Y	N	Y
Avaliação :=	Regular	Bom	Ruim	Bom
Comissão :=	$0.10 \times V$	$0.11 \times V$	$0.09 \times V$	$0.11 \times V$

Existe na literatura [Po-Hi-Ha 71] uma forma mais trabalhosa de se utilizar linhas de ação estendidas, sem perder o poder de especificar a ordem de execução, colocando-se o número de ordem entre parênteses, ao lado das entradas de ação. Segue uma tabela para exemplificar esta forma de se escrever uma tabela de decisão. A sua interpretação é facilmente deduzida da figura.

C_1	Y	N	Y	N
C_2	N	Y	Y	N
$X :=$	Z (1)	-Z (2)	W (1)	Y (2)
$Y :=$	-1 (2)	0 (1)	-X (2)	W (1)

1.3.7 Tabelas Limitadas, Estendidas e Mistas

Dizemos que uma tabela de decisão é *limitada* se todas as suas linhas de condição são limitadas. Ela é *estendida* se todas as suas linhas de condição são estendidas. Ela é *mista* se contém linhas de condição de ambos os tipos.

Na prática usam-se tabelas de decisão mistas, mas todas podem ser transformadas em tabelas limitadas, apenas juntando cada entrada de condição à respectiva descrição, formando uma linha de condição limitada. Este procedimento produzirá uma linha de condição limitada para cada entrada de condição distinta na linha de condição estendida. A figura seguinte mostra um exemplo de uma tabela mista.

X	=ALFA	=BETA	=GAMA	=DELTA	=ÉPSILON
Z = W	Y	Y	N	-	-
A1	1		1	2	
A2		1	2	1	
A3					1

Podemos ver na figura logo a seguir que a tabela mista acima foi transformada numa tabela limitada equivalente, no sentido de que as regras de decisão são as mesmas quanto à interpretação. Mais adiante (1.12), veremos como fazer esta transformação sem utilizar estes símbolos "N".

Esta transformação para a forma limitada é obtida com o desdobramento das linhas de condições estendidas. Cada condição estendida é desdobrada em p condições limitadas, onde p é o número de entradas de condição distintas na respectiva linha.

Observa-se que a formulação limitada da tabela de decisão torna-a mais vertical, enquanto a versão estendida torna-a mais horizontal. Na literatura diz-se que a primeira comprime a tabela horizontalmente, enquanto a segunda comprime-a verticalmente [Po 74].

X = ALFA	Y	N	N	N	N
X = BETA	N	Y	N	N	N
X = GAMA	N	N	Y	N	N
X = DELTA	N	N	N	Y	N
X = ÉPSILON	N	N	N	N	Y
Z = W	Y	Y	N	-	-
A1	1		1	2	
A2		1	2	1	
A3					1

Em razão da facilidade com que tabelas mistas ou estendidas são transformadas em tabelas limitadas, é conveniente tratar dos aspectos teóricos restringindo ao conjunto de tabelas de decisão limitadas. Na prática, esta conveniência, não é custosa, porque os mecanismos de transformação são simples e facilmente automatizáveis. Assim, as tabelas mistas são perfeitamente viáveis e aceitas pelos processadores em geral, mas durante o estudo do assunto adotaremos esta restrição. Doravante, o termo tabela de decisão denotará uma tabela de decisão limitada e os demais tipos serão explicitamente designados.

1.3.8 Condições Dependentes e Independentes

Dizemos que duas condições C_i e C_j , $i \neq j$, são *explicitamente dependentes* se elas operam sobre uma mesma variável. Ou seja, a descrição de ambas contém uma variável comum, como no exemplo da figura 1.1(a).

X = 1	Y	N
X = 2	N	Y

(a)

Sexo = Masculino	Y	Y	N
Tem Certificado de Resevista	Y	N	N

(b)

X ≥ 3	
X = 1	
X ≥ 10	

(c)

Figura 1.1: Dependência entre condições

Duas condições C_i e C_j , $i \neq j$, são *implicitamente dependentes* se, embora não operem sobre uma variável comum, a semântica do problema particular as tornam inter-relacionadas, como no exemplo da figura 1.1(b), onde o fato de “Sexo = Masculino” ser falso, implica automaticamente

que a pessoa não possui Certificado de reservista, pela natureza extremamente particular do problema. Esta implicação torna as duas condições implicitamente dependentes.

Duas condições são *dependentes* [Po-Hi-Ha 71] se são explicitamente dependentes ou implicitamente dependentes. Caso contrário elas são *independentes*.

Para condições que são constituídas de apenas dois operandos, separados por um operador relacional, entre duas condições explicitamente dependentes, definem-se *dependência com exclusão mútua* e *dependência com sobreposição*. O primeiro caso ocorre quando existe um valor para a variável comum que produz a satisfação de ambas. O segundo caso ocorre quando não existe um tal valor. No exemplo da figura 1.1(c), as condições são explicitamente dependentes duas a duas. Entre a segunda e as demais condições existe a dependência com exclusão mútua e entre a primeira e a terceira condições, existe a dependência com sobreposição.

1.3.9 Entradas de Condição Implícitas

Os símbolos “*” e “\$” [Po 74,Po-Hi-Ha 71] nas entradas de condição são casos especiais de “N” e “Y”, respectivamente. Indicam que a condição correspondente é irrelevante para a regra que está sendo testada, mas que se o teste for efetuado, têm o significado idêntico a “N” e “Y”, respectivamente. Estas entradas são utilizadas para condições dependentes que são implicitamente falsas ou implicitamente verdadeiras, em consequência da avaliação de outras condições. Elas indicam o fato de que a satisfação de uma outra condição que tem entrada “Y” ou “N” para a regra provoca obrigatoriamente um determinado valor para a condição. Se provoca obrigatoriamente “N”, o fato é indicado com “*”. Se provoca “Y”, o fato é indicado com “\$”.

X = 5	Y	N
X = 2	N	Y

X = 5	Y	*
X = 2	*	Y

Considere as tabelas de decisão acima, que são apresentadas sem a parte das linhas de ação. Podemos observar que se $X = 5$ então obrigatoriamente a segunda condição das tabelas será falsa. O mesmo ocorre se a segunda condição for verdadeira, pois neste caso, a primeira será obrigatoriamente falsa, independentemente de qualquer teste. As entradas de condição da tabela à direita descrevem estes fatos. A colocação do símbolo “-” em lugar de “*” nesta tabela, apoiando-se no fato de que a condição é irrelevante para a regra, não está rigorosamente correta, pois o primeiro símbolo indica não só a irrelevância da condição como também que ela pode ser verdadeira ou falsa, o que não é o caso. A despeito desta observação, em muitos casos, esta troca não trará prejuízos maiores, a menos que se introduza ambigüidades (vide 1.8) na tabela de decisão.

Observemos que o símbolo “*” dispensa o teste da condição $X = 2$ na primeira regra da tabela à direita, mas ele tem o sentido de “N”, dizendo que se $X = 5$ então X obrigatoriamente não pode valer 2. Os símbolos “*” e “\$” são chamados de “N” implícito e “Y” implícito, respectivamente.

Da mesma maneira, “-” indica que a avaliação da condição é dispensada, mas enquanto “-” corresponde a (“Y”+“N”), “\$” corresponde a “Y” e “*” corresponde a “N”. Assim, os símbolos “-”, “*” e “\$” indicam dispensa de teste, mas têm significados implícitos diferentes. Se a despeito

da dispensa de avaliação, a comparação é feita por algum algoritmo de conversão, então “*” tem toda a função de “N”, “\$” toda a função de “Y” e “-” toda a função de (“Y”+“N”). Em razão dessa dispensa de teste da condição para a seleção da regra, dizemos que a condição é *irrelevante* para a regra, isto é “\$”, “*” e “-” são entradas irrelevantes e “Y” e “N” são entradas relevantes.

Faltou à Prova	Y	*	-	N	N
NotaProva < 5	\$	N	Y	Y	N
NotaTrabalho < 5	Y	Y	N	Y	N
Media := NotaProva		1			
Media := NotaTrabalho			1		
Reprovado				1	
CalculaMedia					1
Faz Substitutiva	1				

Nessa tabela, a dependência implícita está no fato de que, no contexto do regulamento da escola e do que fora acertado no início do semestre letivo, se o aluno faltou à prova, a sua nota da prova é zero, o que torna a segunda condição da primeira regra obrigatoriamente verdadeira. Este fato é assinalado no exemplo, pelo símbolo “\$” na respectiva entrada de condição. Analogamente, na segunda regra da tabela anterior, se a nota da Prova não é menor que 5, logo obrigatoriamente, o aluno não faltou à prova. Isto é assinalado pela entrada “*”.

Existe uma natural dificuldade no tratamento automático deste fenômeno e praticamente fica impossível, se a dependência é implícita. Neste sentido, quando se trata com tabelas de decisão com condições dependentes é preciso maiores cuidados e não esperar que aspectos semânticos do problema possam ser deduzidos a partir de simples análise sintática da tabela [Ib-Ra 78,Bi-Ra 87].

1.4 Desdobramento de “-”

O símbolo “-” (“don’t care”) na entrada de condição de uma regra significa que a condição é indiferente para ela. Chamá-lo-emos de “*Tanto-Faz*” ou “*Indiferente*”. A interpretação para o símbolo é que a condição tanto pode ser verdadeira como falsa para a satisfação dessa regra. Por outro lado, corresponde a existir na tabela duas regras exatamente iguais, a menos desta entrada de condição: Numa estaria com “Y” e noutra com “N”. Assim, uma regra que contém uma entrada indiferente pode ser desdobrada em duas regras, onde se copia todas as demais entradas distintas de “-” e na entrada que corresponderia ao símbolo “-” se coloca “Y” numa regra e “N” em outra. Ou seja, “-” = “Y”+“N”. Esta operação será chamada de *desdobramento de “-”* e dizemos que a regra original foi *desdobrada* nas duas regras recém-criadas.

Def.: Uma regra é dita *simples* se ela não possui entradas de condição “-”. Caso contrário, a regra é dita *complexa*.

Teorema 1 Uma regra R com p símbolos “-” pode ser desdobrada em 2^p regras simples.

Dem.: Por indução em p . Para a base da Indução, toma-se $p = 0$. Claramente, a regra R equivale a uma única regra simples que é ela própria. $2^0 = 1$.

Suponha que a tese seja válida para regras com menos que p entradas de condição “-”, $p > 0$ e que a regra R tem p dessas entradas. Seja C_i uma condição para a qual a regra R tem uma entrada “-”. Aplicando o desdobramento deste símbolo, obtemos regras R_1 e R_2 , cada uma com $p - 1$ entradas de condição “-”. Logo, aplicando a hipótese indutiva em R_1 e R_2 , obtemos

$$2^{p-1} + 2^{p-1} = 2^p \text{ regras simples. } \square$$

Assim, dizemos que a regra R tem o equivalente a 2^p regras simples ou que a regra R representa 2^p regras simples.

	R_1	R_2	R_3
NotaProva ≥ 5	Y	N	Y
NotaTrabalho ≥ 5	Y	-	N
Aprovado	1		
Refaz Trabalho			1
Reprovado		1	

(a)

	R_1	R'_2	R''_2	R_3
NotaProva ≥ 5	Y	N	N	Y
NotaTrabalho ≥ 5	Y	Y	N	N
Aprovado	1			
Refaz Trabalho				1
Reprovado		1	1	

(b)

Figura 1.2: Desdobramento de regras

Na figura 1.2, as regras R'_2 e R''_2 da tabela (b) correspondem ao desdobramento da regra R_2 da tabela (a).

Para uma tabela de decisão T com regras $R_1, \dots, R_n$, com respectivamente $p_1, \dots, p_n$ entradas de condição “-”, seja $n(T)$ o número equivalente de regras simples de T , ou seja, T representa $n(T)$ regras simples. Assim, $n(T)$ é a soma de regras simples representadas por cada regra de T . Facilmente, se obtém:

$$n(T) = 2^{p_1} + 2^{p_2} + \dots + 2^{p_n}.$$

Na verdade, $n(T)$ é o número de regras da tabela T' que é obtida através de desdobramentos consecutivos dos símbolos “-” da tabela T . Se a tabela T só tem regras simples, $T = T'$.

1.5 Combinação ou Intercalação de Regras

Assim como uma entrada de condição “-” numa regra pode ser desdobrada em duas regras, o inverso pode ser feito: duas regras que sejam exatamente iguais, a menos de uma só entrada de condição, em que uma possui “Y” e a outra “N”, podem ser juntadas numa única regra, substituindo-se a tal entrada de condição por “-”. Esta operação é chamada de *Intercalação de regras* (“rule merging”) [Me-Ba 77] ou *Combinação de regras* [Mc 68, Po-Hi-Ha 71, Sh 75]. A repetição sucessiva desta operação sobre uma tabela pode diminuir bastante o tamanho de

algumas tabelas [Ch 67], mas nem sempre se consegue que as regras, com a mesma sequência de ações, sejam combinadas numa única regra. Além disso, as tabelas resultantes são mais compreensíveis, pois mostram com clareza, as situações em que certas condições são irrelevantes.

	R ₁	R ₂	R ₃	R ₄
É tarde da noite	Y	Y	N	N
Está chovendo	Y	N	Y	N
Dormir	1	1		
Ir ao Cinema				1
Ver Televisão			1	

(a)

	R	R ₃	R ₄
É tarde da noite	Y	N	N
Está chovendo	-	Y	N
Dormir	1		
Ir ao Cinema			1
Ver Televisão		1	

(b)

Figura 1.3: Intercalação de regras

Na figura 1.3, a regra R da tabela (b) corresponde à intercalação das regras R₁ e R₂ da tabela (a).

Como esta operação nem sempre é suficiente para reduzir o tamanho de grandes tabelas, existem outras propostas que tentam apresentar métodos sistemáticos para construção de tabelas menores com encadeamento [Ch 67, St 73, Sh 75].

1.6 Função-“And”

Como já vimos, a cada coluna está associada uma interpretação que determina uma regra de decisão da forma

if C then A.

Nesta regra, *C* representa várias condições separadas pelo conectivo “and”. Informalmente, uma *função-And* é a parte “C” de uma regra, ou seja, assim como a regra corresponde à interpretação de uma coluna da tabela de decisão, a função-and corresponde à interpretação de uma coluna de entradas de condição.

Seja uma tabela de decisão com condições $C_1, C_2, \dots, C_m$ e regras $R_1, R_2, \dots, R_n$. Define-se uma *função-And* B_j [Po-Hi-Ha 71, Po 74] como sendo a sequência:

$$B_j = (C_{1j}, C_{2j}, \dots, C_{mj}), \quad 1 \leq j \leq n,$$

onde C_{ij} comporta “Y”, “N”, “-”, “*” ou “\$”.

	R ₁	R ₂	R ₃
Condição1	Y	\$	N
Condição2	Y	–	N
Condição3	*	N	–
Condição4	–	Y	Y
Condição5	\$	*	–
Condição6	Y	N	Y

Para a tabela mostrada, teríamos as seguintes funções-and's:

$$\begin{aligned} B_1 &= (Y, Y, *, -, \$, Y) \\ B_2 &= (\$, -, N, Y, *, N) \\ B_3 &= (N, N, -, Y, -, Y). \end{aligned}$$

Cada função-and pode receber valor falso ou verdadeiro de acordo com a avaliação das condições $C_1, \dots, C_m$. Assim, o valor de B_j , denotado $V(B_j)$ vale:

$$\begin{aligned} V(B_j) &= 1, \text{ se todas as especificações das entradas de} \\ &\quad \text{condição são satisfeitas na Regra } R_j; \\ V(B_j) &= 0, \text{ caso contrário.} \end{aligned}$$

Formalmente, o valor de B_j , $1 \leq j \leq n$ é calculado da seguinte forma: Seja $S = (a_1, a_2, \dots, a_m)$ o vetor de estados do conjunto de condições $C_1, \dots, C_m$, onde, para $1 \leq i \leq m$,

$$a_i = \begin{cases} 0 & \text{se } C_i \text{ é falsa} \\ 1 & \text{se } C_i \text{ é verdadeira} \end{cases}$$

O vetor S é a valoração das condições num determinado instante. Assim,

$$V(B_j) = (V(C_{1j}) \cdot V(C_{2j}) \cdot \dots \cdot V(C_{mj})),$$

onde $V(C_{kj})$ é dado pela seguinte tabela:

C_{kj}	a_k	$V(C_{kj})$
Y,\$	1	1
Y,\$	0	0
N,*	1	0
N,*	0	1
–	0,1	1

Portanto,

$$V(B_j) = \begin{cases} 1 & \text{se } V(C_{kj}) = 1, \quad 1 \leq k \leq m \\ 0 & \text{caso contrário} \end{cases}$$

Por exemplo, a regra

$$B_j = (Y, Y, N, *, \$, -, Y),$$

com as condições

$$\begin{array}{lll} C_1 = true & C_2 = true & C_3 = false \\ C_4 = false & C_5 = true & C_6 = true \end{array}$$

produzem um vetor de estados $S = (1, 1, 0, 0, 1, 1)$. Usando a definição, obtemos:

$$\begin{array}{lll} V(C_1) = 1 & V(C_2) = 1 & V(C_3) = 1 \\ V(C_4) = 1 & V(C_5) = 1 & V(C_6) = 1. \end{array}$$

Portanto, o valor da função-and $B_j, V(B_j) = 1$.

Def.: Dizemos que R_j é *selecionável* numa tabela de decisão contra um vetor de estados S se $V(B_j) = 1$. Caso contrário a regra é *não selecionável*.

Uma função-and é *pura* se não contém “-”. Ela é *mista* se contém um ou mais “-”. Claramente, uma função-and pura implica numa regra simples e função-and mista numa regra complexa [Po-Hi-Ha 71].

Como não sabemos qual o conjunto $Q \subseteq \{0, 1\}^m$ que contém os estados semanticamente possíveis para o vetor de estado das condições, vamos considerar que todas as combinações previstas em cada uma das funções-and são possíveis. São definitivamente impossíveis somente aquelas que estão explicitamente proibidas por entradas de condição implícitas. As combinações não previstas na tabela e nem proibidas são teoricamente possíveis e sua interpretação fica a critério do que será visto na seção 1.17. É claro que não existe sentido lógico na presença de duas regras numa mesma tabela, onde uma delas proíba explicitamente a ocorrência da combinação prevista pela outra.

1.7 Regras e Colunas

Vimos que uma regra é a interpretação associada a uma coluna da tabela de decisão. Esta distinção é sutil e muitas vezes sobrecarrega a notação. Feita esta observação, usaremos a palavra *regra* tanto para representar a interpretação “*if...then...*” associada a uma coluna, como também para a própria coluna da tabela de decisão. Assim, diremos que uma regra $R_j, 1 \leq j \leq n$ é uma sequência

$$(C_{1j}, C_{2j}, \dots, C_{mj}, A_{1j}, A_{2j}, \dots, A_{lj}),$$

para uma tabela de decisão com m condições, l linhas de ação e n regras. Cada termo da sequência C_{ij} comporta um dos símbolos “Y”, “N”, “*”, “\$”, “-” e A_{ij} é um número indicando a ordem de execução da ação descrita na i -ésima linha de ação ou uma entrada nula (“0”, “-” ou “vazio”), indicando que tal ação não é executada na regra em questão.

1.8 Regras Ambíguas

Duas funções-and B_i e B_j são *dependentes* [Po 74] se existe uma avaliação para as condições $C_1, C_2, \dots, C_m$, tal que $V(B_i) = 1$ e $V(B_j) = 1$. Caso contrário são *independentes*. Em outras palavras, duas funções-and são dependentes se existe uma situação sobre o conjunto de condições que permite a ambas as funções-and serem selecionáveis, ou seja, existe uma combinação de valores para o vetor de estados que seleciona as duas regras.

Duas regras se dizem *ambíguas* se as funções-and associadas são dependentes. Caso contrário são *não ambíguas*.

Duas regra ambíguas são *contraditórias* ou *inconsistentes* se as colunas de entradas de ação são distintas. Caso contrário, elas são *redundantes*.

Dizemos que uma tabela de decisão é *ambígua* se contém regras ambíguas. Caso contrário, ela é *não ambígua*. Logo uma tabela de decisão trivial é não ambígua se possui uma única regra.

Parece que não existe muito sentido lógico, do ponto de vista do mundo real, que duas situações distintas possam ser contempladas ao mesmo tempo. Assim, geralmente, se impõe que as tabelas não sejam ambíguas, considerando-se como impropriedade na sua construção se elas ocorrerem. Assim, quando nos referirmos a tabelas de decisão, estaremos tratando com tabelas não ambíguas, a menos que explicitamente designadas.

Teorema 2 *Numa tabela de decisão limitada, duas funções-and são independentes se e somente se existe uma condição para a qual uma das funções tem entrada "Y" ou "\$" e a outra "N" ou "*".*

Dem.: Sejam $B_i = (C_{1i} \cdot C_{2i} \cdot \dots \cdot C_{mi})$ e $B_j = (C_{1j} \cdot C_{2j} \cdot \dots \cdot C_{mj})$ duas funções-and quaisquer.
 $\Leftarrow$. Seja k um índice tal que a linha de condição C_k tem entrada "Y" ou "\$" numa regra e "N" ou "*" na outra. A hipótese garante que existe tal k . Sem perda de generalidade, suponha $C_{ki} = \text{"Y"}$ ou "\$" e $C_{kj} = \text{"N"}$ ou "*". Seja a_k o valor da condição C_k .

caso a) $a_k = 1$. Logo, $V(C_{kj}) = 0$ e conseqüentemente $V(B_j) = 0$.

caso b) $a_k = 0$. Logo, $V(C_{ki}) = 0$ e conseqüentemente $V(B_i) = 0$.

$\Rightarrow$. Suponha que não existe índice k tal que $C_{ki} = \text{"Y"}$ ou "\$" e $C_{kj} = \text{"N"}$ ou "*" ou vice-versa. Vamos mostrar que as funções-and B_i e B_j são dependentes. Seja $S = (a_1, a_2, \dots, a_m)$ o vetor de estado do conjunto de condições, construído da seguinte maneira:

$$a_k = \begin{cases} 0 & \text{se } C_{ki} = \text{"N" ou "*" } \\ 0 & \text{se } C_{ki} = \text{"-"} \text{ e } C_{kj} = \text{"N" ou "*" } \\ 1 & \text{se } C_{ki} = \text{"Y" ou "$" } \\ 1 & \text{se } C_{ki} = \text{"-"} \text{ e } C_{kj} = \text{"Y" ou "$" } \\ 1 & \text{se } C_{ki} = \text{"-"} \text{ e } C_{kj} = \text{"-"} \end{cases}$$

S é um estado possível, pois está previsto na função-and B_i . Por construção, $V(C_{ki}) = 1$ para todo $1 \leq k \leq m$. Logo, $V(B_i) = 1$.

Por outro lado, se

- $C_{kj} = \text{"Y"}$ ou "\$", por hipótese, $C_{ki} = \text{"Y"}$, "\$" ou "-". Logo, por construção de a_k , $V(C_{kj}) = 1$.

- $C_{kj} = \text{"N"} \text{ ou } \text{"*"} \text{, por hipótese, } C_{ki} = \text{"N"}, \text{"*"} \text{ ou } \text{"-"} \text{. Logo, por construção de } a_k, V(C_{kj}) = 1.$
- $C_{kj} = \text{"-"} \text{. Logo, por definição, } V(C_{kj}) = 1.$

Portanto, $V(B_j) = 1$ e isto mostra que B_i e B_j são dependentes. Este resultado estabelece a tese. $\square$

Corolário 3 *Numa tabela de decisão limitada, duas regras são não ambíguas se e somente se existe uma condição para a qual uma das regras tem entrada "Y" ou "\$" e a outra "N" ou "*".*

A demonstração segue do teorema 2 e da definição. $\square$

Corolário 4 *Uma tabela de decisão limitada é não ambígua se e somente se para cada par de regras existe uma condição para a qual uma das regras tem entrada "Y" ou "\$" e a outra "N" ou "*".*

A demonstração segue do corolário 3 e da definição. $\square$

Em consequência do corolário 4, a verificação de ambigüidades numa tabela de decisão é uma tarefa extremamente simples. Basta tomar cada par de regras e comparar as entradas de condição correspondentes e verificar a suficiência que está no enunciado daquele corolário. Para uma tabela com n regras, existem $(n^2 - n)/2$ pares de regras a serem comparadas. Para cada par de regras é necessário comparar, no máximo, as entradas das m condições da tabela. Logo, a tarefa de verificação de ambigüidades pode ser feita com $m \cdot (n^2 - n)/2$ operações de comparação. Nesta discussão de complexidade, precisamos analisar o que é a entrada do problema. A tabela de decisão contém m condições e n regras. Como a verificação de ambigüidades não opera sobre a parte de ações, vamos considerar como entrada do problema somente a parte de linhas de condição. Ou seja, o tamanho da entrada l é $m \cdot n$. Levando em conta que

$$m \cdot n^2 \leq l^2,$$

podemos colocar o número de comparações obtido em função do comprimento da entrada e concluímos que a complexidade desta verificação é $O(l^2)$.

A tabela da figura 1.4(a) é ambígua, pois as condições C_1 e C_2 , sendo ambas verdadeiras, fazem com que as duas regras sejam selecionáveis. A tabela da figura 1.4(b) também é ambígua, pois as regras:

- R_1 e R_4 são redundantes;
- R_4 e R_6 são redundantes;
- R_2 e R_5 são redundantes;
- R_2 e R_7 são inconsistentes e
- R_5 e R_7 são inconsistentes.

	R ₁	R ₂
C ₁	Y	–
C ₂	–	Y

(a)

	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆	R ₇
Salário >1000	Y	N	Y	Y	N	Y	N
TemDependentes	Y	–	N	Y	N	Y	N
Idade >45	N	–	N	–	–	Y	N
IR = 0.2·Salário			1				
IR = 0.1·Salário	1			1		1	1
IR = 0		1			1		

(b)

Figura 1.4: Tabelas ambíguas

1.9 Ambigüidade Aparente

Retomando a tabela da figura 1.4(a) podemos concluir que ela é estruturalmente ambígua, já que a instância do problema em que ambas as condições são verdadeiras produz o vetor de estado $S = (1, 1)$. Nesta situação, calculando os valores das funções-and B_1 e B_2 associadas às regras R_1 e R_2 , obtemos $V(B_1) = 1$ e $V(B_2) = 1$.

Agora, vamos manter a mesma estrutura da tabela, mas substituir C_1 por $X = 4$ e C_2 por $X = 5$ e obtemos a seguinte tabela.

	R ₁	R ₂
X = 4	Y	-
X = 5	-	Y

É claro que o vetor de estados $S = (1, 1)$ é impossível de ocorrer. Ou seja, a ambigüidade é apenas aparente, pois a natureza do problema elimina esta possibilidade. Analisando melhor o problema e o significado do símbolo “-”, podemos observar que ele não está bem utilizado, pois ele indica que para a regra em questão, a condição tanto pode ser verdadeira como falsa, mas isto é impedido pela semântica do problema. Esta ambigüidade é eliminada substituindo-se “-” por “*”. Além disso, este procedimento faz com que a tabela expresse com precisão a semântica do problema. Assim, em lugar de ambigüidade aparente como chamada, ficaria melhor assinalar ocorrências deste tipo como uma impropriedade sutil na construção da tabela de decisão. Esta dificuldade aparece em função da dependência entre condições que foi descrita em 1.3.8. Em geral, não é fácil detectar automaticamente se uma ambigüidade é apenas aparente [Mu-Ra 70], ficando esta tarefa ainda mais complicada se a dependência entre condições é implícita.

1.10 Eliminação de Ambigüidades

O problema da ambigüidade em tabelas de decisão foi discutido por diversos autores [Ki 68, Ki-Jo 73, Ib-Ra 78, Sc 85]. Considere o problema seguinte: Uma certa ação A deve ser executada, quando $X > 0$ ou $Y > 0$. Parece que a tabela que segue é uma representação natural deste problema.

	R_1	R_2
$X > 0$	Y	–
$Y > 0$	–	Y
A	1	1

O problema que surge é que a tabela apresenta uma ambigüidade em sua estrutura. É perfeitamente possível que as duas regras sejam selecionáveis simultaneamente.

Como fica a afirmação de que, nos problemas do mundo real, parece não apropriado que ambigüidades ocorram. Analisando o enunciado, observamos que o problema é apresentado como uma única regra:

se $X > 0$ ou $Y > 0$ então execute A .

O problema decorre do fato de que estamos tentando representar uma só regra original através de duas regras. A idéia de que as regras de uma tabela correspondem a situações distintas não é mais válida. Além disso, como uma regra implementa as funções-and, não conseguimos representar a regra original com o conectivo “ou” numa só regra com duas linhas de condição.

Uma possível abordagem para o problema seria abrir a possibilidade de redundâncias nas tabelas, apontando irregularidades somente nos casos de inconsistências o que é adotado por King [Ki 68]. Um outro recurso é continuar não permitindo ambigüidades e tentar corrigir a tabela.

Existem sistemas de tabelas de decisão que impõem uma sintaxe particular para as condições, como, por exemplo, que devem constituir-se de apenas uma variável seguida de um operador relacional e de uma constante. Nos casos em que as condições podem ser quaisquer expressões lógicas, poderíamos escrever numa única condição como no enunciado do problema, obtendo a tabela seguinte:

	R_1
$X > 0$ ou $Y > 0$	Y
A	1

Outra possibilidade é substituir um dos símbolos “–” por “N”, como está na figura 1.5(a). Uma outra opção é substituir o símbolo “–” por “*”, como está reescrito na tabela da figura 1.5(b).

	R ₁	R ₂
X > 0	Y	N
Y > 0	-	Y
A	1	1

(a)

	R ₁	R ₂
X > 0	Y	*
Y > 0	-	Y
A	1	1

(b)

	R ₁	R ₂
X ≤ 0	N	-
Y ≤ 0	-	N
A	1	1

(c)

Figura 1.5: Eliminação de ambigüidade aparente

Devido a esta forma de utilização, o símbolo “*” é encontrado na literatura com o nome de *eliminador de ambigüidades*, porque a sua utilização também como entrada implícita acaba normalmente eliminando uma ambigüidade estrutural que semanticamente não existe [Po-Hi-Ha 71]. Se a tabela original fosse aquela mostrada na figura 1.5(c), o eliminador de ambigüidades seria o símbolo “\$”.

Quando estes símbolos são usados com esta finalidade, com respeito às regras R_j e R_k , devem ser tomados três cuidados.

- A ambigüidade entre as regras R_j e R_k é semanticamente válida.
- O eliminador de ambigüidade deve substituir o símbolo “-”.
- Suponha que será substituído na i -ésima linha de condição. Logo, $C_{ij} = \text{“-”}$ ou $C_{ik} = \text{“-”}$. Sem perda de generalidade, suponha que $C_{ij} = \text{“-”}$. Se C_{ij} será substituído por “*” (“\$”), então necessariamente $C_{ij} = \text{“Y”}(\text{“N”})$.

Estas restrições não são impeditivas à finalidade; elas apenas evitam problemas na sua utilização. A primeira é claramente óbvia. Quanto à segunda restrição, o uso de “*” para substituir “Y” ou “\$”, provocaria uma alteração da semântica do problema. Além disso, a substituição de “N” por “*” não eliminaria nenhuma ambigüidade. Raciocínio análogo vale para “\$”.

A terceira restrição proíbe liberalidades sem sentido lógico para uma tabela de decisão, que seria inserir uma linha de condição nova, somente com entradas de condição implícitas para eliminação de ambigüidades.

	R ₁	R ₂
C ₁	Y	Y
A	1	1

	R ₁	R ₂
C ₁	Y	Y
C ₂	*	\$
A	1	1

Na figura acima, se não fosse a terceira restrição, a primeira tabela, claramente ambígua, poderia ser substituída pela segunda, que não é ambígua, pela simples colocação de uma condição C_2 arbitrária qualquer que contém somente entradas irrelevantes. Na verdade, estes tipos de

linhas de condição não deveriam aparecer na especificação dos problemas por tabelas de decisão. Já que toda a linha só tem entradas irrelevantes, então ela não deve ser importante para a eliminação de ambigüidades. Os pré-processadores podem fazer uma verificação sobre as tabelas de decisão de modo a descartar estas linhas de condição. Aliás, é o que eles fazem quando linhas deste tipo são produzidas no processamento de tabelas.

1.11 Ambigüidade em Tabelas Mistas

Vamos analisar a ambigüidade em tabelas mistas e estendidas. Considere a seguinte tabela de decisão:

	R ₁	R ₂	R ₃
X	>5	=7	>2
A			

Como duas regras são ambíguas se existe uma instância do problema que faz com que ambas sejam satisfeitas, então as regras da tabela acima são ambíguas duas a duas. Para impedir que isto ocorra, vamos impor uma restrição [Po-Hi-Ha 71, Ki-Jo 75] sobre as linhas de condição estendidas, que chamaremos de *restrição de exclusão*.

Seja C_k uma linha de condição estendida com entradas $C_{k1}, C_{k2}, \dots, C_{kn}$. Cada uma dessas entradas de condição ou é um símbolo irrelevante ou é uma expressão que define um conjunto de valores, eventualmente unitário e, portanto, relevante para a regra. No que segue S_{kj} denota o conjunto de valores definido pela expressão contida na entrada de condição C_{kj} se ela é relevante. A restrição de exclusão é definida da seguinte maneira:

Cada C_{ki} relevante, $1 \leq i \leq n$, é $\left\{ \begin{array}{l} \text{exatamente igual a } C_{kj} \text{ (} C_{ki} = C_{kj} \text{) ou} \\ \text{mutuamente exclusivo com } C_{kj} \text{ (} S_{ki} \cap S_{kj} = \emptyset \text{),} \\ 1 \leq j \leq n. \end{array} \right.$

Esta restrição garante que a linha de condição estendida C_k não produzirá regras semanticamente ambíguas. Na verdade, uma linha de condição limitada é um caso particular de linha de condição estendida em que a descrição da condição é completada com um dos dois valores possíveis "Y" ou "N". Além disso, uma linha de condição limitada está de conformidade com a restrição de exclusão, visto que as entradas possíveis são "Y", "N" e "-", com os dois primeiros mutuamente exclusivos. As entradas de condição implícitas "\$" e "*" são formas particulares de "Y" e "N".

Podemos estender a notação das entradas implícitas para as linhas de condição estendidas, mas com certo cuidado na sua interpretação. Muitos processadores de tabelas de decisão exigem que a parte de descrição da linha de condição estendida contenha uma única variável e chamam-na de *variável de condição*. Na descrição abaixo, vamos manter este nome, mas subentende-se que em lugar de uma única variável podemos ter uma expressão.

- "*": As outras entradas de condição relevantes na regra determinam obrigatoriamente que o valor da variável de condição não pertence a nenhum dos conjuntos de valores presentes na linha ("N" implícito).

- “\$”: As outras entradas de condição relevantes na regra determinam obrigatoriamente que a variável de condição pertence a todos os conjuntos de valores presentes na linha (“Y” implícito). É preciso observar que esta extensão nem sempre é possível. Aliás, somente é válida se na linha de condição existe uma única entrada de condição relevante, pois sendo a linha estendida, os conjuntos são disjuntos pela restrição de exclusão e assim, uma variável não poderia pertencer a mais que um.

	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆
X	=5	=6	=7	=8	*	=9
Z=0	Y	Y	Y	Y	N	Y

(a)

	R ₁	R ₂	R ₃
X	=5	–	\$
Y=0	Y	N	Y
Z=1	Y	Y	N

(b)

Na tabela (a), o símbolo “*” da regra R₅ indica que a condição ($Z = 0$) sendo falsa, automaticamente determina que $X \notin \{5, 6, 7, 8, 9\}$. Na tabela (b) do exemplo, o símbolo “\$” da regra R₃ indica que as entradas das outras condições nesta mesma regra determinam implicitamente que $X = 5$.

A restrição de exclusão, embora seja um motivo de preocupação, não é muito penosa, pois pode ser contornada pelo usuário, particionando os conjuntos numa família de subconjuntos disjuntos. De um modo geral, esta restrição obriga a uma maior precisão na descrição do problema. Além disso, é possível que uma tabela que não atenda à restrição de exclusão contenha erros de especificação.

Vamos analisar as tabelas de decisão seguintes:

X	≥ 2	≥ 5	≥ 9

X	≥ 2 e < 5	≥ 5 e < 9	≥ 9

É possível que a pessoa quisesse, com a tabela da esquerda, a interpretação da tabela à direita [Po-Hi-Ha 71]. Os processadores de tabelas de decisão de um modo geral não admitem entradas de condição complexa, como um intervalo que está na tabela da direita, mas do ponto de vista teórico não existe dificuldade em aceitá-la. Mais adiante, exporemos uma forma de representar conjuntos complexos em entradas de condição. Por enquanto, fiquemos com a idéia de que é possível, sem nos preocuparmos com a sintaxe.

A despeito de uma possível intenção do usuário na tabela do lado esquerdo acima, ela fere a restrição de exclusão. Uma maneira semanticamente correta de representar as condições seria:

X	≥ 2	≥ 5	≥ 9
X	< 5	< 9	–

Do ponto de vista de implementação é difícil passar junto com a tabela informações adicionais que reflitam a semântica particular do problema. Por exemplo, neste caso, precisaríamos que a interpretação da tabela fosse feita considerando as duas condições como tendo uma forma especial de dependência, o que seria dificultado se estas linhas não fossem consecutivas na tabela e se as variáveis de condição e as entradas envolvessem expressões com valores desconhecidos em tempo de compilação.

1.12 Conversão de Tabelas Mistas em Limitadas

Na seção 1.3.7 mostramos uma tabela mista e a correspondente tabela transformada em limitada. O processo, como foi dito, é bastante simples. Como as entradas das linhas estendidas estão de acordo com a restrição de exclusão, podemos substituir os símbolos "N", introduzidos na tabela final, por "*".

Cada linha de condição estendida da tabela mista com p entradas relevantes distintas é transformada em p linhas limitadas.

Vamos considerar uma tabela de decisão mista, com linhas de condição $C_1, \dots, C_m$ e regras $R_1, \dots, R_n$. Sejam C_i uma linha de condição estendida e $\mathcal{D} = \{D_1, \dots, D_p\}$ o conjunto das entradas de condição relevantes da linha C_i . É claro que $\forall k, D_k = C_{ij}$, para algum $1 \leq j \leq n$. Os elementos de $\mathcal{D}$ são conjuntos disjuntos pela restrição de exclusão.

Seja e_i a expressão contida na parte de descrição da linha de condição estendida C_i . Seja C_{ik} a k -ésima linha de condição limitada derivada da condição estendida C_i . Ela contém na parte de descrição a concatenação de e_i com a entrada D_k . Claramente, ela é um predicado lógico e, portanto, é uma descrição de uma condição limitada. Para completar a linha, precisamos preencher as entradas de condição da seguinte forma:

$$C_{ikj}, 1 \leq j \leq n = \begin{cases} \text{"Y"} & \text{se } C_{ij} = D_k \\ \text{"-"} & \text{se } C_{ij} = \text{"-"} \\ \text{"*"} & \text{caso contrário} \end{cases} \quad \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

Repetindo para $k = 1, 2, \dots, p$, obtemos as p linhas de condição limitadas correspondentes à linha estendida C_i . Por exemplo, na figura 1.6, a tabela (a) é convertida na tabela (b).

Se o usuário decidir estender o símbolo "\$" também para linhas estendidas, apesar de sua pouca utilidade neste caso (vide 1.11), deve-se acrescentar entre as linhas (2) e (3) a seguinte possibilidade:

$$C_{ikj} = \text{"\$"} \text{ se } C_{ij} = \text{"\$"}.$$

O ponto que fica faltando quanto ao processo de transformação de tabelas mistas em limitadas é se ele produz de fato uma tabela equivalente, no sentido de que as mesmas regras são selecionadas nas duas tabelas, executando as mesmas ações em situações idênticas.

Seguindo a mesma notação acima da linha estendida C_i e das linhas obtidas pela sua conversão, vamos denotar por a_{ik} a avaliação de cada condição limitada C_{ik} .

Dizemos que uma regra R_j é *selecionável* numa tabela mista se

$$V(C_{1j}) \cdot V(C_{2j}) \cdot \dots \cdot V(C_{mj}) = 1,$$

	R ₁	R ₂	R ₃	R ₄	R ₅
X	=5	>7	<0	-	>7
Y	= -1	=4	>4	>4	=4
Z=1	Y	Y	Y	N	Y

(a)

	R ₁	R ₂	R ₃	R ₄	R ₅
X = 5	Y	*	*	-	*
X > 7	*	Y	*	-	Y
X < 0	*	*	Y	-	*
Y = -1	Y	*	*	*	*
Y = 4	*	Y	*	*	Y
Y > 4	*	*	Y	Y	*
Z=1	Y	Y	Y	N	Y

(b)

Figura 1.6: Conversão de Tabela Mista em Limitada

onde $V(C_{ij})$ foi definido em 1.6 para as linhas de condição limitada. Para uma linha estendida C_i , $V(C_{ij})$ é calculado da seguinte maneira:

$$\begin{aligned}
 &\text{se } C_{ij} \in \mathcal{D}, \quad V(C_{ij}) = \begin{cases} 1, & \text{se } e_i \in C_{ij} \\ 0, & \text{caso contrário} \end{cases} \\
 &\text{se } C_{ij} = "-", \quad V(C_{ij}) = 1 \\
 &\text{se } C_{ij} = "*", \quad V(C_{ij}) = \begin{cases} 1, & \text{se } e_i \notin \cup_{l=1}^p D_l \\ 0, & \text{caso contrário} \end{cases} \\
 &\text{se } C_{ij} = "\$", \quad V(C_{ij}) = \begin{cases} 1, & \text{se } e_i \in D_1 \quad (\text{neste caso, } p = 1) \\ 0, & \text{caso contrário.} \end{cases}
 \end{aligned}$$

Teorema 5 *Sejam C_i uma linha de condição estendida e $C_{i_1}, C_{i_2}, \dots, C_{i_m}$ as linhas limitadas obtidas pela conversão da primeira pelo procedimento acima descrito. $V(C_{ij}) = 1$ se e somente se $V(C_{i_1j}) \cdot V(C_{i_2j}) \cdot \dots \cdot V(C_{i_pj}) = 1$.*

Dem.:

$\Rightarrow$.

caso 1. $C_{ij} = "*" \text{. Como } V(C_{ij}) = 1, e_i \notin \cup_{l=1}^p D_l \text{. Logo } a_{i_k} = 0, \text{ para todo } 1 \leq k \leq p \text{. Por construção, } C_{i_kj} = "*" \text{. Portanto, } V(C_{i_1j}) \cdot V(C_{i_2j}) \cdot \dots \cdot V(C_{i_pj}) = 1 \text{.}$

caso 2. $C_{ij} = "\$" \text{. Logo } p = 1 \text{. Como } V(C_{ij}) = 1, e_i \in D_1 \text{. Logo } a_{i_1} = 1 \text{. Por construção, } C_{i_1j} = "\$" \text{. Portanto, } V(C_{i_1j}) = 1 \text{.}$

caso 3. $C_{ij} = "-" \text{. Por construção, para todo } 1 \leq k \leq p, C_{i_kj} = "-" \text{. Portanto, } V(C_{i_1j}) \cdot V(C_{i_2j}) \cdot \dots \cdot V(C_{i_pj}) = 1 \text{.}$

caso 4. $C_{ij} \in \mathcal{D} \text{. } C_{ij} \text{ é uma entrada de condição relevante. Seja } C_{ij} = D_k \text{ para algum } 1 \leq k \leq p \text{. Tal } k \text{ existe por definição de } \mathcal{D} \text{. Como } V(C_{ij}) = 1, e_i \in D_k \text{. Logo } a_{i_k} = 1 \text{ e por construção, } C_{i_kj} = "Y" \text{. Portanto, } V(C_{i_1j}) = 1 \text{. Além disso, pela restrição de exclusão, } e_i \notin \cup_{l=1, l \neq k}^p D_l \text{. Logo, } a_{i_l} = 0 \text{ para } 1 \leq l \leq p, p \neq k \text{. Por construção, } C_{i_lj} = "*" \text{. Assim, por definição, } V(C_{i_lj}) = 1 \text{. Portanto, } V(C_{i_1j}) \cdot V(C_{i_2j}) \cdot \dots \cdot V(C_{i_pj}) = 1 \text{.}$

⇐.

Existem os seguintes casos para as entradas $C_{i1j}, C_{i2j}, \dots, C_{ipj}$:

caso 1. Todos têm entrada “-”. É claro que $C_{ij} = \text{“-”}$. Portanto, por definição, $V(C_{ij}) = 1$.

caso 2. Todos têm entrada “*”. É claro que $C_{ij} = \text{“*”}$. Como $V(C_{i1j}) \dots V(C_{ipj}) = 1$, $a_{ik} = 0$, $1 \leq k \leq p$. Logo, $e_i \notin \bigcup_{l=0}^p D_l$. Portanto, $V(C_{ij}) = 1$.

caso 3. Todos têm entrada “\$”. Já vimos que $p = 1$. Claramente, $C_{ij} = \text{“$”}$. Como $V(C_{i1j}) = 1$, então, $a_{i1} = 1$. Assim, $e_i \in D_1$. Portanto, $V(C_{ij}) = 1$.

caso 4. Um deles tem entrada “Y” e os demais “*”. Seja $1 \leq k \leq p$ o índice tal que $C_{ikj} = \text{“Y”}$. Por construção, $C_{ij} = D_k$. Como $V(C_{ikj}) = 1$, então $a_{ik} = 1$. Logo $e_i \in C_{ij}$. Portanto, $V(C_{ij}) = 1$. □

Corolário 6 *Sejam T uma tabela de decisão estendida e W a tabela limitada obtida a partir de T com a conversão das linhas estendidas para limitadas. Uma regra é selecionável em T se e somente se a regra correspondente é selecionável em W .*

Dem.: Como as linhas limitadas são conservadas, aplica-se o teorema 5 para cada linha estendida de T e o seu grupo correspondente de linhas limitadas de W . □

Feito isto, já mostramos exatamente como se faz a transformação de tabelas para a versão limitada. Assim, salvo designação explícita, trataremos somente de tabelas limitadas.

1.13 Seleção de Regras

Junto com a tabela de decisão outras informações são necessárias para uma completa e correta interpretação. Por exemplo, suponhamos que será utilizado um método de tradução da tabela que exige que todas as condições sejam previamente avaliadas. Logo, não pode haver nenhuma condição indeterminada quando iniciada a execução da tabela, mesmo aquelas que seriam irrelevantes para a regra que a instância particular do problema selecionaria. Também é preciso especificar o critério de seleção da regra. As possibilidades são [Me-Ba 77]:

1. Selecionar no máximo 1 regra;
2. Selecionar exatamente 1 regra;
3. Selecionar no mínimo uma regra;
4. Selecionar qualquer número de regras.

Estes critérios tem a ver com a posição frente à ambigüidade. De um modo geral, a detecção de ambigüidade é algo positivo. Supondo que ela não será admitida e sua ocorrência retrataria uma impropriedade na construção da tabela, então somente é possível que uma ou nenhuma regra seja satisfeita, ficando somente os critérios 1 e 2. O critério 2 exige que toda instância do problema selecione exatamente uma regra. Isto requer que toda possibilidade esteja prevista na tabela. Ou seja, não pode ocorrer uma transação² que não satisfaça a nenhuma das regras. O

²Chamaremos de transação a uma particular valoração para o conjunto de condições de uma tabela de decisão.

critério 1 é mais flexível, permite que nenhuma regra seja satisfeita e neste caso, nenhuma ação é executada. Existe a possibilidade de executar-se neste caso uma ação especial chamada "else" rule (vide 1.17).

Por outro lado, se admitimos a ambigüidade, para o critério 1 e 2, há a necessidade de se especificar qual das regras selecionar dentre o conjunto de regras que são selecionáveis. Por exemplo, a primeira da esquerda para a direita. Os critérios 2 e 3 apresentam o problema de se prever todas as possibilidades na tabela de decisão. Os critérios 3 e 4 ainda apresentam o problema de saber em que ordem as diversas regras serão selecionadas. Além disso, a execução das ações de uma regra pode alterar a avaliação das condições. Isto posto, é preciso estabelecer se todas as regras serão selecionadas antes da execução de qualquer uma delas ou se serão executadas à medida em que forem selecionadas, reavaliando as condições.

Podemos ver que estes critérios têm a ver com a ambigüidade e a completude (vide 1.16). O primeiro fator interfere com qual regra escolher, se houver mais que uma selecionável e se isto for permitido. O segundo fator tem a ver com a existência de todas as possíveis situações previstas na tabela.

O critério normalmente utilizado é o de admitir-se que no máximo uma regra ou exatamente uma regra seja selecionada, fazendo a detecção de ambigüidades. A diferença entre estes dois critérios é o de, no primeiro caso, existir a possibilidade de nenhuma regra ser selecionada, ou seja, ação alguma é executada, enquanto o segundo exige que toda situação seja prevista.

1.14 Divisão de uma Tabela de Decisão por uma Condição

Vamos considerar uma tabela de decisão T , limitada e não trivial, com condições $C_1, C_2, \dots, C_m$, ações $A_1, A_2, \dots, A_l$ e regras $R_1, R_2, \dots, R_n$. Seja $R_j^i = (C_{1j}, C_{2j}, \dots, C_{i-1,j}, C_{i+1,j}, \dots, C_{mj}, A_{1j}, A_{2j}, \dots, A_{lj})$, com $1 \leq j \leq n$.

Em outras palavras, R_j^i é a coluna R_j sem a entrada de condição da linha C_i . É óbvio que as partes de ações de R_j e R_j^i são idênticas. Sejam $\mathcal{R}_Y^i$ e $\mathcal{R}_N^i$ conjuntos de regras definidos abaixo:

$$\mathcal{R}_Y^i = \{R_j^i : C_{ij} = "Y", "$" \text{ ou } "-" \text{ em } T\}$$

$$\mathcal{R}_N^i = \{R_j^i : C_{ij} = "N", "*" \text{ ou } "-" \text{ em } T\}.$$

Se $\mathcal{R}_Y^i \neq \emptyset$, seja T_Y^i a tabela de decisão com condições $C_1, C_2, \dots, C_{i-1}, C_{i+1}, \dots, C_m$, ações $A_1, A_2, \dots, A_l$ e conjuntos de regras $\mathcal{R}_Y^i$.

Se $\mathcal{R}_N^i \neq \emptyset$, seja T_N^i a tabela de decisão com condições $C_1, C_2, \dots, C_{i-1}, C_{i+1}, \dots, C_m$, ações $A_1, A_2, \dots, A_l$ e conjuntos de regras $\mathcal{R}_N^i$.

Definição: Se $\mathcal{R}_Y^i \neq \emptyset$ e/ou $\mathcal{R}_N^i \neq \emptyset$, dizemos que T_Y^i e/ou T_N^i são a *Divisão de T pela linha de condição C_i* .

Podemos observar que $\mathcal{R}_Y^i$ e $\mathcal{R}_N^i$ não são vazios simultaneamente. Logo, uma tabela não trivial sempre tem uma divisão por uma linha de condição. Esta é basicamente a definição de Miyakawa [Mi 85] para divisão de uma tabela de decisão, embora a notação seja totalmente diferente, em razão dele adotar a definição funcional para tabelas de decisão.

É simples entender esta operação, pensando em como obter T_Y^i e T_N^i , na prática. Para obter T_Y^i , basta fazer o seguinte:

1. Tome uma cópia de T ,
2. Mantenha todas as regras de T com entradas de condição "Y", "\$" ou "-" na linha de condição C_i , retirando as regras com outras entradas de condição na linha C_i .
3. Retire a linha de condição C_i .

Analogamente, para obtermos T_N^i , usamos o mesmo algoritmo trocando as especificações das entradas por "N", "*" ou "-".

Por exemplo, a tabela T a seguir é dividida pela linha C_1 :

	R_1	R_2	R_3	R_4
C_1	Y	Y	N	N
C_2	Y	N	Y	N
C_3	N	N	Y	N
A_1	1		1	
A_2		1		1
A_3		2	2	

Tabela T

	R_1^1	R_2^1
C_2	Y	N
C_3	N	N
A_1	1	
A_2		1
A_3		2

Tabela T_Y^1

	R_3^1	R_4^1
C_2	Y	N
C_3	Y	N
A_1	1	
A_2		1
A_3	2	

Tabela T_N^1

Pelos critérios de construção, podemos ver que as regras que contêm "-" na linha de condição C_i são repetidas em T_Y^i e T_N^i na divisão de T por C_i .

Nos casos em que a divisão é feita por uma condição qualquer e o índice da condição é irrelevante, trataremos as tabelas T_Y^i e T_N^i somente por T_Y e T_N e as regras R_j^i por R_j , desde que não produzam confusão no texto.

Observamos que as tabelas T_Y e T_N podem ser triviais (conjunto vazio de condições), como no exemplo a seguir:

	R_1	R_2
C_1	Y	N
A_1	1	
A_2		1

Tabela T

	R_1
A_1	1

Tabela T_Y^1

	R_2
A_2	1

Tabela T_N^1

Pela definição apresentada, a divisão de uma tabela por uma condição pode produzir apenas uma tabela. Na figura que segue, temos uma tabela T , cuja divisão pela primeira linha de condição produz somente T_Y^1 .

	R ₁	R ₂
X = Z	Y	Y
W = 10	Y	N
A ₁	1	
A ₂		1

Tabela T

	R ₁	R ₂
W = 10	Y	N
A ₁	1	
A ₂		1

Tabela T_Y^1

Na figura seguinte, temos uma tabela T cuja divisão pela linha de condição produz somente T_N^1 .

	R ₁
X = Z	N
A ₁	1

Tabela T

	R ₁
A ₁	1

Tabela T_N^1

As tabelas T_Y e T_N podem produzir linhas de ação com todas as entradas vazias. Como isto não afronta o conceito de tabelas de decisão, elas podem ser mantidas, sem nenhum problema. Do ponto de vista teórico é até desejável, pela definição dada às regras destas subtabelas, mas do ponto de vista prático³, dependendo da forma como elas são armazenadas pode ser vantajoso eliminá-las. Na opção prática de eliminação, convém deixar pelo menos 1 linha de ação, para não conflitar com a definição inicial de tabelas de decisão, a menos que se faça uma revisão naquela definição. Estas entradas vazias podem ser substituídas por “-” ou pelo número zero. É claro que, na prática, numa tabela representando um problema do mundo real, não existe razão para uma linha deste tipo. Assim, os processadores de tabelas podem fazer uma verificação deste quesito sobre a tabela de entrada, sem nada a opor a este tipo de linhas em tabelas resultantes de uma divisão por uma linha de condição.

Teorema 7 *Seja T uma tabela de decisão limitada com $m > 1$ condições. Sejam T_Y^i e T_N^i a divisão de T por uma linha de condição C_i , $1 \leq i \leq m$. T é não ambígua se e somente se T_Y^i e T_N^i são não ambíguas.*

Dem.:

$\Rightarrow$. Vamos mostrar que T_Y^i não é ambígua. Sejam R_j^i e R_k^i duas regras quaisquer em T_Y^i . Como T não é ambígua, existe $1 \leq p \leq m$ tal que

$$\begin{aligned} C_{pj} &= \text{"Y"} \text{ ou } \text{"\$"} \text{ e } C_{pk} = \text{"N"} \text{ ou } \text{"*"} && \text{ou} \\ C_{pj} &= \text{"N"} \text{ ou } \text{"*"} \text{ e } C_{pk} = \text{"Y"} \text{ ou } \text{"\$"} . \end{aligned}$$

³Esta operação é utilizada no algoritmo de fatorações sucessivas.

Claramente, $i \neq p$, pois caso contrário, por construção uma das regras estaria em T_Y^i e a outra em T_N^i . Logo, a entrada C_{pj} está na regra R_j^i e C_{pk} está na regra R_k^i . Portanto, R_j^i e R_k^i não são ambíguas. Conseqüentemente, T_Y^i não é ambígua.

O mesmo raciocínio demonstra que T_N^i não é ambígua.

$\Leftarrow$. Sejam R_j e R_k duas regras em T .

caso 1. R_j^i e R_k^i estão em subtabelas distintas. Por construção de T_Y^i e T_N^i , $C_{ij} = \text{"Y"}$ ou $\text{"\$"}$ e $C_{ik} = \text{"N"}$ ou "*" , ou vice-versa. Portanto, R_j e R_k não são ambíguas em T .

caso 2. R_j^i e R_k^i estão numa mesma subtabla (ou em ambas). Sem perda de generalidade, suponha que R_j^i e R_k^i estão em T_Y^i . Como R_j^i e R_k^i não são ambíguas em T_Y^i , existe $p \neq i$ tal que $C_{pj} = \text{"Y"}$ ou $\text{"\$"}$ e $C_{pk} = \text{"N"}$ ou "*" , ou vice-versa. Como C_{pj} e C_{pk} estão em R_j e R_k , respectivamente, R_j e R_k não são ambíguas.

Portanto, a tabela T não é ambígua. $\square$

Teorema 8 *Sejam T uma tabela de decisão limitada, T_Y e T_N , a divisão de T por uma condição C_i qualquer. O número equivalente de regras simples de T $n(T) = n(T_Y) + n(T_N)$.*

Dem.: Sejam as regras de T , $R_1, R_2, \dots, R_l$, ordenadas de uma tal forma que:

$R_1, \dots, R_j$ têm entradas de condição "Y", ou "\$" na linha C_i .

$R_{j+1}, \dots, R_k$ têm entradas de condição "N", ou "*" na linha C_i .

$R_{k+1}, \dots, R_l$ têm entradas de condição "-" na linha C_i .

Sejam $p_1, p_2, \dots, p_l$, o número de "-" nas regras $R_1, R_2, \dots, R_l$, respectivamente. Pelo teorema 1, a regra R_s , pode ser desdobrada em 2^{p_s} , $1 \leq s \leq l$, regras simples. Logo, $n(T) = 2^{p_1} + 2^{p_2} + \dots + 2^{p_l}$.

A tabela T_Y , por construção herda as regras $R_1^i, \dots, R_j^i, R_{k+1}^i, \dots, R_l^i$, com respectivamente $p_1, \dots, p_j, p_{k+1} - 1, \dots, p_l - 1$ entradas de condição "-", pois as regras $R_{k+1}, \dots, R_l$ perdem cada uma delas, o símbolo "-", quando passam para a tabela T_Y na divisão. Logo, $n(T_Y) = 2^{p_1} + \dots + 2^{p_j} + 2^{p_{k+1}-1} + \dots + 2^{p_l-1}$.

A tabela T_N , por construção herda as regras $R_{j+1}^i, \dots, R_k^i, R_{k+1}^i, \dots, R_l^i$, com respectivamente $p_{j+1}, \dots, p_k, p_{k+1} - 1, \dots, p_l - 1$ entradas de condição "-". Logo, $n(T_N) = 2^{p_{j+1}} + \dots + 2^{p_k} + 2^{p_{k+1}-1} + \dots + 2^{p_l-1}$.

Assim,

$$\begin{aligned} n(T_Y) + n(T_N) &= 2^{p_1} + \dots + 2^{p_j} + 2^{p_{k+1}-1} + \dots + \\ &\quad 2^{p_l-1} + 2^{p_{j+1}} + \dots + 2^{p_k} + \\ &\quad 2^{p_{k+1}-1} + \dots + 2^{p_l-1} \\ &= 2^{p_1} + 2^{p_2} + \dots + 2^{p_j} + \dots + \\ &\quad 2^{p_k} + \dots + 2^{p_l} \\ &= n(T). \end{aligned}$$

Portanto, $n(T) = n(T_Y) + n(T_N)$. $\square$

Uma operação que, de certa forma, é contrária à divisão e junta duas tabelas numa só, é chamada síntese de tabelas de decisão ("join") [Ch-Ra 76] e atua sobre dois conjuntos de regras de decisão. Como ela não será utilizada nos algoritmos, fica apenas como citação.

1.15 Fatoração de Tabelas de Decisão por uma Linha de Condição

Seja T uma tabela de decisão limitada não trivial e T_Y e T_N a divisão de T por C_i . Seja T' uma tabela de decisão com uma única condição C_i e duas ações A_1 e A_2 , onde A_1 é uma chamada para a execução de T_Y e A_2 para T_N , como mostra a figura a seguir:

C_i	Y	N
Execute T_Y	1	
Execute T_N		1

Chamamos de fatoração de T pela linha de condição C_i ao conjunto de tabelas T' , T_Y e T_N , com execução iniciada por T' . É claro que se uma das duas subtabelas da divisão não existe, então ela também está ausente da fatoração, e T' contém somente uma regra e 1 ação. A operação de divisão não está definida para tabelas triviais, mas a fatoração de uma tabela T , trivial, é a própria tabela T .

Esquemáticamente, a fatoração pode ser representada como na figura 1.7, que é uma árvore de fatoração de T pela linha C_i .

Figura 1.7: Representação da fatoração da tabela T

Como T' possui uma única linha de condição, podemos adotar uma outra forma de esquematização, mais freqüente na literatura, por exemplo, [Po 74], apresentada na figura 1.8.

Como a inversão de T_Y e T_N como filhos direito e esquerdo, respectivamente, produziria uma árvore isomorfa, vamos convencionar que a aresta à esquerda da raiz é rotulada de “Y” e a direita de “N”, com filhos T_Y e T_N esquerdo e direito, respectivamente.

Considere, por exemplo, a tabela de decisão limitada seguinte:

Figura 1.8: Representação usual da fatoração da tabela T

	R_1	R_2	R_3
C_1	Y	N	N
C_2	N	Y	N
C_3	–	Y	N
A_1	1		1
A_2	2	1	

Tabela T

A fatoração da tabela T pela condição C_3 produz o seguinte resultado:

Tabela T_Y Tabela T_N

A representação da fatoração de uma tabela trivial é uma árvore com um só vértice, onde a raiz é a própria folha que contém a tabela trivial.

A execução da fatoração de T consiste na execução da tabela T' e em sequência de uma das tabelas T_Y ou T_N , conforme a condição C_i , utilizada na fatoração, seja verdadeira ou falsa.

Teorema 9 *Sejam T uma tabela de decisão limitada, não trivial e não ambígua, com condições $C_1, C_2, \dots, C_m$ e regras $R_1, R_2, \dots, R_n$, S o vetor de estados do conjunto de condições para uma*

certa transação, T', T_Y^i e T_N^i a fatoração de T pela linha de condição C_i , com execução iniciada por T' . Uma regra R_j é executada em T se e somente se R_j^i é executada a partir de T' em T_Y ou T_N .

Dem.:

$\Rightarrow$. Seja $S = (a_1, \dots, a_m)$ o vetor de estados que seleciona a regra R_j em T . Por definição de seleção de regra, $V(C_{1j}) \cdot V(C_{2j}) \cdots V(C_{mj}) = 1$. Em particular, $V(C_{ij}) = 1$.

caso 1. $a_i = 1$ (A condição C_i é verdadeira). Logo, a execução de T' , produz a execução de T_Y . Como $V(C_{ij}) = 1$ e $a_i = 1$, então $C_{ij} = "Y"$ ou $"\$"$ ou $"-"$. Assim, por construção, R_j^i está em T_Y . Como $V(C_{1j}) \cdots V(C_{i-1,j}) \cdot V(C_{i+1,j}) \cdots V(C_{mj}) = 1$, a regra R_j^i é selecionada em T_Y .

caso 2. $a_i = 0$ (A condição C_i é falsa). Logo, a execução de T' , produz a execução de T_N . Como $V(C_{ij}) = 1$ e $a_i = 0$, então $C_{ij} = "N"$ ou $"*"$ ou $"-"$. Assim, por construção, R_j^i está em T_N . Como $V(C_{1j}) \cdots V(C_{i-1,j}) \cdot V(C_{i+1,j}) \cdots V(C_{mj}) = 1$, a regra R_j^i é selecionada em T_N .

Portanto, a regra R_j^i é selecionada na execução da fatoração de T .

$\Leftarrow$. Seja $S = (a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_m)$, o vetor de estados que seleciona a regra R_j^i na execução da fatoração de T . Logo $V(C_{1j}) \cdots V(C_{i-1,j}) \cdot V(C_{i+1,j}) \cdots V(C_{mj}) = 1$.

caso 1. R_j^i é selecionada em T_Y . Para que T_Y seja executada a partir de T' é preciso que a aresta $"Y"$ seja escolhida na árvore de fatoração, isto é, $a_i = 1$. Por construção de T_Y , $C_{ij} = "Y"$, $"\$"$ ou $"-"$. Assim, $V(C_{ij}) = 1$. Portanto, $V(C_{1j}) \cdots V(C_{i-1,j}) \cdot V(C_{ij}) \cdot V(C_{i+1,j}) \cdots V(C_{mj}) = 1$.

caso 2. R_j^i é selecionada em T_N . Para que T_N seja executada a partir de T' é preciso que a aresta $"N"$ seja escolhida na árvore de fatoração, isto é, $a_i = 0$. Por construção de T_N , $C_{ij} = "N"$, $"*"$ ou $"-"$. Assim, $V(C_{ij}) = 1$. Portanto, $V(C_{1j}) \cdots V(C_{i-1,j}) \cdot V(C_{ij}) \cdot V(C_{i+1,j}) \cdots V(C_{mj}) = 1$.

Portanto, nos dois casos, a regra R_j é selecionada em T . $\square$

Este resultado, embora simples, é importante, pois garante a equivalência entre a execução de uma tabela de decisão T ou da sua fatoração por uma linha de condição qualquer, pois ambas selecionam regras correspondentes. Além disso, a parte de ações de R_j e R_j^i são idênticas. Deste modo, exatamente as mesmas ações serão executadas tanto aplicando a tabela T ou a sua fatoração por uma condição. Além disso, o teorema acima garante que se nenhuma regra for selecionável em T nenhuma será na execução da fatoração. Isto significa que em caso de explicitada e selecionada a "else rule" (vide 1.17) em T , ela será selecionada também na execução de T' .

1.16 Completude de uma Tabela de Decisão

Considere uma tabela de decisão com única condição $"X = 10"$ e duas regras não ambíguas com entradas $"Y"$ e $"N"$.

$X = 10$	Y	N
A_1	1	
A_2		1

Como a condição somente pode assumir os valores verdadeiro ou falso, é claro que qualquer valoração para variável X satisfaz uma das regras. Dizemos que a tabela acima é completa.

Uma tabela de decisão é *completa* se toda valoração para as variáveis de condição contempla uma situação que está prevista na tabela, isto é, para todo vetor de estado $S = (a_1, a_2, \dots, a_m)$, existe uma função-and B_j tal que $V(B_j) = 1$, ou seja, $V(C_{kj}) = 1, 1 \leq k \leq m$. Em outras palavras, qualquer que seja a transação, existe uma regra que é satisfeita pelas condições e, portanto, selecionável.

Teorema 10 *Sejam T uma tabela de decisão limitada, e T_Y e T_N a divisão de T pela linha de condição C_i . Se T é completa, então T_Y e T_N são completas.*

Dem.: Vamos mostrar que T_Y é completa.

Sejam $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_m$ as linhas de condição de T_Y . Vamos supor por contradição que T_Y não é completa. Logo existe uma valoração para as condições com vetor de estados $S = (a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_m)$ para o qual nenhuma regra em T_Y é satisfeita, ou seja, para toda regra R_k em T_Y , existe $1 \leq j \leq m, j \neq i$, tal que $V(C_{jk}) = 0$. Seja S' a valoração sobre o conjunto de condições de T , com valores iguais aos de S para as condições $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_m$ e $a_i = 1$.

Assim, $S' = (a_1, \dots, a_{i-1}, 1, a_{i+1}, \dots, a_m)$. É claro que todas as regras R_k com entradas "N" ou "*" na condição C_i , produz $V(C_{ik}) = 0$, para esta transação e tais regras não são selecionadas em T . Por construção, todas as demais regras estão em T_Y . Como para toda regra R_k em T_Y , existe $C_{jk}, 1 \leq j \leq m, j \neq i$, tal que $V(C_{jk}) = 0$, então nenhuma destas regras é selecionável em T pela transação. Logo, a situação que produz a valoração S' sobre o conjunto de condições de T não satisfaz a nenhuma regra em T . Mas isto é uma contradição, pois T é uma tabela completa. Esta contradição mostra que T_Y é completa. Raciocínio análogo para T_N estabelece o teorema. $\square$

Teorema 11 *Seja T uma tabela de decisão limitada, não ambígua e com m condições independentes. T é completa se e somente se ela possui o equivalente a $n(T) = 2^m$ regras simples.*

Dem.:

$\Rightarrow$. Por indução em m .

Base da indução: $m = 0$. Neste caso, T é trivial. Como ela não é ambígua, $n(T) = 1 = 2^0$.

Suponha que a tese é válida para tabelas com menos que $m > 0$ condições. Seja C uma condição qualquer do conjunto de condições de T e T_Y e T_N a divisão de T por C . Por construção, T_Y e T_N têm $m - 1$ condições. Como T é completa, pelo teorema 10, T_Y e T_N são completas. Como T não é ambígua, pelo teorema 7, T_Y e T_N não são ambíguas. As condições de T_Y e T_N são herdadas de T e, portanto, são independentes. Assim, podemos aplicar a hipótese indutiva em T_Y e T_N e obtemos $n(T_Y) = n(T_N) = 2^{m-1}$.

Pelo teorema 8,

$$\begin{aligned} n(T) &= n(T_Y) + n(T_N) \\ &= 2^{m-1} + 2^{m-1} \\ &= 2^m. \end{aligned}$$

$\Leftarrow$. Seja $S = (a_1, a_2, \dots, a_m)$ o vetor de estados das m condições de T . Como $a_i \in \{0, 1\}$, existem 2^m possíveis estados para S . Como T possui $n(T) = 2^m$ regras simples distintas, T é completa. $\square$

Desta forma, para tabelas não ambíguas com condições independentes, temos um método fácil de verificação da completude. Primeiramente, verifica-se a ambigüidade conforme 1.8. Feito isto, basta contar o número de regras simples que a tabela representa. Pela expressão vista anteriormente:

$$n(T) = 2^{p_1} + 2^{p_2} + \dots + 2^{p_n},$$

onde $p_1, p_2, \dots, p_n$ são os números de símbolos “-” nas regras $R_1, R_2, \dots, R_n$, respectivamente.

Vamos discutir a complexidade deste procedimento, excluindo a verificação da ambigüidade que já foi analisada.

Para efetuar a tarefa simples acima de contar o número de regras simples representada por uma tabela T com m condições e n regras, basta consultar as $m \cdot n$ entradas de condição da tabela, percorrendo-a por coluna e contando organizadamente os símbolos “-”. A seguir, está esquematizado um algoritmo que recebe uma tal tabela e calcula $n(T)$.

Algoritmo: Número de regras simples representadas em T .

Importação: Tabela T .

Exportação: $n(T)$, o número de regras simples representadas em T .

início

```

1.       $n(T) := 0;$ 
2.      para cada coluna  $j, 1 \leq j \leq n$  faça
3.           $P_j := 1;$ 
4.           $k := 0;$ 
5.          para cada  $i, 1 \leq i \leq m$  faça
6.              se  $C_{ij} = \text{“-”}$  então
7.                   $P_j := 2 \cdot P_j$ 
8.                   $k := k + 1;$ 
9.           $n(T) := n(T) + P_j;$ 
fim
```

Os comandos das linhas 4 e 8 são desnecessários numa implementação. Eles estão presentes para possibilitar uma discussão sobre a “corretude” do algoritmo.

Lema 12 A expressão $P_j = 2^k$ é um invariante do “loop” interno do algoritmo acima, que calcula o número de regras simples representadas por uma tabela T .

Dem.: Por indução no número de iterações do “loop” interno.

Base da indução: Entrada da primeira iteração. Por definição das variáveis no algoritmo, $P_j = 1$ e $k = 0$. Como $2^0 = 1$, a expressão está correta neste caso.

Vamos mostrar que a tese é válida para uma iteração $i > 0$, supondo que ela é satisfeita para todas iterações anteriores. Para toda variável x na iteração i , vamos representar por x' a variável correspondente na $(i - 1)$ -ésima iteração. Temos que analisar dois casos:

caso 1. $C_{ij} \neq \text{“-”}$. Neste caso, as linhas 7 e 8 não são executadas. Logo, $P_j = P'_j$. Aplicando a hipótese de indução em P'_j , a tese está garantida.

caso 2. $C_{ij} = \text{“-”}$. As linhas 7 e 8, produzem

$$P_j = 2 \cdot P'_j \quad \text{e} \quad k = k' + 1.$$

Aplicando a hipótese indutiva sobre P'_j , obtemos

$$\begin{aligned} P_j &= 2 \cdot (2^{k'}) \\ &= 2^{k'+1} \\ &= 2^k. \quad \square \end{aligned}$$

Teorema 13 *Seja T uma tabela de decisão com condições $C_1, C_2, \dots, C_m$, e $p_1, p_2, \dots, p_n$ o número de símbolos “-” nas regras $R_1, R_2, \dots, R_n$, respectivamente. O algoritmo que calcula o número de regras simples representadas por T está correcto.*

Está claro que o algoritmo termina, pois os “loops” são limitados pelos valores de m e n . É fácil ver que o comando da linha 9, calcula

$$n(T) = \sum_{j=1}^n P_j,$$

onde P_j é o valor da variável no momento de sua saída do “loop” interno. Pelo lema 12, para cada valor de j , $P_j = 2^k$ é um invariante do “loop” interno. Como na saída do “loop” interno, $k = p_j$, então, no comando da linha 9, $P_j = 2^{p_j}$. Portanto, o algoritmo calcula

$$n(T) = \sum_{j=1}^n 2^{p_j}$$

que, como vimos em 1.4, é o número de regras representadas por T . $\square$

Podemos ver facilmente que o algoritmo executa n vezes o “loop” externo e $m \cdot n$ vezes o “loop” interno. Assim, ele faz $m \cdot n$ operações de comparação, no máximo este mesmo número de multiplicações e n operações de soma. Temos então que o tempo gasto t é :

$$t \leq 2 \cdot m \cdot n + n$$

tomando como unidade de tempo a operação mais custosa dentre as envolvidas. Logo a complexidade é $O(m \cdot n)$. Em termos do comprimento da entrada $l = m \cdot n$, é $O(l)$.

1.16.1 Completude em Tabelas com condições dependentes

Considere uma tabela de decisão com condições arbitrárias C_1 e C_2 como mostra a figura 1.9(a).

Como a tabela da figura 1.9(a) não é ambígua e tem 4 regras simples, pelo teorema 11, ela é completa. Considere agora que as condições sejam substituídas por

$$X = 5 \quad \text{e} \quad X = 10,$$

respectivamente. Claramente, a regra R_1 não ocorre e além disso, se $X = 5$, então a condição $X = 10$ é obrigatoriamente falsa. A forma, mais correcta de descrever o problema é através da tabela da figura 1.9(b) que com apenas três regras é completa, no sentido de que todas

	R ₁	R ₂	R ₃	R ₄
C ₁	Y	Y	N	N
C ₂	Y	N	Y	N

(a)

	R ₁	R ₂	R ₃
X = 5	Y	*	N
X = 10	*	Y	N

(b)

Figura 1.9: Tabelas Completas

as possibilidades estão previstas. Não existe a quarta possibilidade, como seria esperado em tabelas com condições independentes [Ha 71].

A completude torna-se difícil de ser verificada se as condições são dependentes, sobretudo se forem implicitamente dependentes. Os detalhes relacionados com a semântica de um problema particular não é uma tarefa fácil de ser analisada automaticamente [Ib-Ra 78].

No caso de tabelas estendidas ou mistas, o problema se repete, pois as linhas de condição estendidas são uma forma diferente de representar um conjunto de condições explicitamente dependentes.

Observemos a tabela (a) da figura seguinte composta de uma única linha de condição estendida (a parte de ações não interessa para o exemplo):

X =	1	2	3	4
-----	---	---	---	---

(a)

X = 1	Y	*	*	*
X = 2	*	Y	*	*
X = 3	*	*	Y	*
X = 4	*	*	*	Y

(b)

Uma linha de condição assim descrita, leva a ser interpretada como se o usuário afirmasse que as situações com $X \leq 0$ ou $X \geq 5$ não ocorrem ou nada deve ser feito se ocorrerem, ou ainda, uma situação imprevista. Vamos supor que para o problema particular, tais situações não ocorrem. Logo a tabela estendida acima é completa.

Considerando a tabela (b), da figura acima, obtida pela transformação das linhas estendidas da tabela (a), observamos uma tabela com 4 linhas de condição limitadas. Assim, temos uma tabela limitada, completa, com apenas 4 regras simples. Totalmente diferente do esperado para tabelas completas com condições independentes. Tal diferença fica por conta da dependência entre as condições da tabela limitada que foi derivada da tabela estendida.

1.17 “Else Rule”

Existe uma regra especial muito difundida nos sistemas de tabela de decisão chamada “else rule”. Esta regra “senão” é muito útil para contornar o problema da completude, como também

amplia a facilidade de representação dos problemas.

A “else rule” é a regra especificada na tabela de decisão, indicando que ela deve ser selecionada sempre que nenhuma das regras $R_1, R_2, \dots, R_n$ é satisfeita. Uma tabela com esta possibilidade precisa ser escrita com este requisito. Existem muitas formas de escrevê-la [Mo 74, Lo 72, Po 74], mas vamos adotar a que está na figura 1.10, que parece ser a mais usada.

	R_1	R_2	R_3	ELSE
C_1	Y	N	Y	
C_2	Y	Y	N	
C_3	N	N	N	
A_1	1			3
A_2		1		2
A_3	2	2	1	
A_4			2	1

(a)

	R_1	R_2	R_3	R_4	ELSE
X	1	2	3	≥ 4	
A_1	1	2	1		
A_2	2	1		1	
Erro					1

(b)

Figura 1.10: Tabelas com “else rule”

Como podemos observar no exemplo da figura 1.10(a), se a avaliação das condições não satisfizer a nenhuma das regras R_1, R_2 e R_3 , então serão executadas as ações A_4, A_2 e A_1 , nesta ordem. A “else rule” garante a completude de uma tabela. Pode ser empregada em tabelas mistas, estendidas e limitadas.

A figura 1.10(b) mostra um exemplo de uso da “else rule” em tabela estendida. No caso da tabela ser limitada com condições independentes e não ambígua, o número de regras simples representadas pela “else rule” é

$$2^m - (2^{p_1} + 2^{p_2} + \dots + 2^{p_n}),$$

onde m é o número de linhas de condição e $p_1, \dots, p_n$, é o número de símbolos “-” nas regras $R_1, \dots, R_n$, respectivamente. Isto é óbvio, pois 2^m é o número de regras simples de uma tabela completa e $2^{p_1} + 2^{p_2} + \dots + 2^{p_n}$ é o número de regras simples representadas pelas regras $R_1, \dots, R_n$. No exemplo da tabela da figura 1.10, que é uma tabela limitada e supondo que as condições fictícias C_1, C_2 e C_3 são independentes, a “else rule” representa $2^3 - 3 = 5$ regras simples.

Para uma tabela de decisão limitada com condições independentes a regra “else” sempre pode ser escrita em termos de regras convencionais. Basta escrever, uma a uma, as regras representadas por ela, mas isto pode ser um trabalho cansativo.

1.17.1 Estratégia para a “else rule”

Existem ponderações na literatura no sentido de se evitar o uso da “else rule”, para que os usuários sejam forçados a pensar explicitamente em todas as situações possíveis do problema. Citamos algumas estratégias na sua utilização.

Erro automático para “else rule”

A “else rule” é reservada para uma rotina interna especial de erro, informando que nenhuma regra foi selecionada. Nesta situação, podemos ainda ter duas alternativas: interromper a execução da tabela de decisão ou prosseguir. Neste caso, a “else rule” não está presente na tabela e é automaticamente ativada.

	R ₁	R ₂	R ₃
C ₁	Y	N	Y
C ₂	N	N	Y
A ₁	1		1
A ₂		1	2

Neste exemplo, a rotina de erro seria ativada automaticamente no caso de $C_1 = false$ e $C_2 = true$. Na verdade, esta estratégia não é propriamente uma “else rule”, no sentido de que ela não é especificada na tabela, mas um mecanismo automático de detecção de tabelas incompletas, com a execução de uma rotina de erro padrão ou emissão de um simples aviso.

Regra de erro para a “else rule”

A “else rule” é reservada somente para o caso de erro, mas diferentemente da estratégia anterior, torna obrigatória a especificação da rotina de erro a ser executada. O problema é que diversas rotinas de erros podem ser necessárias ao problema. Assim, deveria ser utilizada para erro do tipo “Regra imprevista”.

	R ₁	R ₂	R ₃	ELSE
C ₁	Y	N	Y	
C ₂	N	N	Y	
A ₁	1		1	
A ₂		1	2	
Erro				1

Neste exemplo, a tabela ativa uma rotina de erro explicitamente especificada caso as regras R_1 e R_2 não sejam selecionadas.

Regra vazia para a “else rule”

A “else rule” é reservada para não se executar absolutamente nada. Existem problemas em que em diversas situações não se deseja executar nenhuma ação. Uma possível saída para isto é utilizar a regra “else” com esta finalidade.

Suponha o seguinte problema: Seja S uma cadeia arbitrária de caracteres do alfabeto. Deseja-se saber quantos caracteres ‘A’, ‘D’, ‘G’, ‘T’ e ‘Z’ existem na cadeia, contabilizando a ocorrência de cada uma delas em n_A , n_D , n_G , n_T e n_Z , respectivamente. Considere a tabela de

decisão estendida a seguir, que resolve o problema, no que se refere à decisão. Estamos supondo que outros detalhes como inicialização de contadores e os comandos de incremento são feitos em locais apropriados.

X =	'A'	'D'	'G'	'T'	'Z'
Incrementa	n_A	n_D	n_G	n_T	n_Z

A título de ilustração de como seria uma linha de ação estendida, escrevemos a tabela nesta forma, embora não seja muito usual, em razão da necessidade de incluir mais informações nas entradas de ação para preservar a ordem de sua execução. Neste caso, como existe uma única ação para cada regra, este problema não existe.

O problema requer que esta tabela seja executada uma vez para cada símbolo da cadeia S . Na linha de condição, X representa o elemento de S que está sendo investigado. A princípio, todos os símbolos do alfabeto podem estar presentes e desta forma a tabela não está completa. O problema requer que nada seja feito se o símbolo for diferente daqueles que estão na tabela. Assim, a tabela completa requer a presença de uma lista de todos os demais elementos do alfabeto com uma regra vazia (operação “NOP”). Isto exigiria um esforço e espaço razoável, mesmo que fosse possível especificar na forma de pertinência de conjuntos:

$$X \in \{ 'B', 'C' \} \cup \{ 'E', 'F' \} \cup \{ 'H', \dots, 'S' \} \cup \{ 'U', \dots, 'Y' \}.$$

Assim, se assumirmos a “else rule” como sendo uma regra implícita a ser automaticamente selecionada quando nenhuma regra é satisfeita, e que executa um conjunto vazio de ações, este problema estaria facilmente solucionado. Dentro desta estratégia, a tabela acima descrita resolveria o problema.

“Else rule” como uma regra normal

Suponhamos, agora, uma ligeira modificação do problema anterior. Além de contabilizar separadamente as ocorrências dos símbolos ‘A’, ‘D’, ‘G’, ‘T’ e ‘Z’, deseja-se contar o número de caracteres diferentes de todos eles. Se a linguagem de descrição da tabela não permite a notação de pertinência a conjunto, o problema seria extremamente tedioso, na construção da seguinte tabela.

X=	'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	...	'Z'
Incrementa	n_A	m	m	n_D	m	m	n_G	m		n_Z

A variável m acumula o número de caracteres diferentes de ‘A’, ‘D’, ‘G’, ‘T’ e ‘Z’. Se a linguagem de descrição da tabela permite o uso de pertinência a conjuntos, poderíamos definir os seguintes conjuntos e escrever a tabela de decisão de uma forma diferente.

$$\begin{aligned}
C_A &= \{'A'\} \\
C_D &= \{'D'\} \\
C_G &= \{'G'\} \\
C_T &= \{'T'\} \\
C_Z &= \{'Z'\} \\
C &= \{'B','C'\} \cup \{'E','F'\} \cup \{'H',\dots,'S'\} \cup \\
&\quad \{'U',\dots,'Y'\},
\end{aligned}$$

X∈	C_A	C_D	C_G	C_T	C_Z	C
Incrementa	n_A	n_D	n_G	n_T	n_Z	m

Por outro lado, se adotarmos a estratégia de que a “else rule” deve ser explicitamente especificada e que se deve executar a ação associada a ela, independentemente de ser uma rotina de erro, este problema fica facilmente resolvido com a tabela seguinte.

	R_1	R_2	R_3	R_4	R_5	ELSE
X=	'A'	'D'	'G'	'T'	'Z'	
Incrementa	n_A	n_D	n_G	n_T	n_Z	m

Podemos ver que todas as estratégias apresentadas têm a sua utilidade. A escolha por uma delas deve considerar a adequação aos problemas que serão tratados. Mas a melhor alternativa é escolher as três últimas estratégias simultaneamente, já que podem não ser conflitantes, oferecendo maiores alternativas ao usuário e conferindo maior poder às tabelas de decisão. Para não haver conflito, nesta escolha, permite-se tabelas com ou sem a especificação da “else rule”. A sua ausência ou não determina qual estratégia seguir. É esta a opção feita no caso do pré-processador a ser apresentado posteriormente. A segunda estratégia é um caso particular da última em que a ação especificada se limita a ser uma rotina de erro. Assim, se não estiver especificada a “else rule”, adota-se a terceira estratégia que é executar uma regra vazia. Se, ao contrário houver uma “else rule”, especificada explicitamente na tabela de decisão, executam-se as ações lá consignadas, como se fossem as de uma outra regra qualquer, seguindo a última estratégia.

1.18 Linguagem de Tabela de Decisão

Convencidos de que a tabela de decisão é um mecanismo adequado para a especificação de decisões de um problema, desde o projeto até a documentação, passando pela implementação e testes, alguns pesquisadores trabalharam no sentido de converter programas em tabelas de decisão [Ca 74]. Podemos perguntar se tabelas podem descrever problemas completos. Lew [Le 82b, Le 83] defende a aplicabilidade das tabelas de decisão em todas as aplicações, inclusive científica e dá exemplos de pequenos problemas especificados totalmente em tabelas de decisão.

Naturalmente, uma sequência de ações é uma tabela trivial; Os comandos de decisão são a própria essência das tabelas de decisão. As iterações podem ser facilmente obtidas através de chamadas a outras tabelas ou a si mesmas, sendo que a execução das ações pode alterar o valor das condições para a próxima iteração. Com a capacidade de incorporar os comandos de sequência, decisão e iteração, podemos entender que um programa pode ser totalmente escrito através de tabelas de decisão. Embora reafirmando a utilidade delas, Maes [Ma 78] contestou esta forma de utilização, afirmando que, comparativamente, elas não possuem a capacidade de expressão dos fluxogramas. Neste sentido, dado um fluxograma, nem sempre é possível obter uma tabela de decisão equivalente, sem introduzir complicações. Isto se deve á possibilidade de avaliar condições entre duas ações nos fluxogramas, o que não é permitido nas tabelas de decisão, isto é, nestas últimas as condições são avaliadas antes da execução das ações, o que não é feito obrigatoriamente nos fluxogramas. Outro problema mais crucial são as iterações e seus encaixamentos. Uma tabela de decisão, normalmente, seria executada inteiramente a cada iteração, mas num problema genérico, nem sempre isto é válido. Para superar estas dificuldades, muitas sugestões foram feitas, como a inclusão de rótulos para cada linha de condição e de ação, para que possam ser referenciadas por comandos de desvio [Mo-Hi 72] e permitam que iterações possam ser iniciadas em pontos distintos da tabela. Isto, na verdade, propõe uma ordenação sobre o conjunto das condições de uma tabela de decisão, contrariando frontalmente a interpretação usual, onde a ordem das linhas de condição e das regras é totalmente irrelevante. Outra sugestão [Mu 84, Mi-Ch 89] originalmente descrita para fins de documentação, chamada de Tabela de decisão estruturada, consiste num sistema de blocos encaixados dentro de uma tabela de decisão, para permitir as iterações e testes de condição entre as ações a serem executadas, além de permitir a especificação de uma regra R_0 de inicialização, no caso de a tabela ser um processo iterativo.

Todas estas modificações no formato introduzem complicações que acabam por eliminar a clareza das tabelas de decisão, perdendo as suas vantagens inerentes. Como veremos adiante, a nossa tabela de decisão mantém o formato tradicional, somado a um novo elemento que não prejudica a sua transparência e aumenta o seu poder de notação. A utilização que damos às tabelas de decisão é somente como um mecanismo alternativo de especificação das decisões, o que já é um grande serviço, sem pretender que elas sozinhas descrevam completamente todos os problemas.

Capítulo 2

Tradução de Tabelas de Decisão

Uma tabela de decisão serve bem para uma visualização humana, mas precisa ser processada, para que seja executada por uma máquina. Os processadores traduzem as especificações de uma tabela para algum código. Entre os sistemas baseados em tabelas que produzem código de baixo nível estão, por exemplo, o LOBOC [Me-Ba 77, Histórico] e o DELTA [Le 71] e existem vários processadores que convertem tabelas em linguagem de alto nível como o TABSOL [St 71], o DETAB/65 [Ca-Ch 67, Si 71] e o DETAP/55 [We 71] que produzem comandos em COBOL. Outras linguagens também são produzidas como o BASIC [Ar 71], Algol, PL/I, Fortran [Me-Ba 77, Tabela 3.1], Pascal e C [Dl 89]. Os processadores que produzem linguagem de alto nível são chamados de pré-processadores [Lo 72]. De qualquer modo, todos eles precisam passar por uma etapa de tradução, mudando apenas o tipo do código produzido, além de outros tratamentos que podem ser realizados, como verificação de ambigüidades. Para exemplificar, vamos traduzir, à mão, a tabela de decisão a seguir, produzindo comandos em Pascal:

X = 0	Y	Y	N
Y = 0	N	Y	-
X := Y	1		1
Y := -1		1	2

```
if X = 0 then
  if Y = 0 then
    Y := -1
  else X := Y
else
  begin
    X := Y;
    Y := -1;
  end;
```

Esta sequência de comandos é uma possível tradução para a tabela anterior.

Existem diversas técnicas para tradução de uma tabela, sendo que cada uma delas comportam diversos algoritmos. Existem basicamente quatro métodos [Me-Ba 77, Po 74]

- “Switch Method”
- Teste seqüencial
- Método de máscara
- Método de produção de árvores de decisão.

Os métodos se diferenciam pela maneira como a tradução é feita, como também pelos distintos códigos que produzem. O primeiro usa a técnica de linearização de uma matriz multi-dimensional, para a seleção da regra. O segundo método é o trivial, ou seja, aquele de teste seqüencial regra por regra. O terceiro produz no código gerado uma matriz que retrata as propriedades da tabela de decisão. O último, o mais largamente utilizado [Me-Ba 77, Tabela 3.4, página 67], também chamado na literatura internacional de “test and branch method” [Me-Ba 77] comporta duas classes de algoritmos: o de fatorações sucessivas ou método de decomposição [Po-Hi-Ha 71, Za 84] que é um algoritmo semi-ótimo e os algoritmos ótimos para construção de árvores de decisão.

2.1 “Switch method”

Este método tem este nome por utilizar uma variável “switch” na implementação em ALGOL [Me-Ba 77]. No FORTRAN pode utilizar o comando “GOTO” computado e no COBOL o comando “GO TO ... DEPENDING ON ...”.

Veinott [Ve 66a] descreveu este método como sendo adequado para uma tradução manual para linguagens de alto nível, exigindo tabelas completas com a especificação de todas as regras simples, incluindo as semanticamente impossíveis de ocorrer. Além disso, exigia linhas de condição numa forma que chama de “multivalorada”. Isto significa que cada condição individualmente deve ser completa (v. 1.16), no sentido de que cada condição deve receber entradas relevantes e todos os valores possíveis na linha correspondente. E também deve estar de acordo com a restrição de exclusão. Outra necessidade do algoritmo, então apresentado, é o requisito de que as regras fossem ordenadas como será visto mais adiante, mas isto não é fundamental. O inconveniente da não ordenação é um maior tempo de computação para buscar a regra correspondente a um determinado número. Mesmo a restrição da completude foi relaxada em artigo posterior do próprio autor [Ve 66b]. Um algoritmo similar foi apresentado por Smillie & Shave [Sm-Sh 75].

O método consiste em atribuir um número único para cada combinação de condições. As m condições são representadas por m variáveis e cada uma delas assume valores inteiros diferentes, começando de zero, para cada valor testado desta condição com respeito às entradas de condição. Considere uma tabela de decisão com condições $C_1, \dots, C_m$ e entradas de condição $C_{ij}, 1 \leq j \leq n, 1 \leq i \leq m$. Seja $\mathcal{C}_i = \{c_{i1}, c_{i2}, \dots, c_{in_i}\}$ o conjunto de entradas (distintas) de condição da linha $C_i, 1 \leq i \leq m$ e $n_i \leq n$. No caso de C_i ser uma linha de condição limitada, $n_i = 2$, pois C_i pode assumir valores “Y” e “N”. Nos demais casos, n_i é o número de diferentes conjuntos de valores definidos nas entradas de condição.

Segundo o algoritmo, para cada condição C_i , $1 \leq i \leq m$ deve ser produzido uma variável auxiliar I_i para cada condição C_i , $1 \leq i \leq m$ e os seguintes comandos:

```

if  $C_i = c_{i1}$  then  $I_i := 0$ 
else if  $C_i = c_{i2}$  then  $I_i := 1$ 
    :
    :
else if  $C_i = c_{i n_i - 1}$  then  $I_i := n_i - 2$ 
else  $I_i := n_i - 1$ ;

```

Feito isto é gerada para a tabela uma variável I para indexação da regra ("switch variable" de Algol) e a seguinte expressão:

$$I := (n_2 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot \dots \cdot n_m) \cdot I_2 + \dots + (n_m) \cdot I_{m-1} + I_m + 1 \quad (2.1)$$

Esta expressão é utilizada para a linearização de uma matriz de várias dimensões. A constante 1 somada na expressão do índice I é necessária para que o método funcione em FORTRAN no comando "goto" computado.

A expressão para a variável I produz valores de 1 a $n_1 \cdot n_2 \cdot \dots \cdot n_m$. Este limite superior para I pode ser obtido fazendo o cálculo da expressão, substituindo-se as variáveis I_i pelo seu valor máximo, $n_i - 1$: De fato, seja I' o valor máximo assumido pela atribuição acima; então

$$I' = 1 + \sum_{i=1}^m I'_i,$$

onde

$$I'_i = \begin{cases} (n_{i+1} \cdot n_{i+2} \cdot \dots \cdot n_m) \cdot (n_i - 1), & 1 \leq i < m \\ n_m - 1, & i = m. \end{cases}$$

Assim,

$$\begin{aligned}
 I'_1 &= n_2 \cdot n_3 \cdot n_4 \cdot \dots \cdot n_m \cdot (n_1 - 1) \\
 &= n_1 \cdot n_2 \cdot n_3 \cdot \dots \cdot n_m - n_2 \cdot n_3 \cdot \dots \cdot n_m \\
 I'_2 &= n_3 \cdot n_4 \cdot \dots \cdot n_m \cdot (n_2 - 1) \\
 &= n_2 \cdot n_3 \cdot n_4 \cdot \dots \cdot n_m - n_3 \cdot n_4 \cdot \dots \cdot n_m \\
 &\vdots \\
 &\vdots \\
 I'_{m-2} &= n_{m-1} \cdot n_m \cdot (n_{m-2} - 1) \\
 &= n_{m-2} \cdot n_{m-1} \cdot n_m - n_{m-1} \cdot n_m \\
 I'_{m-1} &= n_m \cdot (n_{m-1} - 1) = n_{m-1} \cdot n_m - n_m \\
 I'_m &= n_m - 1.
 \end{aligned}$$

Portanto, $I' = n_1 \cdot n_2 \cdot \dots \cdot n_m$.

Para completar a tradução, deve ser produzido o comando "goto" indexado para o FORTRAN ou o "case" para o Pascal, onde o valor de I calculado pela fórmula 2.1 indica que a I -ésima regra deve ser selecionada. Da mesma forma como o algoritmo calcula o valor de I para

uma certa valoração do conjunto de condições durante a execução, podemos calcular o seu valor para cada uma das regras. A ordenação acima citada significa que este valor obtido para a regra R_j é maior que o obtido para a regra R_i se $j > i$. Na verdade, como $1 \leq I \leq n_1 \cdot n_2 \cdot \dots \cdot n_m$ e este limite superior é o número de regras, a ordenação faz com que $I = j$, para a regra R_j . Esta ordenação garante que a I -ésima regra tem os seguintes elementos nas entradas de condição:

$$\begin{aligned} C_{1I} &= c_{1,I_1+1} \\ C_{2I} &= c_{2,I_2+1} \\ &\vdots \\ C_{mI} &= c_{m,I_m+1} \end{aligned}$$

Vamos examinar a seguinte tabela que traz uma linha de ação estendida. Note a ordenação das regras. Como cada entrada de condição corresponde a um número inteiro, podemos associar a cada regra o número calculado pela fórmula 2.1, cujo valor dá a ordenação que é proposta pelo método.

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}
$Z=0$	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
$Y=X$	Y	Y	Y	N	N	N	Y	Y	Y	N	N	N
X	≤ 0	1	≥ 2	≤ 0	1	≥ 2	≤ 0	1	≥ 2	≤ 0	1	≥ 2
execute	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}

```

if  $Z = 0$  then  $I_1 := 0$ 
    else  $I_1 := 1$ ;
if  $Y = X$  then  $I_2 := 0$ 
    else  $I_2 := 1$ ;
if  $X \leq 0$  then  $I_3 := 0$ 
    else if  $X = 1$  then  $I_3 := 1$ 
        else  $I_3 := 2$ ;
 $I := 2 * 3 * I_1 + 3 * I_2 + I_3 + 1$ ;
case  $I$  of
    1 :  $A_1$ ;           2:  $A_2$ ;           3:  $A_3$ ;
    4 :  $A_4$ ;           5:  $A_5$ ;           6:  $A_6$ ;
    7 :  $A_7$ ;           8:  $A_8$ ;           9:  $A_9$ ;
    10 :  $A_{10}$ ;        11:  $A_{11}$ ;          12:  $A_{12}$ ;
end;
```

Vamos apresentar abaixo o esquema de um algoritmo que efetua a tradução de uma tabela por este método. Ele recebe como entrada uma tabela de decisão com condições, $C_1, \dots, C_m$ e regras $R_1, \dots, R_n$ e produz como saída a tabela traduzida.

Algoritmo S: Conversão por "Switch method".

Importação: Tabela de Decisão T .

Saída: Comandos que executam a tabela de decisão.

início

```

1.   para cada linha de condição  $C_i, 1 \leq i \leq m$ , faça
2.        $C := \emptyset; n_i := 0;$ 
3.       para cada  $C_{ij}, 1 \leq j \leq n$ , faça
4.           se  $C_{ij} \notin C$  então
5.               incrementa  $n_i;$     $c_{in_i} := C_{ij};$ 
6.               se  $C = \emptyset$  então
7.                   Produz Comando “if  $C_i \dots$  then  $I_i := 0$ ”
8.               senão
9.                   Produz Comando “else if  $C_i \dots$  then  $I_i := n_i - 1$ ”
10.           $C := C \cup \{c_{in_i}\};$ 
11.      fim para
12.  fim para
13.  Grava a expressão que calcula o valor de  $I$ ;
14.  para cada  $1 \leq M \leq n_1 \cdot n_2 \cdot \dots \cdot n_m = n$  faça
15.      determine  $I_1, I_2, \dots, I_m$ , tais que
           $(n_2 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot \dots \cdot n_m) \cdot I_2 +$ 
           $+ \dots + (n_m) \cdot I_{m-1} + I_m + 1 = M$ 
16.      determine a regra  $k$  tal que  $C_{ik} = c_{i, I_i + 1}, 1 \leq i \leq m$ 
17.      Grava o comando que executa a ação da regra  $R_k$ .
18.  fim para
fim
```

O algoritmo calcula os conjuntos:

$$\begin{aligned}
 C_1 &= \{c_{11}, c_{12}, \dots, c_{1n_1}\} \\
 C_2 &= \{c_{21}, c_{22}, \dots, c_{2n_2}\} \\
 C_3 &= \{c_{31}, c_{32}, \dots, c_{3n_3}\} \\
 &\vdots \\
 C_m &= \{c_{m1}, c_{m2}, \dots, c_{m n_m}\}.
 \end{aligned}$$

Para esta construção são utilizados os comandos das linhas 1 a 12. Para cada linha de condição, utiliza um conjunto de trabalho C , para saber quais entradas de condição são distintas das que já ocorreram. Nesta parte também são produzidos os comando “if ... then ...” para as variáveis internas do código produzido. Na linha 13 é produzida a expressão que calcula o número da regra a ser selecionada. Estamos adotando uma ordenação de tal modo que o índice de uma regra é calculado pela fórmula 2.1.

Existe uma outra ordenação possível. No exemplo apresentado, vimos que a primeira linha está ordenada e dentro desta a segunda está ordenada. A terceira linha, por sua vez, está ordenada dentro da segunda. Se colocarmos a ordenação numa hierarquia exatamente inversa, o índice seria calculado pela fórmula totalmente similar:

$$I = I_1 + (n_1) \cdot I_2 + (n_1 \cdot n_2) \cdot I_3 + \dots + (n_1 \cdot \dots \cdot n_{m-1}) \cdot I_m + 1.$$

As linhas de 14 a 18 produzem os comandos que executam a regra selecionada pelo índice I calculado. Se existe a garantia de uma ordenação adequada das regras, como a do exemplo,

então pode-se imediatamente executar a i -ésima regra. Os comandos das linhas 15 e 16 superam o problema da ordenação, encontrando na tabela a regra correspondente ao índice i . O comando da linha 15 poderia ser substituído por uma sequência mais simples de m “loops” encaixados, com o i -ésimo deles variando de 1 a n_i , mas teria o inconveniente de poder solucionar somente tabelas com exatamente m linhas de condição. Outra forma para isto é começar com todas as variáveis $I_1, I_2, \dots, I_m$ iguais a zero e, a cada novo valor de M , acrescenta 1 ao valor de I_m . Se $I_m > n_m$ então produz um “carry” para I_{m-1} , que então é incrementado também de 1 e I_m recebe o valor zero. Igualmente, se $I_{m-1} > n_{m-1}$, produz um “carry” para a variável I_{m-2} e assim sucessivamente. O limite superior para M garante que o processo pára em I_1 .

Relembrando o comando da linha 15, ligeiramente modificado, temos

“determine $I_1, \dots, I_m$, tais que

$$(n_2 \cdots n_m) \cdot I_1 + (n_3 \cdots n_m) \cdot I_2 + \cdots + (n_m) \cdot I_{m-1} + I_m = L”.$$

As restrições para as variáveis envolvidas na equação são:

- $0 \leq L \leq n_1 \cdot n_2 \cdots n_m - 1, \quad L = M - 1.$
- $0 \leq I_i \leq n_i - 1, 1 \leq i \leq m.$

É preciso encontrar a solução para as variáveis que deve ser única.

Algoritmo E: Calcula a solução da nova linha 15 do algoritmo S.

Importação: $L, n_1, \dots, n_m$.

Exportação: $I_1, \dots, I_m$.

início

Sejam $L_1 = L$ e $R_1 = 0$;

para cada i em 1 .. $m - 1$ faça

$I_i := L_i \text{ div } (n_{i+1} \cdot n_{i+2} \cdots n_m);$

$R_{i+1} := R_i + I_i \cdot (n_{i+1} \cdot n_{i+2} \cdots n_m);$

$L_{i+1} := L_i \text{ mod } (n_{i+1} \cdot n_{i+2} \cdots n_m);$

fim para

$I_m := L_m;$

fim

Claramente o algoritmo E termina, pois o único “loop” que possui é executado $m - 1$ vezes. Vamos demonstrar a seguir que o algoritmo E está correto.

Lema 14 $L = R_i + L_i$ é um invariante do “loop”.

Dem.: Por indução no número de iterações i .

Base da indução: $i=1$. Por inicialização das variáveis, $L_1 = L$ e $R_1 = 0$. Portanto, $L = L+0 = L$.

Suponha que a tese seja válida até a i -ésima iteração com $R_i + L_i = L$, Vamos mostrar que vale também para a $(i+1)$ -ésima iteração.

$$R_{i+1} + L_{i+1} = R_i + I_i \cdot (n_{i+1} \cdot \dots \cdot n_m) + L_i \bmod (n_{i+1} \cdot \dots \cdot n_m)$$

Como $I_i = L_i \operatorname{div} (n_{i+1} \cdot \dots \cdot n_m)$,

$$R_{i+1} + L_{i+1} = R_i + (L_i \operatorname{div} (n_{i+1} \cdot \dots \cdot n_m)) \cdot (n_{i+1} \cdot \dots \cdot n_m) + L_i \bmod (n_{i+1} \cdot \dots \cdot n_m)$$

Como $(L_i \operatorname{div} (n_{i+1} \cdot \dots \cdot n_m)) \cdot (n_{i+1} \cdot \dots \cdot n_m) + L_i \bmod (n_{i+1} \cdot \dots \cdot n_m) = L_i$,

$$R_{i+1} + L_{i+1} = R_i + L_i \\ = L, \text{ pela hipótese de indução. } \square$$

Lema 15 *O algoritmo E calcula*

$$R_{i+1} = (n_2 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot \dots \cdot n_m) \cdot I_2 + \dots + (n_{i+1} \cdot \dots \cdot n_m) \cdot I_i, \text{ para } 1 \leq i < m.$$

Dem.: Por indução em i .

Base da indução: $i = 2$. Pela fórmula, $R_2 = (n_2 \cdot \dots \cdot n_m) \cdot I_1$. Por outro lado, o algoritmo, calcula $R_2 = R_1 + (n_2 \cdot \dots \cdot n_m) \cdot I_1$. Como R_1 foi inicializado com zero, então o algoritmo calcula corretamente o valor de R_2 .

Suponha que o algoritmo calcula corretamente a expressão acima até R_i . Vamos calcular para R_{i+1} .

$R_{i+1} = R_i + I_i \cdot (n_{i+1} \cdot \dots \cdot n_m)$ Aplicando a hipótese indutiva em R_i , obtemos

$$R_{i+1} = (n_2 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot \dots \cdot n_m) \cdot I_2 + \dots + (n_i \cdot \dots \cdot n_m) \cdot I_{i-1} + (n_{i+1} \cdot \dots \cdot n_m) \cdot I_i. \square$$

Teorema 16 *O algoritmo E determina $I_1, \dots, I_m$ tais que*

$$(n_2 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot \dots \cdot n_m) \cdot I_2 + \dots + (n_m) \cdot I_{m-1} + I_m = L,$$

$$0 \leq L \leq n_1 \cdot n_2 \cdot n_m - 1 \text{ e } 0 \leq I_i \leq n_i - 1, 1 \leq i \leq m.$$

Dem.:

Pelo lema 14, $L = R_i + L_i$ é um invariante do “loop”. Assim, na última iteração, obtemos, $L = R_m + L_m$. Como no final o algoritmo faz $I_m = L_m$, $L = R_m + I_m$.

Pelo lema 15, $R_m = (n_2 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot \dots \cdot n_m) \cdot I_2 + \dots + (n_m) \cdot I_{m-1}$.

Portanto, $L = (n_2 \cdot \dots \cdot n_m) \cdot I_1 + (n_3 \cdot \dots \cdot n_m) \cdot I_2 + \dots + (n_m) \cdot I_{m-1} + I_m. \square$

Acabamos de encontrar uma solução para o comando da linha 15 do algoritmo. Precisamos mostrar que não existe outra solução possível.

Teorema 17 *Seja $0 \leq L \leq n_1 \cdot n_2 \cdot n_m - 1$. Existe uma única solução para*

$$n_2 \cdot \dots \cdot n_m \cdot I_1 + n_3 \cdot \dots \cdot n_m \cdot I_2 + \dots + n_m \cdot I_{m-1} + I_m = L,$$

com $0 \leq I_i \leq n_i - 1, 1 \leq i \leq m$.

Dem.: Sejam $I_1, \dots, I_m$ e $I'_1, \dots, I'_m$ duas soluções. Logo $(n_2 \cdot \dots \cdot n_m) \cdot (I_1 - I'_1) + (n_3 \cdot \dots \cdot n_m) \cdot (I_2 - I'_2) + \dots + (n_m) \cdot (I_{m-1} - I'_{m-1}) + (I_m - I'_m) = 0$.

Vamos mostrar que $I_1 = I'_1$. Suponhamos por contradição que

$$|I_m - I'_m| \geq 1. \text{ Logo } |(n_2 \cdot \dots \cdot n_m) \cdot (I_1 - I'_1)| \geq n_2 \cdot \dots \cdot n_m.$$

Por outro lado, a seguinte desigualdade é válida:

$$\begin{aligned} & |(n_3 \cdots n_m) \cdot (I_2 - I'_2) + (n_4 \cdots n_m) \cdot (I_3 - I'_3) + \cdots + (I_m - I'_m)| \leq \\ & |(n_3 \cdots n_m) \cdot (I_2 - I'_2)| + |(n_4 \cdots n_m) \cdot (I_3 - I'_3)| + \cdots + |(I_m - I'_m)| \\ & \text{Como } 0 \leq I_j, I'_j \leq n_j - 1, \text{ então } |I_j - I'_j| \leq n_j - 1. \text{ Logo,} \\ & |(n_3 \cdots n_m) \cdot (I_2 - I'_2)| + |(n_4 \cdots n_m) \cdot (I_3 - I'_3)| + \cdots + |(I_m - I'_m)| \leq \\ & (n_3 \cdots n_m) \cdot (n_2 - 1) + (n_4 \cdots n_m) \cdot (n_3 - 1) + \cdots + (n_m - 1) = \\ & n_2 \cdot n_3 \cdots n_m - 1. \end{aligned}$$

Portanto, $|(n_2 \cdots n_m) \cdot (I_1 - I'_1)| > |(n_3 \cdots n_m) \cdot (I_2 - I'_2) + (n_4 \cdots n_m) \cdot (I_3 - I'_3) + \cdots + (I_m - I'_m)|$. Mas isto é uma contradição, pois deveria prevalecer a igualdade, em função de duas soluções para a equação. Este resultado estabelece que $I_1 = I'_1$. Considerando este fato, repete-se o processo, para provar que $I_2 = I'_2$ e assim sucessivamente para $I_3 = I'_3, \dots, I_m = I'_m$. $\square$

Na implementação do algoritmo E , a variável L_i não precisa ser um vetor, basta reutilizar a mesma variável. A variável R_i é desnecessária; sua utilização se prestou unicamente para demonstração dos teoremas.

O comando da linha 16 do algoritmo consiste em buscar a regra correspondente a M -ésima regra dentro da ordenação esperada. Determinada a solução $I_1, \dots, I_m$, a regra R_k a ser executada é tal que para toda linha de condição $1 \leq i \leq m$, $C_{ik} = c_{i, I_i+1}$. A exigência de que a tabela deve ser completa garante que tal k sempre existe. No caso da tabela ser incompleta e nenhuma regra satisfazer os requisitos, deveria ser acionado o tratamento convencionado para a regra "else". Esta busca pode ser feita trivialmente, regra por regra.

A exigência de que as entradas de condição devem contemplar todos os valores possíveis da condição, tem a ver com o valor indevido que será atribuído à respectiva variável auxiliar. Isto pode ser superado com poucas alterações na atribuição de valores à variável auxiliar, fazendo com que, por exemplo, ela receba um valor maior que n .

Veinott [Ve 66b] apontou uma solução para o problema da sua proposta inicial com relação à obrigatoriedade da tabela ser completa. Ele propôs a manutenção de um vetor no código gerado, onde seria colocada a lista de todas as regras presentes. Assim, a cada índice calculado, verifica-se a presença da regra correspondente no vetor. A tabela fica mais simples se regras semanticamente impossíveis estão ausentes, mas a solução acima tem o inconveniente de o número das regras que ficam no vetor ter que ser aquele correspondente ao da tabela completa.

Complexidade do "switch method"

A tarefa executada pelo algoritmo E é $O(m)$. As linhas 1 a 12 consomem tempo da ordem de $O(m \cdot n)$. Como a tabela é completa $n = n_1 \cdot n_2 \cdots n_m$. Assim, o loop iniciado pela linha 14 é executado n vezes. A linha 16 consome tempo $O(m \cdot n)$. Logo, a complexidade total do algoritmo S é $O(m \cdot n^2)$. Quando o número de condições é grande, a simples produção da tabela completa como entrada para o algoritmo é muito custosa e neste sentido a complexidade acima pode ser enganosa.

A execução do código gerado pelo algoritmo produz sempre o mesmo número de comparações, independentemente da transação, para a determinação do índice da regra a ser executada. A partir daí, a localização das ações é imediata no uso do comando "goto" computado da linguagem FORTRAN, mas não o é no caso do comando "case" da Pascal.

2.2 Método de Teste Seqüencial

Este é o método mais imediato para a tradução de tabela de decisão. É considerado como um caso do “Test and branch method”, correspondendo à interpretação proposta por V. W. Setzer (vide 1.3.1). A sua simplicidade pode ser vista no exemplo seguinte.

	R ₁	R ₂	R ₃	R ₄	ELSE
X = Y	Y	N	N	Y	
Z = 0	N	Y	Y	N	
W ≥ 1	Y	N	Y	N	
A ₁	1	1		2	
A ₂	2		1	1	
Erro					1

O código produzido por este método teria o conteúdo análogo ao mostrado a seguir.

```

if ((X = Y) and (not(Z = 0)) and (W ≥ 1)) {A1; A2;}
else if ((not(X = Y)) and (Z = 0) and (not(W ≥ 1))) {A1;}
else if ((not(X = Y)) and (Z = 0) and (W ≥ 1)) {A2;}
else if ((X = Y) and (not(Z = 0)) and (not(W ≥ 1))) {A2; A1;}
else {Erro;}

```

A idéia é gerar um comando “if” para cada regra. O tradutor é extremamente simples e pode ser facilmente entendido pelo esquema de conversão mostrado na figura 2.1, onde supomos que um comando do tipo

“if C₁ and C₂ ...”,

é implementado com um desvio para a parte “else” no caso de C₁ ser “falso” independentemente da avaliação da condição C₂. Se esta premissa não é válida, o esquema deve ser alterado para que a representação seja correta.

Para seleção da i-ésima regra, o método produz os testes para todas as regras R₁, R₂, ..., R_{i-1}. Como em cada teste o método produz m comparações em cada regra, o pior caso de entrada produzirá a execução de m · n comparações para a seleção da regra. Assim, o código obtido por este método não é bom com relação à eficiência. Por outro lado, ele admite tabelas mistas, sem necessidade de conversão para tabelas limitadas.

2.3 Método de Máscara

De um modo geral, os algoritmos de tradução de tabelas de decisão mantêm uma imagem matricial da tabela que está sendo traduzida, durante a conversão, mas este método é o único que mantém esta imagem no programa gerado. Disto deriva o seu nome. Ele mantém uma máscara da tabela no código objeto. Existem diversas variantes deste método. Vamos apresentar algumas delas, sem nos preocupar com a enumeração exhaustiva de todos os algoritmos existentes e suas variações.

Figura 2.1: Esquema do Método Sequencial

2.3.1 Máscara de Regra

O primeiro artigo sobre este método [Ki 65] foi publicado em 1965, mas provavelmente é a forma mais antiga utilizada por processadores de tabelas de decisão, pois o autor descreve uma técnica usada em 1958. O método ("Rule Mask") trata de tabelas de decisão limitadas e consiste na utilização de três elementos básicos para uma tabela com m condições e n regras:

- O Vetor de estados S com m elementos, um para cada condição.

$$S = (a_1, a_2, \dots, a_m), \text{ com } a_i = \begin{cases} 1, & \text{se } C_i = \text{"Y"} \text{ (true)} \\ 0, & \text{se } C_i = \text{"N"} \text{ (false)} \end{cases}$$

Este vetor é mantido dinamicamente pelo programa objeto para cada transação.

- Uma matriz de máscara ("masking matrix") $M_{m \times n}$, de bits.

$$M_{ij} = \begin{cases} 1 & \text{se } C_{ij} = \text{"Y"} \text{ ou "N"} \\ 0 & \text{caso contrário} \end{cases}$$

- Uma matriz Tabela ("Table matrix") $T_{m \times n}$, de bits.

$$T_{ij} = \begin{cases} 1 & \text{se } C_{ij} = \text{"Y"} \\ 0 & \text{caso contrário} \end{cases}$$

As matrizes M e T são construídas em tempo de tradução, armazenadas no programa objeto e manipuladas durante a execução.

O funcionamento é extremamente simples. Faz-se uma operação “and” ($\cdot$) elemento a elemento (bit a bit) de S com a coluna j da matriz M , $1 \leq j \leq n$. Se o resultado desta operação é igual à j -ésima coluna de T então a regra R_j correspondente é selecionada. Vamos denotar por S' o vetor que é o resultado da operação “and” entre S e uma coluna j arbitrária da matriz M , ou seja,

$$S'_i := S_i \cdot M_{ij}, 1 \leq i \leq m.$$

Se $S'_i = T_{ij}$, $1 \leq i \leq m$, então seleciona a regra R_j . O programa objeto deve repetir esta operação até que uma regra seja selecionada ou até que se acabem todas as colunas de M . Neste último caso, o índice $j > n$, e isto é a indicação de que nenhuma regra foi selecionada, devendo desviar-se para a “else rule” se esta for a convenção adotada.

Pode-se ver que a matriz de máscara M é desnecessária, se todas as entradas de condição da tabela são relevantes (“Y” e “N”), bastando comparar S com as colunas de T . Logicamente a matriz “M” é introduzida para superar o problema das entradas irrelevantes, pois nas posições correspondentes das matrizes M e T temos o valor zero.

No exemplo abaixo, usaremos uma matriz $S'_m \times n$ para facilitar o entendimento, mas precisamos de um vetor de apenas m elementos. Retomemos a seguinte tabela, acompanhada da matriz de máscara M e também da matriz de tabela T .

	R ₁	R ₂	R ₃
NotaProva ≥ 5	Y	N	Y
NotaTrabalho ≥ 5	Y	–	N
Aprovado	1		
Refaz Trabalho			1
Reprovado		1	

$$M = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Suponhamos que a nota da prova seja 6 e a nota do trabalho 4. Claramente, o vetor de estados é $S = (1, 0)$; fazendo a operação “and” entre S e as colunas de M , obtemos

$$S' = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

As terceiras colunas de S' e T são iguais e assim, a regra R_3 é selecionada, como seria esperado.

Na verdade, cada coluna de S' é calculada e comparada com a coluna correspondente de T . Se elas são diferentes, repete-se o processo para a próxima coluna. No momento que forem iguais, a regra correspondente a esta coluna é selecionada e as demais colunas não precisam ser verificadas, desde que não se admita ambigüidades.

Para mostrar que o método está correto, precisamos mostrar que dada uma tabela de decisão limitada e um vetor de estados S , o método seleciona corretamente uma regra. O teorema seguinte resolve o problema.

Teorema 18 *Sejam T uma tabela de decisão limitada com condições $C_1, \dots, C_m$, regras $R_1, \dots, R_n$ e $S = (a_1, \dots, a_m)$ um vetor de estados para uma certa transação. $S_i \cdot M_{ij} = T_{ij}$, para todo $1 \leq i \leq m$ se e somente se a regra R_j é selecionável.*

Dem.:

$\Rightarrow$. Vamos considerar separadamente todos os valores possíveis para C_{ij} .

- caso 1. $C_{ij} = \text{"Y"}$. Por construção, $M_{ij} = 1$ e $T_{ij} = 1$. Como $a_i \cdot M_{ij} = T_{ij}$ então $a_i = 1$. Portanto, $V(C_{ij}) = 1$.
- caso 2. $C_{ij} = \text{"N"}$. Por construção, $M_{ij} = 1$ e $T_{ij} = 0$. Como $a_i \cdot M_{ij} = T_{ij}$ então $a_i = 0$. Portanto, $V(C_{ij}) = 1$.
- caso 3. $C_{ij} = \text{"-"}$. Por construção, $M_{ij} = 0$ e $T_{ij} = 0$. Como $a_i \cdot M_{ij} = T_{ij}$ então $a_i = 0$ ou $a_i = 1$. Portanto, $V(C_{ij}) = 1$.
- caso 4. $C_{ij} = \text{"\$"}$. Por construção, $M_{ij} = 0$. Por definição de "\$", $a_i = 1$. Portanto, $V(C_{ij}) = 1$.
- caso 5. $C_{ij} = \text{"*"}$. Por construção, $M_{ij} = 0$. Por definição de "*", $a_i = 0$. Portanto, $V(C_{ij}) = 1$.

$\Leftarrow$. Por hipótese, a regra R_j é selecionável. Logo, a função-and B_j correspondente tem o seu valor

$$V(B_j) = (V(C_{1j}) \cdot V(C_{2j}) \cdots V(C_{mj})) = 1.$$

Assim, $V(C_{ij}) = 1$, para todo $1 \leq i \leq m$. Verificando a tabela de cálculo de $V(C_{ij})$, observamos que existem 5 possíveis situações para que $V(C_{ij}) = 1$.

- caso 1. $C_{ij} = \text{"Y"}$ e $a_i = 1$. Por construção, $M_{ij} = 1$ e $T_{ij} = 1$. Claramente $a_i \cdot M_{ij} = T_{ij}$.
- caso 2. $C_{ij} = \text{"\$"}$ e $a_i = 1$. Por construção, $M_{ij} = 0$ e $T_{ij} = 0$. Claramente $a_i \cdot M_{ij} = T_{ij}$.
- caso 3. $C_{ij} = \text{"N"}$ e $a_i = 0$. Por construção, $M_{ij} = 1$ e $T_{ij} = 0$. Claramente $a_i \cdot M_{ij} = T_{ij}$.
- caso 4. $C_{ij} = \text{"*"}$ e $a_i = 0$. Por construção, $M_{ij} = 0$ e $T_{ij} = 0$. Claramente $a_i \cdot M_{ij} = T_{ij}$.
- caso 5. $C_{ij} = \text{"-"}$. Por construção, $M_{ij} = 0$ e $T_{ij} = 0$. Claramente $a_i \cdot M_{ij} = T_{ij}$. $\square$

Este algoritmo é fácil de implementar. A sua característica de armazenar "bits", torna-o mais adequado para processadores que não traduzem tabelas para linguagem de alto nível, pois o código produzido contém elementos alheios ao que foi especificado e pouco compreensível.

Em tempo de execução, este método exige que todas as condições sejam avaliadas para a construção do vetor de estado S . Além disso, o vetor de estado é submetido à operação "and" com cada coluna da matriz M e comparada com a respectiva coluna da matriz T . Assim, o tempo gasto t , para a execução do código traduzido da tabela, para uma transação, é

$$t = t_1 + t_2 + \cdots + t_m + m \cdot n \cdot t_a + m \cdot n \cdot t_b,$$

onde

$t_i, 1 \leq i \leq m$ é o tempo para avaliação da condição C_i , isto é, para calcular o valor de a_i .

t_a é o tempo para calcular $a_i \cdot M_{ij}$.

t_b é o tempo para comparar $a_i \cdot M_{ij}$ com T_{ij} .

É claro que $m \cdot t_a$ pode tornar-se simplesmente t_a , se admitirmos que a operação “and” pode ser feita entre colunas inteiras e não entre os elementos das colunas. A mesma observação é válida para $m \cdot t_b$. Se supusermos que as regras são igualmente prováveis e que após selecionada uma regra, outras não mais são pesquisadas, o tempo médio de execução de uma tabela de decisão para a seleção da regra é

$$t = t_1 + t_2 + \cdots + t_m + \frac{m \cdot n}{2} \cdot (t_a + t_b).$$

Este tempo de execução contempla apenas o tempo gasto para a seleção da regra, não incluindo o tempo de execução de ações selecionadas.

2.3.2 Máscara de Regra com Interrupção

É uma variante do método de Máscara de Regra que tenta melhorar o código do programa gerado, quanto à sua eficiência no tempo de execução. Uma forma imediata de conseguir uma melhoria no resultado final seria ordenar as regras da tabela de decisão em ordem decrescente de frequência com que elas ocorrem. Existem outras estratégias para melhoria da eficiência [Ki 66] que procura interromper a avaliação das condições assim que seja possível avaliar uma regra, prosseguindo a avaliação das outras condições à medida em que forem necessárias para as regras, numa sequência adequada para a obtenção de código melhorado.

Representação

Considere o seguinte exemplo, com uma tabela de decisão e os esquemas de seleção de regra pelo método de Máscara de Regra e pelo método com interrupção.

	R ₁	R ₂
C ₁	Y	N
C ₂	-	Y
A ₄	1	
A ₂		1

A interpretação do esquema é simples. O losangos indicam as operações “and” entre o vetor de estado S e a j -ésima coluna da matriz M e também a comparação deste resultado com a j -ésima coluna da matriz T . Nos retângulos à esquerda da vertical formada pelos losangos, estão as operações de avaliação das condições, ou seja o cálculo do valor de a_i correspondente à condição C_i . Nos retângulos à direita da vertical formada pelos losangos estão as chamadas de execução de uma regra, indicando que a comparação efetuada no losango à esquerda teve resultado verdadeiro.

Seleção de uma Regra

Como podemos observar na figura anterior, o método com interrupção aproveita-se do fato de que existe uma entrada “-” na regra R_1 para melhorar o desempenho do código gerado. Em cada transação executada, o método de máscara básico, à esquerda da figura, sempre faz a avaliação das duas condições, enquanto o método com interrupção, à direita na figura, avalia somente uma única condição se a regra R_1 é a escolhida. Se a regra escolhida é a R_2 , os dois métodos executam exatamente o mesmo número de operações. Se o universo de transações é tal que a primeira regra é muito mais freqüente que a segunda, então o método com interrupção pode ser mais vantajoso.

É claro que para esta melhoria no programa objeto, o programa tradutor precisa de trabalho adicional, mas como se supõe que o programa gerado será executado inúmeras vezes, esta relação de custo e benefício é favorável ao método com interrupção, ainda mais que as complicações não são numerosas.

Para entender que a execução do algoritmo do método de máscara com interrupção seleciona corretamente uma regra, basta observar que a sua execução é idêntica ao método de máscara básico, com exceção da dispensa de algumas avaliações irrelevantes. Este algoritmo propicia diversas estratégias conforme a escolha da condição a ser avaliada. King [Ki 66] propôs quatro estratégias.

Seja uma tabela de decisão T com condições $C_1, \dots, C_m$, regras $R_1, \dots, R_n$ com frequências relativas $f_1, \dots, f_n$, indicando que no universo de transações que está sendo tratado, ou numa

amostra dele, a regra R_i ocorre f_i vezes em relação ao número total de eventos. Seja $\bar{f}$, a frequência relativa com que nenhuma das regras é satisfeita. Se a tabela é completa, $\bar{f} = 0$. Sejam $t_1, \dots, t_m$, o tempo gasto para avaliar as condições $C_1, \dots, C_m$, respectivamente e s o tempo gasto para efetuar a operação “and” entre o vetor de estados S e uma coluna da matriz M e comparar o resultado com a coluna correspondente da matriz T . Estamos supondo que é possível fazer operações deste tipo entre colunas. Isto é simples se supusermos que as colunas são cadeias de “bits” em registradores ou em palavras únicas de memória. Chamaremos a frequência relativa simplesmente de frequência.

No método de máscara básico, obtemos para o tempo médio t , gasto para a seleção de uma regra para uma transação, a seguinte expressão.

$$\begin{aligned}
 t &= (t_1 + t_2 + \dots + t_m + s) \cdot f_1 + (t_1 + t_2 + \dots + t_m + 2s) \cdot f_2 + \\
 &\quad + \dots + (t_1 + t_2 + \dots + t_m + n \cdot s) \cdot f_n + \\
 &\quad (t_1 + t_2 + \dots + t_m + n \cdot s) \cdot \bar{f} \\
 &= (t_1 + t_2 + \dots + t_m) \cdot f_1 + (t_1 + t_2 + \dots + t_m) \cdot f_2 + \dots + \\
 &\quad (t_1 + t_2 + \dots + t_m) \cdot f_n + \\
 &\quad (t_1 + t_2 + \dots + t_m) \cdot \bar{f} + n \cdot s \cdot \bar{f} + s \cdot \sum_{i=1}^n i \cdot f_i \\
 &= t' \cdot A \cdot f + \bar{f} \cdot \left(\sum_{i=1}^m t_i + n \cdot s \right) + s \cdot \sum_{i=1}^n i \cdot f_i,
 \end{aligned}$$

onde t' , A e f são as matrizes seguintes:

$$t'_{1 \times m} = \begin{bmatrix} t_1 & t_2 & \dots & t_m \end{bmatrix} \quad A_{m \times n} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix} \quad f_{n \times 1} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

A vantagem desta notação matricial é que esta fórmula se adapta a medir o tempo gasto, apenas colocando-se “zeros” na matriz A , nas posições A_{ij} em que a condição C_i não foi avaliada para a verificação da regra R_j . Além disso, é preciso colocar no termo $\sum i \cdot f_i$, a ordem de teste das regras, já que não será obrigatoriamente na sequência $R_1, R_2, \dots, R_n$. Assim,

$$t = t' \cdot A \cdot f + \bar{f} \cdot \left(\sum_{i=1}^m t_i + n \cdot s \right) + s \cdot \sum_{i=1}^n i \cdot f_{j_i},$$

onde $j_1, j_2, \dots, j_n$ é a ordem em que as regras são testadas, representa o tempo médio gasto para seleção de uma regra da tabela de decisão para uma transação do universo em questão.

É claro que a matriz $A_{m \times n}$ representa o particular critério de escolha das condições que são avaliadas antes das regras serem testadas, ou seja, a primeira coluna da matriz A contém o número 1 somente nas linhas correspondentes às condições que já foram avaliadas quando o teste para a primeira regra foi efetuado. Claramente, se $M_{ij} = 1$, então $A_{ij} = 1$, mas nada se pode adiantar, a priori, sobre o valor de A_{ij} , se $M_{ij} = 0$. Vejamos o seguinte exemplo:

	R ₁	R ₂	R ₃	R ₄	R ₅
C ₁	Y	Y	N	N	Y
C ₂	N	–	Y	N	Y
C ₃	Y	N	–	–	Y

Vamos fazer dois esquemas para duas possíveis seqüências de avaliação de condições e testes das regras na figura 2.2.

Figura 2.2: Máscara de Regra

As matrizes A_e e A_d , correspondentes aos esquemas da esquerda e direita respectivamente da figura 2.2, seriam as seguintes.

$$A_e = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$A_d = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Podemos ver que existem $A_{ij} = 1$ com o correspondente $M_{ij} = 0$. Nem sempre é possível obter $A_{ij} = 0$ para todo $M_{ij} = 0$. Tanto no esquema da esquerda como da direita da figura anterior, após a avaliação das três condições, as regras são testadas dentro de uma ordem qualquer. Esta ordem também é um dos aspectos considerados no método de máscara com interrupção.

Na expressão,

$$t = t' \cdot A \cdot f + \bar{f} \cdot \left(\sum_{i=1}^m t_i + n \cdot s \right) + s \cdot \sum_{i=1}^n i \cdot f_{j_i} ,$$

o termo $\bar{f} \cdot (\sum_{i=1}^m t_i + n \cdot s)$ independe da particular matriz A e da sequência de avaliação das regras. Logo, o problema de conseguir o código ótimo consiste em minimizar a expressão

$$t = t' \cdot A \cdot f + s \cdot \sum_{i=1}^n i \cdot f_{j_i} ,$$

onde as variáveis são a matriz A e a ordenação $j_1, j_2, \dots, j_n$. Como disse King [Ki 66], “não é claro que tal problema seja solucionável a menos para pequenas tabelas onde é possível a enumeração de todos os casos possíveis”. Como a matriz A e a sequência $j_1, j_2, \dots, j_n$ são dependentes da particular ordem em que as condições são avaliadas, temos $m!$ possíveis casos que é o número de permutações na ordem de avaliação das condições [Sc-Se 76].

Estratégias para o método com interrupção

O método de máscara com interrupção segue então algumas estratégias na ordem de avaliação das condições, que tentam obter um bom resultado no código gerado, visto que parece inviável uma técnica ótima com esta finalidade. Vamos apresentar as quatro estratégias propostas por King [Ki 66], embora diversas outras variações possam ser apresentadas. Todas elas assumem que as regras testadas consecutivamente são feitas em ordem decrescente de frequência com que ocorrem. Nenhuma delas garante que sejam ótimas, mas resultados empíricos mostram uma melhora em relação ao método básico.

Seja $p_i = \sum_{j=1}^n f_j \cdot M_{ij}$, para cada condição C_i , $1 \leq i \leq m$.

	R ₁	R ₂
C ₁	Y	Y
C ₂	N	–

Na tabela mostrada, suponha que

$$\begin{aligned} f_1 &= 0,4 \\ f_2 &= 0,2 \\ f_3 &= 0,4. \end{aligned}$$

Como $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$, os valores de p_1 e p_2 são 0,6 e 0,4, respectivamente.

Pela fórmula, podemos observar que p_i é calculado somando-se a frequência relativa das regras para as quais a condição C_i é relevante. A sua interpretação é de que p_i mede a frequência com que ocorrem as regras em que a condição deve ser necessariamente avaliada. Em outras palavras, p_i é a frequência com que C_i deve ser obrigatoriamente avaliada. Esta grandeza é chamada *probabilidade de relevância*.

A estratégia “A” seleciona as condições na ordem decrescente de probabilidade de relevância p_i . Ou seja, as condições que mais freqüentemente são necessárias de serem avaliadas, o são em primeiro lugar. A observação importante é que esta ordem de prioridade não leva em conta o tempo de avaliação de cada condição. Poderíamos imaginar uma situação em que a condição mais prioritária nesta estratégia consome pouco tempo e a de prioridade mais baixa consome um tempo extremamente alto, de tal sorte que o fator “tempo de avaliação da condição” deva ser considerado.

	R ₁	R ₂
C ₁	Y	N
C ₂	–	N
C ₃	Y	–

Consideremos como exemplo a tabela anterior, com tempos de avaliação das condições e frequências relativas

$$\begin{array}{rcl}
 t_1 & = & 1,0 \\
 t_2 & = & 1,0 \\
 t_3 & = & 10,0 \\
 s & = & 1,0.
 \end{array}
 \qquad
 \begin{array}{rcl}
 f_1 & = & 0,6 \\
 f_2 & = & 0,4 \\
 \bar{f} & = & 0
 \end{array}$$

Como $p_1 = 1,0$, $p_2 = 0,4$ e $p_3 = 0,6$, o método de escolha pela estratégia “A” avaliaria as condições C_1 , C_3 e C_2 , nesta ordem. Neste caso, o tempo médio

$$t = (t_1 + t_3 + s) \cdot f_1 + (t_1 + t_2 + t_3 + 2 \cdot s) \cdot f_2 = 12 \cdot 0,6 + 14 \cdot 0,4 = 12,8.$$

Se escolhermos na ordem C_1 , C_2 e C_3 , obteríamos o tempo

$$t = (t_1 + t_2 + s) \cdot f_2 + (t_1 + t_2 + t_3 + 2 \cdot s) \cdot f_1 = 3 \cdot 0,4 + 14 \cdot 0,6 = 9,6.$$

O exemplo mostra que às vezes é interessante considerar o tempo gasto na avaliação de cada condição. Assim, a estratégia “B” avalia as condições em ordem decrescente de p_i/t_i .

Já dissemos que uma maneira trivial de melhorar o método de máscara básico seria simplesmente ordenar decrescentemente as regras pelas suas frequências. A estratégia “C” testa as regras nesta ordem, avaliando as condições assim que se tornem necessárias.

É possível que a escolha da ordem de teste das regras da estratégia “C” não seja a melhor. A escolha de uma regra mais freqüente pode produzir a avaliação de uma condição altamente custosa numa regra menos freqüente, mas que tem entrada irrelevante na condição escolhida o que pode aumentar o tempo médio de execução do código obtido por esta estratégia.

	R ₁	R ₂
C ₁	Y	N
C ₂	N	–

Por exemplo, suponha que na tabela da figura precedente, $f_1 > f_2$. Assim, pela estratégia “C”, a regra R_1 seria testada antes da R_2 . Como a regra R_2 não necessita da avaliação da condição C_2 e se esta é muito custosa, é possível que a inversão da ordem seja uma prática interessante. Seja

$$d_j = \sum_{i=1}^m t_i \cdot M_{ij}, 1 \leq j \leq n.$$

O valor d_j é interpretado como sendo a soma dos tempos de avaliação das condições relevantes da regra R_j . Representando de forma matricial, $d = t \cdot M$. A estratégia “D” considera a seleção das regras em ordem decrescente de f_i/d_i .

Resumidamente, a estratégia “A” se baseia na hipótese de que é melhor avaliar as condições com maior probabilidade de relevância em primeiro lugar, ou seja, as condições mais prováveis de serem obrigatoriamente avaliadas têm prioridade sobre as outras menos prováveis. A estratégia “B” considera o argumento anterior, mas também leva em conta a possibilidade de ser melhor avaliar as condições com menor custo de avaliação, mesmo que esta possa ter menor probabilidade de relevância. A estratégia “C” considera que pode ser melhor avaliar as condições na ordem em que forem necessárias para o teste das regras em ordem decrescente de frequências. A estratégia “D” considera o argumento da estratégia “C”, mas leva em conta que pode ser melhor testar as regras cujas condições relevantes apresentam baixo custo de avaliação, mesmo que conflite com a ordem de frequência.

As estratégias “A” e “B” escolhem as condições a serem avaliadas e as regras são testadas à medida em que as condições já avaliadas sejam suficientes para a realização destes testes. Por outro lado, as estratégias “C” e “D” avaliam as condições com base na necessidade imposta pela ordem das regras, ou seja, as regras são escolhidas em primeiro lugar e as condições são avaliadas à medida em que são necessárias.

Vejamos um exemplo abstrato através da tabela da figura 2.3 para compararmos as diversas formas de tradução de uma tabela pelo método de máscara de regra básico e com interrupção proposta por King [Ki 66].

	R ₁	R ₂	R ₃	R ₄
C ₁	–	N	Y	Y
C ₂	Y	–	–	N
C ₃	N	–	Y	N
C ₄	N	Y	–	–

Figura 2.3: Tabela para exemplo

Vamos supor $s = 1$ e que as frequências relativas das regras e os tempos de avaliação das condições são

$$\begin{array}{ll} f_1 = 0,15 & t_1 = 5 \\ f_2 = 0,10 & t_2 = 3 \\ f_3 = 0,30 & t_3 = 2 \\ f_4 = 0,35 & t_4 = 1 \\ \bar{f} = 0,10. & \end{array}$$

Vamos chamar de t' a soma $t_1 + t_2 + t_3 + t_4 = 11$. Calculando os valores de p_i e p_i/t_i para cada linha da tabela, obtemos:

$$\begin{array}{ll} p_1 = 0,75 & p_1/t_1 = 0,15 \\ p_2 = 0,50 & p_2/t_2 = 0,16 \\ p_3 = 0,80 & p_3/t_3 = 0,40 \\ p_4 = 0,25 & p_4/t_4 = 0,25. \end{array}$$

Calculando os valores de d_i e f_i/d_i para cada regra da tabela temos:

$$\begin{array}{ll} d_1 = 6 & f_1/d_1 = 0,025 \\ d_2 = 6 & f_2/d_2 = 0,016 \\ d_3 = 7 & f_3/d_3 = 0,042 \\ d_4 = 10 & f_4/d_4 = 0,035. \end{array}$$

Na figura 2.4(a) está o esquema de representação do código obtido pela tradução da tabela através do método de máscara de regra básico, sem interrupção, que tem o seguinte tempo médio de execução:

$$\begin{aligned} t &= (t' + s) \cdot f_1 + (t' + 2 \cdot s) \cdot f_2 + (t' + 3 \cdot s) \cdot f_3 + \\ &\quad (t' + 4 \cdot s) \cdot f_4 + (t' + 4 \cdot s) \cdot \bar{f} \\ &= 12 \cdot 0,15 + 13 \cdot 0,1 + 14 \cdot 0,30 + 15 \cdot 0,35 + 15 \cdot 0,1 \\ &= 1,8 + 1,3 + 4,2 + 5,25 + 1,5 \\ &= 14,05. \end{aligned}$$

A figura 2.4(b) mostra o esquema de execução do código obtido pela tradução da tabela através do método de máscara de regra básico sem interrupção, testando as regras pela simples ordenação decrescente de suas frequências. O seu tempo médio de execução para o exemplo é

$$\begin{aligned} t &= (t' + s) \cdot f_4 + (t' + 2 \cdot s) \cdot f_3 + (t' + 3 \cdot s) \cdot f_1 + \\ &\quad (t' + 4 \cdot s) \cdot f_2 + (t' + 4 \cdot s) \cdot \bar{f} \\ &= 12 \cdot 0,35 + 13 \cdot 0,3 + 14 \cdot 0,1 + 15 \cdot 0,15 + 15 \cdot 0,1 \\ &= 4,2 + 3,9 + 1,4 + 2,25 + 1,5 \\ &= 13,25. \end{aligned}$$

Seguindo a estratégia “A” do método de máscara de regra com interrupção, que escolhe as linhas de condição por ordem decrescente de p_i , testando as regras possíveis com as avaliações

Figura 2.4: Máscara de regra básico e com ordenação pelas frequências

disponíveis, obtemos o código cuja execução pode ser representada como segue na figura 2.5(a). O seu tempo médio de execução para o exemplo é:

$$\begin{aligned}
 t &= (t_1 + t_3 + s) \cdot f_3 + (t_1 + t_2 + t_3 + 2 \cdot s) \cdot f_4 + \\
 &\quad (t_1 + t_2 + t_3 + t_4 + 3 \cdot s) \cdot f_1 + (t_1 + t_2 + t_3 + t_4 + 4 \cdot s) \cdot f_2 + \\
 &\quad (t_1 + t_2 + t_3 + t_4 + 4 \cdot s) \cdot \bar{f} \\
 &= 9 \cdot 0,3 + 12 \cdot 0,35 + 14 \cdot 0,15 + 15 \cdot 0,1 + 15 \cdot 0,1 \\
 &= 2,7 + 4,2 + 2,1 + 1,5 + 1,5 \\
 &= 12,0.
 \end{aligned}$$

A figura 2.5(b) mostra o esquema de execução do código obtido pela tradução da tabela, através do método de máscara de regra com interrupção, usando a estratégia “B”, que escolhe as condições a serem avaliadas por ordem decrescente de p_i/t_i , testando as regras assim que for

Figura 2.5: Estratégias “A” e “B”

possível. O seu tempo médio de execução no exemplo é:

$$\begin{aligned}
 t &= (t_2 + t_3 + t_4 + s) \cdot f_1 + (t_1 + t_2 + t_3 + t_4 + 2 \cdot s) \cdot f_4 + \\
 &\quad (t' + 3 \cdot s) \cdot f_3 + (t' + 4 \cdot s) \cdot f_2 + (t' + 4 \cdot s) \cdot \bar{f} \\
 &= 7 \cdot 0,15 + 13 \cdot 0,35 + 14 \cdot 0,30 + 15 \cdot 0,1 + 15 \cdot 0,1 \\
 &= 1,05 + 4,55 + 4,2 + 1,5 + 1,5 \\
 &= 12,8.
 \end{aligned}$$

Segue na figura 2.6(a) o esquema de execução do código produzido pelo método de máscara de regra com interrupção pela estratégia “C”, que adota o critério de avaliação das condições por ordem de necessidade para testes das regras em sequência decrescente de suas frequências

relativas f_i . O seu tempo médio de execução para o exemplo é:

$$\begin{aligned}
 t &= (t_1 + t_2 + t_3 + s) \cdot f_4 + (t_1 + t_2 + t_3 + 2 \cdot s) \cdot f_3 + \\
 &\quad (t' + 3 \cdot s) \cdot f_1 + (t' + 4 \cdot s) \cdot f_2 + (t' + 4 \cdot s) \cdot \bar{f} \\
 &= 11 \cdot 0,35 + 12 \cdot 0,3 + 14 \cdot 0,15 + 15 \cdot 0,1 + 15 \cdot 0,1 \\
 &= 3,85 + 3,6 + 2,1 + 1,5 + 1,5 \\
 &= 12,55.
 \end{aligned}$$

Figura 2.6: Estratégias “C” e “D”

Por último, a figura 2.6(b) mostra o esquema de execução do código produzido pelo método de máscara de regra com interrupção e estratégia “D” que adota o critério de avaliação das condições por ordem de necessidade para teste das regras em sequência decrescente de f_i/d_i .

Neste exemplo particular, o esquema de execução dos códigos obtidos pelas estratégias “D” e “A” são os mesmos, exceto pela troca da ordem de avaliação das condições C_1 e C_3 . Mas esta ordem é indiferente para efeito do cálculo do tempo médio de execução, em razão de serem

avaliações consecutivas, sem nenhum teste de regra entre elas. Assim, o tempo médio para a seleção de uma regra para uma transação, nesta estratégia, é igual ao calculado para a estratégia "A".

Este exemplo com os esquemas de execução dos códigos produzidos pelas quatro estratégias serve para facilitar o entendimento do método de máscara de regra com interrupção. É claro que esta técnica requer um trabalho maior durante a conversão, visto que ela determina qual das estratégias produz o melhor resultado a partir da tabela e informações sobre as frequências e o custo de avaliação de cada condição. Feito isto, converte a tabela pela estratégia escolhida. Basicamente, o algoritmo de conversão constrói a matriz A da fórmula que calcula o tempo médio de execução do programa para uma transação, para cada uma das estratégias, que pode ser feito sem dificuldades. Obtidas as quatro matrizes, obtém-se os tempos médios consumidos por elas, que comparadas, indicam qual o método que deve ser empregado para o problema particular.

A complexidade do código obtido para a seleção de regra é da mesma ordem para o método de máscara de regra básico e os obtidos com as estratégias de interrupção. As diferenças ficam por conta de constantes multiplicativas.

Para termos uma idéia de como o método seria implementado, vejamos esquematicamente como seriam os programas tradutores, para o método de máscara de regra.

Algoritmo: Conversão pelo método de máscara.

Importação: Tabela de decisão T com condições $C_1, \dots, C_m$, e regras $R_1, \dots, R_n$

Saída: Sequência de comandos que executam a tabela T .

início

1. Gera matrizes M e T
2. Gera comando que avalia $C_i, 1 \leq i \leq m$.
3. Gera comando que avalia a expressão $S \cdot M_j = T_j$,
desviando para regra R_j se for verdadeira.

fim

No algoritmo acima, M e T são as matrizes de máscara e de tabela, respectivamente como definido. Na linha 3, M_j é um vetor correspondente à j -ésima coluna da matriz M . Analogamente, T_j é o vetor correspondente à j -ésima coluna da matriz T .

O esquema acima é simples e de implementação imediata. Veja que o comando

"Gera comando que avalia $C_i, 1 \leq i \leq m$ ",

deve produzir código do tipo:

if C_i then $a_i := 1$ else $a_i := 0$;

para cada condição C_i que é um dado do programa tradutor, pois consta da tabela que é lida por ele.

Analogamente, o tradutor deve gerar o comando que avalia as expressões

$$S \cdot M_j = T_j$$

e desvia para o conjunto de ações especificadas na regra que for satisfeita, ou seja, deve gerar um comando do tipo:

if $(S \cdot M_j = T_j)$ **then** R_j .

No caso do exemplo R_j é uma ação do tipo procedimento em Pascal. É claro que se a linguagem objeto não permite uma expressão " $S \cdot M_j = T_j$ ", que envolve vetores de "bits", ela pode ser facilmente substituída pelo nome de uma função que recebe os objetos S , M_j e T_j como parâmetros e devolve um valor lógico. Tal função é trivialmente implementada.

Agora vamos descrever esquematicamente o tradutor para o método de máscara de regra com interrupção.

Algoritmo: Máscara com Interrupção.

Importação: Tabela de decisão T com condições $C_1, C_2, \dots, C_m$, regras $R_1, R_2, \dots, R_n$, frequências relativas $f_1, f_2, \dots, f_n$, $bar f$, custos de avaliação das condições, $t_1, t_2, \dots, t_m$ e o tempo gasto s para fazer uma operação do tipo " $(S \cdot M_j) = T_j$ ".

Saída: Comandos que executam T pelo método de máscara com interrupção.

início

1. Calcula $p_i, p_i/t_i, d_j$ e $f_i/d_j, 1 \leq i \leq m$ e $1 \leq j \leq n$.
2. Calcula tempo médio t_A para a estratégia "A"
3. Calcula tempo médio t_B para a estratégia "B"
4. Calcula tempo médio t_C para a estratégia "C"
5. Calcula tempo médio t_D para a estratégia "D"
6. Gera as matrizes M e T
7. **se** t_A é mínimo em $\{t_A, t_B, t_C, t_D\}$ **então**
8. traduz pela estratégia "A"
9. **senão se** t_B é mínimo em $\{t_A, t_B, t_C, t_D\}$ **então**
10. traduz pela estratégia "B"
11. **senão se** t_C é mínimo em $\{t_A, t_B, t_C, t_D\}$ **então**
12. traduz pela estratégia "C"
13. **senão se** t_D é mínimo em $\{t_A, t_B, t_C, t_D\}$ **então**
14. traduz pela estratégia "D".

fim

Analisando os comandos do algoritmo acima, vemos que a linha 1 pode ser executada facilmente pela definição dos termos que estão sendo calculados.

$$p_i = \sum_{j=1}^n f_j \cdot M_{ij}, 1 \leq i \leq m$$

$$d_j = \sum_{i=1}^m t_i \cdot M_{ij}, 1 \leq j \leq n.$$

O cálculo do tempo médio t_A da estratégia "A", pode ser obtido na linha 2 pelo seguinte sub-algoritmo.

Algoritmo: Calcula o Tempo médio da estratégia “A”.

Importação: Tabela de decisão T com condições $C_1, \dots, C_m$ e regras $R_1, \dots, R_n$, frequências relativas $f_1, \dots, f_n, \bar{f}$ e custos de avaliação das condições, $t_1, \dots, t_m$ o tempo gasto s para fazer uma operação do tipo “ $(S \cdot M_j) = T_j$ ”.

Exportação: Tempo médio t_A .

início

```

2.1       $T_0 := 0; C := \{C_1 \dots, C_m\}; \bar{C} := \emptyset; R := \{R_1, \dots, R_n\};$ 
2.2       $t_A = 0; i := 0; \bar{R} = \emptyset;$ 
2.3      enquanto  $R \neq \emptyset$  faça
2.4          se  $\bar{R} = \emptyset$  então (* Não existe regra testável *)
2.5              seja  $p_l = \max\{p_k : C_k \in C\};$ 
2.6               $T_i := T_i + t_l;$ 
2.7               $\bar{C} := \bar{C} \cup \{C_l\}; C := C - \{C_l\};$ 
2.8              seja  $\bar{R} \subseteq R : \forall R_k \in \bar{R}, \text{ se } C_{pk} = \text{“Y” ou “N”}$ 
                  então  $C_p \in \bar{C}, 1 \leq p \leq m;$ 
2.9               $R := R - \bar{R};$ 
2.10         fim-então
2.11         senão
2.12              $i := i + 1;$ 
2.13             seja  $f_{j_i} = \max\{f_k : R_k \in \bar{R}\};$ 
2.14              $\bar{R} := \bar{R} - \{R_{j_i}\};$ 
2.15              $T_i := T_{i-1} + s;$ 
2.16              $t_A := t_A + T_i \cdot f_{j_i};$ 
2.17         fim-senão
2.18     fim-enquanto
2.19      $t_A := t_A + T_i \cdot \bar{f};$ 
fim

```

A especificação do sub-algoritmo usa notação de conjuntos para facilitar o entendimento e pode ser implementado de diversas maneiras. [Ah-Ho-UI 74, Te 82] trazem estruturas e algoritmos eficientes para o tratamento de conjuntos. Uma das formas de representação é a árvore hierárquica que uma vez construída permite o acesso imediato ao elemento máximo do conjunto, além de possibilitar um armazenamento simples e compacto. Como a construção dos conjuntos é feita com todos os elementos disponíveis, existe técnica de construção de tal representação com complexidade linear com o número de elementos do conjunto. A remoção do elemento máximo com o rearranjo da árvore tem custo logarítmico. De qualquer forma, vamos adotar rotinas triviais de busca do elemento máximo, por varredura seqüencial, no vetor que representa o conjunto, para efeito da estimativa da complexidade do sub-algoritmo.

A fase de inicialização de conjuntos e variáveis consome tempo proporcional a $(m + n)$, para a construção dos conjuntos C e R .

O “loop” executa os comandos das linhas 2.5 a 2.9, m vezes, uma para cada condição e os comandos das linhas 2.12 a 2.16 n vezes, uma para cada regra. Cada execução da linha 2.5 extrai um elemento do conjunto C . Nesta escolha pode fazer no máximo m comparações. Cada execução das linhas 2.6 e 2.7 exige tempo constante. Especificamente a linha 2.7 altera o bit

correspondente ao conjunto sendo incluído e excluído. Cada execução da linha 2.8 percorre todas as regras em R com no máximo n regras e para cada uma delas, verifica cada entrada de condição. Portanto, o número de comparações não excede $(m \cdot n)$. A linha 2.13 efetua uma varredura sobre os elementos de R que tem no máximo n regras. As linhas 2.12, 2.14, 2.15 e 2.16 efetuam operações com tempo constante. Na estimativa de complexidade deste sub-algoritmo, supomos que a manipulação de conjuntos, além de tratar os bits do vetor que os representam, mantém um contador de elementos, de forma que as comparações feitas para o controle do “loop” sejam simples e feitas sobre um único elemento. Assim, a complexidade do sub-algoritmo é da ordem de

$$(m + n) + m \cdot (m + m \cdot n) + n \cdot n).$$

Fazendo $p = m \cdot n$ o tamanho da entrada e colocando a expressão em função de p , obtemos que a complexidade do sub-algoritmo é $O(p^2)$.

A discussão deste sub-algoritmo será informal. Como vimos, o tempo de execução de uma tabela, medida pelo código obtido pela estratégia “A” é dado por

$$t_A = t' \cdot A \cdot f + \bar{f} \cdot (\sum t_i + n \cdot s) + s \cdot \sum i \cdot f_{j_i},$$

que pode ser reescrito da seguinte forma

$$t_A = \sum_{i=1}^n (\sum_{C_p \in \bar{C}} t_p) \cdot f_{j_i} + s \cdot \sum_{i=1}^n i \cdot f_{j_i} + \bar{f} (\sum t_i + n \cdot s),$$

onde $\bar{C}$ é o conjunto das condições avaliadas até o i -ésimo teste que é feito contra a regra R_{j_i} . Reacertando os termos da expressão, obtemos

$$t_A = \sum_{i=1}^n (\sum_{C_p \in \bar{C}} t_p + s \cdot i) \cdot f_{j_i} + \bar{f} \cdot (\sum_{i=1}^n t_i + n \cdot s). \quad (2.2)$$

Teorema 19 *No sub-algoritmo, o termo*

$$T_i = \sum_{C_p \in \bar{C}} (t_p) + i \cdot s$$

é um invariante do “loop”.

Dem.: Por indução no número de iterações.

Para base, tomamos o instante da primeira entrada no loop, em que $T_0 = 0$, $\bar{C} = \emptyset$ e $i = 0$. Claramente,

$$\sum_{C_p \in \bar{C}} t_p + i \cdot s = 0,$$

pois $\bar{C} = \emptyset$ e $i = 0$. Isto está de acordo com o valor inicial de T_0 .

Considere uma iteração arbitrária do “loop” e suponha a tese válida para até a iteração anterior. Vamos calcular o valor de T_i da iteração corrente.

caso 1. $\bar{R} = \emptyset$. Como o valor de i fica inalterado, vamos chamar de T'_i o valor de T_i na iteração anterior. Pelo comando da linha 2.6, $T_i = T'_i + t_l$. Chamando de $\bar{C}'$ o conjunto $\bar{C}$ na iteração

anterior, temos que $\bar{C} = \bar{C}' \cup \{C_l\}$. Como T'_i e $\bar{C}'$ referem-se à iteração anterior, podemos aplicar a hipótese indutiva sobre T'_i na expressão e obtemos:

$$T'_i = \sum_{C_p \in \bar{C}'} t_p + i \cdot s + t_l.$$

Como $\bar{C} = \bar{C}' \cup \{C_l\}$,

$$T_i = \sum_{C_p \in \bar{C}} t_p + i \cdot s.$$

caso 2. $\bar{R} \neq \emptyset$. Pela linha 2.15, $T_i = T_{i-1} + s$. Seja $\bar{C}'$ o conjunto $\bar{C}$ na iteração anterior. Na forma como o algoritmo foi escrito, $\bar{C} = \bar{C}'$. Assim, aplicando a hipótese indutiva sobre T_{i-1} , obtemos

$$\begin{aligned} T_i &= \sum_{C_p \in \bar{C}'} t_p + (i-1) \cdot s + s. \\ T_i &= \sum_{C_p \in \bar{C}} t_p + i \cdot s. \quad \square \end{aligned}$$

Podemos observar que a descrição é ineficiente, tanto em espaço como em tempo. Isto é feito para obter o invariante de forma mais simples. É claro que numa implementação real, as variáveis T_i podem ser representadas numa única variável simples. Na linha 2.11, a palavra “senão” faz com que os dois blocos do “loop” sejam mutuamente exclusivos, mas isto não é necessário. Se for substituído por “se $R \neq \emptyset$ ”, funcionará normalmente e ainda economizará um retorno do “loop”, quando já existem regras testáveis. Até mesmo a variável i pode ser esquecida numa implementação.

O conjunto $\bar{C}$ no algoritmo contém as condições que já foram avaliadas. O Conjunto R mantém dinamicamente o conjunto das regras que ainda não foram testadas, enquanto $\bar{R}$ é o conjunto das regras não testadas mas que tem todas as condições relevantes já avaliadas.

Podemos observar no algoritmo que inicialmente t_A vale zero e que a cada novo valor de i , a ele é incrementado o valor recém-calculado de $T_i \cdot f_{j_i}$. Assim, podemos escrever:

$$t_A = \sum_{i=1}^n T_i \cdot f_{j_i}$$

Pelo teorema 19,

$$T_i = \sum_{C_p \in \bar{C}_i} t_p + i \cdot s,$$

onde $\bar{C}_i$ é o valor de $\bar{C}$ na iteração em que T_i foi calculado. Então podemos escrever

$$t_A = \sum_{i=1}^n \left(\sum_{C_p \in \bar{C}_i} t_p + i \cdot s \right) \cdot f_{j_i}.$$

Por fim, a linha 2.13, faz com que

$$t_A = \sum_{i=1}^n \left(\sum_{C_p \in \bar{C}_i} t_p + i \cdot s \right) \cdot f_{j_i} + \left(\sum_{p=1}^m t_p + i \cdot s \right) \cdot \bar{f},$$

pois na última iteração, $\bar{C} = \{C_1, \dots, C_m\}$.

Confrontando este resultado com a fórmula 2.2 que determina o tempo médio, podemos ver que o sub-algoritmo calcula corretamente o valor de t_A .

Analogamente, os tempos médios de seleção de uma regra por outras estratégias podem ser obtidos com pouquíssimas alterações na parte relativa à ordem de escolha das condições.

A geração das matrizes M e T no algoritmo com interrupção é idêntica ao caso do método de máscara de regra básico. Os demais comandos que devem ser gerados no código objeto produzido pela estratégia "A" segue os passos do sub-algoritmo que calcula t_A . A tarefa de calcular custos é trocada pela geração de comandos de comparação e desvio para as ações selecionadas. Assim, na primeira metade do "loop" são produzidos os comandos de avaliação das condições, do tipo

if C_l then $a_l = 1$ else $a_l = 0$;

e na segunda metade são produzidos os comandos de seleção de uma regra, do tipo

if $S \cdot M_{ji} = T_{ji}$ then A_{ji} .

De forma análoga, a tradução de tabelas por outras estratégias de interrupção pode ser feita sem grandes alterações.

No nosso exemplo abstrato mostrado para ilustrar o método de tradução por máscara de regra com interrupção, percebemos uma pequena economia de tempo em comparação com o método básico. Resultados semelhantes são apresentados por Pollack [Po 74]. King [Ki 66] em seu artigo original apresenta resultados em que dependendo da situação o ganho pode ser altamente favorável, chegando em alguns casos extremos a uma economia de mais de 70%. Mas uma característica comum entre os dados, é que os resultados obtidos pelas diferentes estratégias de interrupção não são muito diferentes e em grande número dos casos estão muito próximos. Em função disto, pode ser uma alternativa adequada a escolha de uma estratégia para conversão, a priori, sem calcular os tempos médios de execução para cada uma das estratégias.

2.3.3 Máscara de Condição

Este é mais um dos métodos de máscara para conversão de tabelas de decisão e foi descrito separadamente por dois grupos [Oe 68, Bj 68], sendo o que mais de perto segue a definição de regra selecionável. Uma versão modificada deste método pode ser encontrado em [Mu-Ra 70] e uma descrição de sua implementação em [Da-We 73]. Ele consiste basicamente [Me-Ba 77] na utilização do vetor de estados $S = (a_1, \dots, a_m)$ e de uma matriz de máscara M construída da seguinte forma:

$$\begin{aligned} \text{se } a_i = 1 \quad M_{ij} &= \begin{cases} 0 & \text{se } C_{ij} = \text{"N"} \text{ ou } \text{"*"} \\ 1 & \text{caso contrário.} \end{cases} \\ \text{se } a_i = 0 \quad M_{ij} &= \begin{cases} 0 & \text{se } C_{ij} = \text{"Y"} \text{ ou } \text{"\$"} \\ 1 & \text{caso contrário.} \end{cases} \end{aligned}$$

Obtém-se um vetor T a partir da operação "and" entre as linhas da matriz de máscara M , ou seja,

$$T = (t_1, t_2, \dots, t_n),$$

onde $t_j = M_{1j} \cdot M_{2j} \cdot M_{3j} \cdots M_{mj}$. O vetor T representa o resultado de operações “and” entre as linhas da matriz M . Denotando a matriz M por uma sequência de linhas $M_1, M_2, \dots, M_m$, $T = M_1 \cdot M_2 \cdots M_m$.

O método assegura que o vetor T contém bit 1 nas posições correspondentes às regras selecionáveis, isto é, a regra R_j é selecionável se o j -ésimo elemento do vetor T contém o valor 1. É claro que, se a tabela não é ambígua, não mais que um elemento com valor 1 pode existir no vetor T . Vamos exemplificar através da tabela seguinte:

	R_1	R_2	R_3	R_4
C_1	Y	N	N	N
C_2	–	Y	N	N
C_3	–	Y	Y	N

Suponha um vetor de estados $S = (0, 0, 1)$, ou seja,

$$C_1 = \text{falso} \qquad C_2 = \text{falso} \qquad C_3 = \text{verdadeiro}.$$

Vamos construir a matriz de máscara M por linhas M_1, M_2 e M_3 , para a transação especificada por S .

$$\text{Como } a_1 = 0, \qquad M_1 = (0, 1, 1, 1)$$

$$\text{Como } a_2 = 0, \qquad M_2 = (1, 0, 1, 1)$$

$$\text{Como } a_3 = 1, \qquad M_3 = (1, 1, 1, 0)$$

Assim, a matriz de máscara M é a seguinte:

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Fazendo as operações “and” entre os elementos da matriz M , linha a linha, encontramos o vetor $T = (0, 0, 1, 0)$. Assim, seleciona-se a regra correspondente à coluna da matriz de máscara M que tem todos os elementos iguais a 1.

Teorema 20 *Seja t_j um elemento do vetor $T = (t_1, t_2, \dots, t_n)$. Então $t_j = 1$ se e somente se R_j é selecionável.*

Dem.:

$\Leftarrow$. Como R_j é selecionável, então $V(B_j) = 1$, onde B_j é a função-and associada à regra R_j . Logo, $V(C_{ij}) = 1, 1 \leq i \leq m$.

caso 1. $a_i = 1$. Como $V(C_{ij}) = 1$, então $C_{ij} = \text{“Y”}$, $\text{“$”}$ ou “–” . Logo, por construção, $M_{ij} = 1$.

caso 2. $a_i = 0$. Como $V(C_{ij}) = 1$, então $C_{ij} = \text{“N”}$, “*” ou “–” . Logo, por construção, $M_{ij} = 1$. Portanto, $t_j = M_{1j} \cdot M_{2j} \cdots M_{mj} = 1 \cdot 1 \cdots 1 = 1$.

$\Rightarrow$.

Como $t_j = 1$, então $M_{ij} = 1, 1 \leq i \leq m$.

caso 1. $a_i = 1$. Como $M_{ij} = 1$, então $C_{ij} = "Y", "$" ou "-"$. Logo, $V(C_{ij}) = 1$.

caso 2. $a_i = 0$. Como $M_{ij} = 1$, então $C_{ij} = "N", "*" ou "-"$. Logo, $V(C_{ij}) = 1$.

Portanto, $V(B_j) = V(C_{1j}) \cdot V(C_{2j}) \cdot \dots \cdot V(C_{mj}) = 1$ e, por definição, R_j é selecionável. $\square$

O teorema 20 anterior garante que o algoritmo "Condition Mask" produz um código que seleciona corretamente uma regra. Da maneira como foi aqui apresentado, ele tem o inconveniente de gerar em tempo de execução a matriz M , uma para cada transação. Esta forma tem o objetivo de facilitar o entendimento, mas na prática, se produz em lugar da matriz M , duas matrizes V e F estáticas, a primeira refletindo o caso das avaliações das condições serem verdadeiras e a segunda, para o caso de serem falsas. O vetor T é calculado sequencialmente, a cada condição avaliada. Se a condição C_1 é verdadeira T inicia com a primeira linha da matriz V . Caso contrário, inicia com a primeira linha da matriz F . Se a condição C_2 é verdadeira T é atualizado com o resultado da operação "and" entre a segunda linha de V e o próprio T . Caso contrário, ele é atualizado com o resultado da operação "and" entre a segunda linha da matriz F e o próprio T . Assim, sucessivamente com cada linha de condição da tabela de decisão, até encontrar o vetor T final. Desta forma, as matrizes de máscara são produzidas em tempo de conversão. Produzido assim sequencialmente à medida que as condições são avaliadas, permite interromper o processo, se todos os elementos de T são nulos. Neste ponto, pode-se desviar para executar a "else rule", pois nenhuma das regras é satisfeita pela transação. Se existe a convenção de que sempre existe uma única regra a ser satisfeita e não se permite a "else rule", então o processo pode ser interrompido quando um único elemento de T contém o valor 1, independentemente de se ter completado o exame de todas as condições. Chegando ao final do processo, a verificação da ambigüidade em tempo de execução é possível pelo simples fato de se ter mais que uma ocorrência do valor "1" no vetor T . Oerter [Oe 68], Bjork [Bj 68] e Dathe [Da 72] descreveram este método através de um vetor de n elementos inicializados com 1, sendo que o i -ésimo elemento é alterado para zero assim que a avaliação de uma condição impeça que a i -ésima regra seja satisfeita, que é uma forma diferente de expressar o algoritmo descrito.

2.4 Método de Fatorações Sucessivas

É também chamado de método por decomposição ou bifurcação [Ki-Jo 75] e é encontrado na literatura como uma das variações do "test and branch method"¹. Consiste na aplicação sucessiva da operação de fatoração (vide 1.15) sobre a tabela de decisão, até que as tabelas finais sejam triviais. Para ilustrar, suponhamos a seguinte tabela T, com condições arbitrárias C_1, C_2 e C_3 e regras $R_1, R_2, \dots, R_5$ e ações quaisquer que não são mostradas.

	R_1	R_2	R_3	R_4	R_5
C_1	Y	Y	N	–	N
C_2	–	Y	Y	N	N
C_3	N	Y	–	Y	N

Fatorando a tabela T pela linha de condição C_1 , obtemos uma árvore de fatoração com a

¹Método genérico de conversão que produz árvores de decisão.

condição C_1 na raiz e as tabelas T_Y^1 e T_N^1 nas folhas.

Fatorando, agora, as sub-tabelas T_Y^1 e T_N^1 pela linha de condição C_2 , obtemos uma árvore de fatoração de T de profundidade 2, mostrada logo a seguir. Como assinalado na seção 1.15, a aresta à esquerda sempre será rotulada de “Y” e à direita de “N” e isto justifica que os rótulos das arestas não constem mais das figuras. Observamos que as sub-tabelas poderiam também ser fatoradas por linhas distintas, por exemplo, T_Y^1 pela linha C_2 e T_N^1 pela linha C_3 .

Fatorando novamente, uma a uma, as quatro sub-tabelas que estão nas folhas, obtemos a árvore de fatoração que está mostrada na figura 2.7. Como podemos observar todas elas são triviais, indicando que o algoritmo de fatorações sucessivas terminou.

A árvore resultante destas fatorações sucessivas com tabelas triviais nas folhas é chamada “STP” (“sequential testing procedure”) [Me-Ba 77]. A execução do código gerado por este algoritmo faz com que a árvore de fatoração seja percorrida até atingir uma folha. Neste ponto, como a tabela é trivial, as ações são executadas. O algoritmo deve produzir estes caminhos a serem percorridos e os comandos que executam as ações no final de cada percurso.

Para não sobrecarregar a notação, as regras R_j , R_j^1 , $(R_j^1)^2$, $((R_j^1)^2)^3$ (vide 1.14) estão anotadas por R_j , sem prejuízo do entendimento em razão do contexto.

As tabelas produzidas pela fatoração de T por C_1 são T_Y^1 e T_N^1 . As produzidas pela fatoração destas por C_2 são $(T_Y^1)_Y^2$, $(T_Y^1)_N^2$, $(T_N^1)_Y^2$, $(T_N^1)_N^2$. Desta forma, as tabelas produzidas pela

Figura 2.7: Uma STP de T

fatoração têm uma notação mais carregada, quanto mais próximas estiverem das folhas. Na maioria das vezes, dispensaremos esta notação sobrecarregada que dificulta a leitura.

Consideremos a seguinte sub-tabela produzida no passo intermediário do exemplo anterior.

	R ₃
C ₃	—

A fatoração desta tabela produz duas sub-tabelas idênticas, como está ilustrado a seguir, representado apenas pela sub-árvore direita da STP construída no exemplo.

Fatorando por linha irrelevante

Ignorando linha irrelevante

Na figura à esquerda, as duas sub-tabelas produzidas são idênticas em razão da fatoração por uma linha irrelevante. Isto significa que independentemente da condição C_3 ser verdadeira ou falsa as mesmas ações são executadas. Ou seja, a avaliação da condição C_3 é desnecessária nesta etapa, como mostra a figura da direita. Assim, obtendo-se as sub-tabelas T_N e T_Y eliminam-se as linhas de condição irrelevantes (contêm somente “—”, “\$” e “*”).

O exemplo acima trata de fatorações sucessivas sobre uma tabela completa, pois caso contrário ocorreriam nós na árvore com apenas um filho. Na prática, quando isto ocorre, existindo a “else rule”, é o momento da sua execução. Não há a necessidade de se carregar a “else rule” por todas as etapas das fatorações. Pode-se guardá-la na memória global para que sua execução seja especificada somente quando um nó da árvore tiver um único filho.

Poderíamos obter STP's diferentes para a mesma tabela, escolhendo outra ordem das condições pelas quais as tabelas inicial e intermediárias são fatoradas. Precisamos também assegurar que o algoritmo seleciona corretamente o conjunto de ações a ser executado. A regra é selecionada percorrendo o caminho desde a raiz da STP até a folha.

Teorema 21 *Seja T uma tabela de decisão limitada com condições $C_1, C_2, \dots, C_m$ e S uma STP de T . A tabela T e a STP S produzem a execução das mesmas ações.*

Dem.: Por indução em m .

Base da Indução: $m = 0$. Como T é trivial, S é uma árvore que tem somente a raiz com a tabela T . Logo, a tese é válida.

Suponhamos que a tese é válida para tabelas com menos que m condições, $m > 0$. Seja C_i a linha de condição utilizada para a fatoração de T e construção de T_Y e T_N , que são a divisão de T por C_i . Sejam S_Y e S_N as sub-árvores esquerda e direita de S . Por construção, S_Y e S_N são as STP's de T_Y e T_N respectivamente. Pelo teorema 9, R_j é executada em T se e somente se R_j^i é executada na fatoração de T .

Além disso, T_Y e T_N possuem menos que m condições e são tabelas limitadas. Assim, vale a hipótese indutiva e a tese está garantida para T_Y e T_N com respeito a S_Y e S_N . Logo, as mesmas ações são executadas por S_Y e T_Y e também por S_N e T_N . Portanto, as mesmas ações são executadas por T e S . $\square$

Teorema 22 *Sejam T uma tabela de decisão limitada com condições $C_1, C_2, \dots, C_m$, S uma STP de T e $T_1, T_2, \dots, T_p$ o conjunto de tabelas triviais que estão nas folhas de S . T não é ambígua se e somente se $T_1, T_2, \dots, T_p$ não são ambíguas.*

Dem.: Por indução no número de condições m .

Base da indução: $m = 0$. Como T é trivial, $T = T_1$ e a tese é válida.

Seja T uma tabela de decisão com $m > 0$ condições e suponha a tese válida para tabelas com menos que m condições. Sejam C_i a linha de condição utilizada para a fatoração de T na primeira etapa da construção de S , e T_Y e T_N a divisão de T por C_i . Sejam S_Y e S_N as sub-árvores

esquerda e direita de S . Por construção, S_Y e S_N são as STP's de T_Y e T_N , respectivamente. Pelo teorema 7, T não é ambígua se e somente se T_Y e T_N não são ambíguas. Além disso, T_Y e T_N têm menos que m condições e são limitadas. Assim, pela hipótese de indução, T_Y e T_N não são ambíguas se e somente se $T_1, T_2, \dots, T_p$ não são ambíguas. Portanto, T não é ambígua se e somente se $T_1, T_2, \dots, T_p$ não são ambíguas. $\square$

O teorema 22 acima garante que a verificação de ambigüidades pode ser feita nas tabelas triviais da STP produzida pelo algoritmo, não havendo a necessidade desta tarefa ser feita separadamente em etapa anterior. O algoritmo constrói a STP para efetuar a tradução e paralelamente pode detectar ambigüidades. A verificação de ambigüidades em tabelas triviais é extremamente simples, bastando examinar se tem mais que uma regra, como já foi visto em 1.8. Este método apresenta já, à primeira vista, esta vantagem.

A tradução de uma tabela de decisão por este algoritmo é imediata através dos passos de uma STP. Cada nó interno (nós que não são folhas) da árvore gera um comando "if". As folhas produzem a enumeração das ações a serem executadas. Assim, a STP da figura 2.7 produziria a seguinte sequência de comandos em Pascal.

```

if  $C_1$  then
  if  $C_2$  then
    if  $C_3$  then  $R_2$ 
    else  $R_1$ 
  else if  $C_3$  then  $R_4$ 
  else  $R_1$ 
else if  $C_2$  then  $R_3$ 
  else if  $C_3$  then  $R_4$ 
  else  $R_5$ .

```

Neste trecho do programa acima, os nomes $R_1, R_2, \dots, R_n$ representam as ações (comandos) a serem executadas por regras com os respectivos nomes. Cada nó da STP que contém uma condição produz uma comparação e cada folha da árvore produz a execução das ações sem qualquer comparação.

A figura anterior mostra duas sub-árvores contendo nós com um único filho. Se a "else rule" E está presente na tabela original, obteríamos a seguinte sequência de comandos para as sub-árvores acima:

```

if  $C_i$  then ...
else  $E$ 
if not  $C_i$  then ...
else  $E$ 

```

Como dissemos, a seleção de uma regra é feita percorrendo-se um caminho com origem na raiz da STP S , terminando numa folha. É fácil ver que num tal caminho os nós internos não repetem condições.

Teorema 23 *Sejam S uma STP associada a uma tabela de decisão e $P = (v_0, v_1, \dots, v_p)$ um caminho em S , onde v_0 é a raiz e v_p uma folha qualquer de S . Seja $c(v)$ a condição presente no nó interno v . Então $c(v_i) \neq c(v_j), i \neq j$ e $0 \leq i, j < p$.*

Dem.: Suponhamos sem perda de generalidade que v_j está na STP S' com raiz em v_i que contém a condição C e seja T' a tabela de decisão associada à STP S' . Sejam S'_Y e S'_N as sub-árvores esquerda e direita de v_i . Claramente S'_Y e S'_N são as STP's de T'_Y e T'_N , onde estas são a divisão de T' por C . Por construção, T'_Y e T'_N não contêm a linha de condição C . Logo, S'_Y e S'_N não contêm vértice v tal que $c(v) = C$.

Como $v_i \neq v_j$, v_j está em S'_Y ou em S'_N . Além disso, $c(v_i) = C$. Portanto, $c(v_i) \neq c(v_j)$. $\square$

Corolário 24 *Seja T uma tabela de decisão limitada com m linhas de condição. A seleção de uma regra na execução do código produzido pelo método de fatorações sucessivas faz no máximo m avaliações de condição (no máximo m comparações).*

Dem.: Como a seleção de uma regra se faz percorrendo um caminho $P = (v_0, v_1, \dots, v_p)$ da raiz à folha de uma STP de T e os vértices internos (inclusive v_0) da STP contêm uma condição, são feitas p avaliações de condição.

Seja $V = \{v_0, v_1, \dots, v_{p-1}\}$ o conjunto dos vértices que não são folhas de P . Claramente, o número de avaliações de condição para a seleção de uma regra é $|V| = p$.

Sejam $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$ o conjunto das condições de T e c a seguinte função:

$$\begin{aligned} c : V &\longrightarrow \mathcal{C} \\ c(v) &= \text{condição presente no vértice } v. \end{aligned}$$

Como, por construção, todo vértice de V contém somente uma condição, a função está definida. Além disso, pelo teorema 23, $c(v_i) \neq c(v_j), \forall i \neq j$. Logo a função c é injetora. Portanto, $|V| \leq |\mathcal{C}| = m$. $\square$

O corolário 24 acima garante que o código produzido por este método não avalia uma mesma condição duas vezes para uma transação.

É interessante observar que por tratar-se de uma árvore existe um único caminho da raiz a uma folha [Bo-Mu 77], mas folhas distintas podem conter uma mesma tabela trivial, ou seja, produzem a seleção de uma mesma regra.

2.4.1 Número de STP's

Considere a tabela de decisão T abaixo:

	R_1	R_2	R_3
C_1	Y	N	N
C_2	–	Y	N

Figura 2.8: Tabela T

Vamos apresentar agora as STP's S_1 e S_2 de T , onde a primeira começa fatorando por C_1 e a segunda por C_2 .

Figura 2.9: STP's S_1 e S_2 de T

Podemos observar que embora todas as STP's produzam códigos eficientes, no sentido de que a execução do código gerado fará no máximo m avaliações de condição para a seleção de uma regra, existem variações de desempenho entre elas. Por exemplo, a seleção da regra R_1 na STP S_1 faz somente a avaliação da condição C_1 , enquanto a mesma seleção na STP S_2 avalia as condições C_1 e C_2 . Assim, se as regras são executadas com probabilidade uniforme, em S_1 teremos em média $3/2$ comparações, contra 2 no caso de S_2 . Disto podemos deduzir que a escolha da condição a ser utilizada para a fatoração é importante.

Nesta discussão, caímos no problema de calcular o número de STP's existentes para uma tabela de decisão T [Me-Ba 77, Apêndice]. No caso do exemplo anterior com duas condições, existem somente as STP's S_1 e S_2 . Do ponto de vista puramente de árvores, existem outras possibilidades, porém isomorfas. Na verdade, estamos construindo as diversas árvores com diferentes sequências de comparações possíveis.

Seja uma tabela de decisão T com condições $C_1, C_2, \dots, C_m$. Vamos supor que a STP S de T tem a condição C_1 na raiz. Dividindo T por C_1 , obtemos T_Y e T_N . Vamos denotar o número de STP's diferentes que se pode construir para uma dada tabela T por $f(T)$. Assim, o número de STP's com esta raiz é $f(T_Y) \cdot f(T_N)$. Como cada uma das m condições podem estar na raiz da STP,

$$f(T) = f(T_Y^1) \cdot f(T_N^1) + f(T_Y^2) \cdot f(T_N^2) + \dots + f(T_Y^m) \cdot f(T_N^m).$$

De um modo geral, todas as parcelas desta expressão tem valores distintos, pois as sub-tabelas produzidas em cada caso têm tamanhos distintos. Para um estudo da grandeza do número

$f(T)$, vamos nos limitar a calcular o número de árvores possíveis para tabelas completas e sem entradas irrelevantes. Neste caso, todo caminho da raiz a uma folha qualquer passa por m vértices, cada uma delas contendo uma condição distinta. Além disso, cada nó interno da STP tem dois filhos. Assim $f(T)$ passa a depender somente do número de condições de uma tabela, pois $f(T)$ é o número de combinações diferentes das condições nos vértices de uma árvore binária completa de altura m , onde cada caminho da raiz a uma folha não encontra condições repetidas e podemos representar $f(T)$ por $f(m)$. O número de árvores possíveis para cada condição C_i na raiz é o produto do número de possibilidades para as sub-árvores direita e esquerda. Isto é verdade, porque, para cada uma das sub-árvores possíveis da esquerda, podemos combinar todas as possibilidades para a sub-árvore da direita.

Como T_Y e T_N têm $m - 1$ condições e, pelo teorema 10, herdam as propriedades de completude, o número de possíveis árvores para cada uma delas é $f(m - 1)$. Assim, temos $f(m - 1) \cdot f(m - 1)$ possíveis árvores para a tabela original com uma determinada condição C_i na raiz. Como existem m possibilidades para a raiz, então o número de possíveis STP's é [Me-Ba 77]

$$f(m) = m \cdot f(m - 1) \cdot f(m - 1) = m \cdot (f(m - 1))^2.$$

Teorema 25 *O número de STP's distintas para uma tabela completa não trivial com m condições e sem entradas irrelevantes é*

$$f(m) = \prod_{i=0}^{m-1} (m - i)^{2^i}.$$

Dem.: Para demonstrar o teorema, basta provar que a expressão para a função $f(m)$ que está no enunciado é uma solução para a fórmula de recorrência $f(m) = m \cdot (f(m - 1))^2$, sabendo que para tabelas com uma só condição existe uma única STP.

Por indução em m .

Base da indução: $m = 1$. $f(m) = \prod_{i=0}^0 (m - i)^{2^i} = (1)^{2^0} = 1^1 = 1$.

Suponha que a tese é válida para valores menores que m , $m > 1$. Pela recorrência,

$$f(m) = m \cdot (f(m - 1))^2.$$

Mas pela hipótese de indução,

$$\begin{aligned} f(m-1) &= \prod_{i=0}^{m-2} (m-i-1)^{2^i} \\ &= (m-1) \cdot (m-2)^2 \cdot (m-3)^4 \cdot \dots \cdot (m-m+1)^{2^{m-2}}. \end{aligned}$$

Assim,

$$f^2(m-1) = (m-1)^2 \cdot (m-2)^4 \cdot (m-3)^8 \cdot \dots \cdot (m-m+1)^{2^{m-1}}.$$

Portanto,

$$\begin{aligned} f(m) &= m \cdot f^2(m-1) \\ &= m \cdot (m-1)^2 \cdot (m-2)^4 \cdot (m-3)^8 \cdot \dots \cdot (m-m+1)^{2^{m-1}} \\ &= \prod_{i=0}^{m-1} (m-i)^{2^i}. \quad \square \end{aligned}$$

A fórmula anterior é o número total de árvores sem eliminação daquelas que são isomorfas. Vamos tentar obter a fórmula de recorrência para o número $g(m)$ que é o número de STP's distintas não isomorfas entre si para tabelas de decisão completas sem entrada de condição irrelevante e com m linhas de condição.

Neste caso, a raiz pode abrigar condições $C_1, C_2, \dots, C_m$. Para cada condição C_i na raiz, o número de árvores possíveis corresponde às combinações das suas sub-árvores esquerda e direita. Como a sub-árvore esquerda é uma STP correspondente a uma tabela T_Y com $(m-1)$ linhas de condição, existem $g(m-1)$ sub-árvores possíveis para T_Y . Digamos que sejam sub-árvores $S_1, S_2, \dots, S_{g(m-1)}$. Para S_1 como sub-árvore esquerda existem também $g(m-1)$ sub-árvores à direita possíveis para a STP de T_N .

Para S_2 como sub-árvore esquerda, existem $g(m-1)$ sub-árvores possíveis à direita como mostra a figura seguinte. Mas podemos ver que a segunda STP da figura anterior com S_1 e S_2 como sub-árvores esquerda e direita, respectivamente e a primeira STP da figura seguinte com S_2 e S_1 como sub-árvores esquerda e direita respectivamente são isomorfas. Elas diferem apenas de uma simples troca de lados entre as sub-árvores. Assim contabilizamos $(g(m-1) - 1)$ STP's distintas e não isomorfas às já contabilizadas com S_2 como sub-árvore esquerda.

Analogamente, para S_3 como sub-árvore esquerda, contabilizamos $(g(m-1) - 2)$ STP's e assim sucessivamente, até que para $S_{g(m-1)}$ como sub-árvore esquerda existe somente uma sub-árvore direita de modo a não produzir STP's isomorfas, qual seja, a própria $S_{g(m-1)}$.

Assim, o número de STP's distintas e não isomorfas entre si com a condição C_i na raiz é

$$\begin{aligned}
 & g(m-1) + (g(m-1) - 1) + (g(m-1) - 2) + \\
 & + (g(m-1) - 3) + \cdots + 1 = \\
 & = \frac{g(m-1) + 1}{2} \cdot g(m-1) \\
 & = \frac{g(m-1) \cdot (g(m-1) + 1)}{2}.
 \end{aligned}$$

Como existem m possibilidades para a condição da raiz, [Me-Ba 77]

$$g(m) = m \cdot \frac{g(m-1) \cdot (g(m-1) + 1)}{2}.$$

Podemos obter $f(m)$ pela expressão. Como sabemos que $g(1) = 1$ e $g(2) = 2$, podemos calcular $g(m)$ pela fórmula de recorrência.

m	$f(m)$	$g(m)$
1	1	1
2	2	2
3	12	9
4	576	180
5	1.658.880	81.450
6	16.511.297.126.400	19.902.551.850

A tabela acima apresentada em [Me-Ba 77] ilustra os números de STP's para uma tabela completa sem entradas irrelevantes e com m condições. Pode-se observar o crescimento absurdo da função em relação a m . De imediato, pode-se concluir que não é uma tarefa computacionalmente viável um estudo caso a caso de todas as STP's de uma tabela para se escolher a melhor delas. Deste modo, posto que o método de conversão por STP produz um código que avalia cada condição no máximo uma vez para a seleção de uma regra, e isto parece um bom desempenho para o código gerado, pode-se adotar a ordem de avaliação das condições para a construção de uma STP como aquela em que aparecem na tabela de decisão. Existem algumas heurísticas que tentam produzir uma sequência melhor de comparações, como veremos mais adiante e também algoritmos que produzem STP's ótimas para tabelas com pequeno número de condições.

Hyfil & Rivest [Hy-Ri 76] demonstraram que o problema de construção da árvore de decisão binária ótima é NP-Completo [Ah-Ho-UI 74, Lu-Si-Ko 79], através de uma redução do problema da cobertura exata por conjuntos de três elementos (EC3) [Hy-Ri 76, Ga-Jo 79].

2.4.2 Estrutura de Dados para o Algoritmo

A eficiência de um algoritmo é influenciada pela estrutura de dados utilizada. A tabela de decisão pode ser representada matricialmente. Estes algoritmos que constróem sub-tabelas precisam produzir estas matrizes inteiras à medida que novas tabelas são criadas. No caso das tabelas produzidas pela divisão de outra tabela por uma linha de condição, observamos que as tabelas resultantes, na verdade, são a reprodução de partes da tabela original. Assim, uma alternativa de representação destas novas tabelas seria uma estrutura indicativa de quais partes da tabela original ela é composta e não reconstruir integralmente as matrizes de dados. Como os algoritmos utilizados sempre partirão de uma tabela inicial e produzirão sub-tabelas, vamos adotar uma estrutura de dados do seguinte tipo, sem entrarmos em detalhes:

- Uma representação matricial para a tabela original, utilizada como estrutura de dados global a ser consultada, sempre que for necessário.
- Dois vetores C e R para cada tabela, contendo informações sobre quais partes da tabela original está sendo utilizada. O vetor C é uma lista de condições e R uma lista de regras.

Para explicar melhor, seja T a tabela de decisão inicial com um conjunto de condições

$$C = \{C_1, C_2, \dots, C_m\},$$

um conjunto de regras

$$R = \{R_1, R_2, \dots, R_n\}$$

e conjunto de ações

$$A = \{A_1, A_2, \dots, A_l\}.$$

Toda sub-tabela W de T (incluse T) será representada por dois vetores C_W e R_W da seguinte forma:

$$\begin{aligned} C_W &= (C_{i_1}, C_{i_2}, \dots, C_{i_p}), \text{ onde } C_{i_k} \text{ está em } W \text{ e } C_{i_k} \in C, 1 \leq k \leq p \\ R_W &= (R_{j_1}, R_{j_2}, \dots, R_{j_q}), \text{ onde } R_{j_k} \text{ está em } W \text{ e } R_{j_k} \in R, 1 \leq k \leq q \end{aligned}$$

É claro que para a tabela inicial $W = T$, $C_{i_k} = C_k$, $R_{j_l} = R_l$, $|C_W| = |C|$ e $|R_W| = |R|$. Normalmente, vamos chamar simplesmente de vetores C e R em lugar de C_W e R_W .

Deste modo, para obter a representação de uma sub-tabela, basta simplesmente construir uma sequência de condições e outra de regras. Numa implementação real, usam-se C e R não como sequências das linhas de condição e das regras presentes na tabela, mas dos seus índices.

2.4.3 Linhas de Condição irrelevantes

Seja T uma tabela de decisão limitada e não trivial. Uma *linha de condição é irrelevante* se todas as entradas de condição da linha são “-”, “\$” ou “*”. É claro que uma tabela de decisão que descreve um problema qualquer parece não comportar, de forma natural, condições desta espécie, pois elas não participam dos mecanismos de decisão do problema. Uma tal ocorrência poderia até ser detectada pelos pré-processadores para as tabelas de entrada. Assim, a eliminação de tais linhas não altera a tabela original, no sentido de que a execução de uma ou outra produz as mesmas ações. Nestes termos, dizemos que são *tabelas equivalentes*.

A despeito da inoportunidade de linhas irrelevantes em tabelas de decisão feitas para um determinado problema, elas podem ocorrer como sub-tabelas produzidas por uma operação de divisão de uma tabela por uma linha de condição, como no seguinte exemplo:

	R ₁	R ₂	R ₃
C ₁	Y	Y	N
C ₂	Y	N	N
C ₃	\$	–	N

Dividindo a tabela de decisão acima pela linha de condição C_1 , obtemos como T_Y e T_N , as seguintes tabelas.

	R ₁	R ₂
C ₂	Y	N
C ₃	\$	–

	R ₃
C ₂	N
C ₃	N

A tabela à esquerda da figura é T_Y e contém a linha de condição C_3 com entradas “\$” e “–”, ambas irrelevantes. Assim, a linha de condição C_3 da tabela T_Y é irrelevante e pode ser eliminada, porque ela não interfere no processo de seleção das regras. Precisamos apenas estar seguros de que tal eliminação não introduz ambigüidades na tabela.

Lema 26 *Seja T uma tabela de decisão limitada, não trivial e não ambígua com condições $C_1, C_2, \dots, C_m$ e regras $R_1, R_2, \dots, R_n$ e C_{ij} uma entrada de condição com o símbolo “–”. Seja T' a tabela obtida a partir de T , desdobrando a entrada de condição “–” de C_{ij} . As tabelas T e T' são equivalentes.*

Dem.: Temos que mostrar que as mesmas ações são executadas por T e T' . Como visto em 1.4, a operação de desdobramento do símbolo “–” na entrada de condição C_{ij} produz duas regras no lugar da regra R_j original.

Por construção, T' possui regras $R_1, R_2, \dots, R_{j-1}, R_{j+1}, \dots, R_n, R_{n+1}, R_{n+2}$, onde as regras R_{n+1} e R_{n+2} são originárias de R_j com o desdobramento de “–” que está em C_{ij} . Assim,

$$\begin{aligned} C_{tj} &= C_{t,n+1} = C_{t,n+2}, 1 \leq t \leq m, t \neq i, \\ C_{i,n+1} &= \text{“Y”} \text{ e} \\ C_{i,n+2} &= \text{“N”}. \end{aligned}$$

Claramente, R_k é selecionável em T se e somente se R_k é selecionável em T' , $1 \leq k \leq n, k \neq j$. Assim, T e T' são equivalentes em relação a estas regras.

Se R_j é selecionável em T , então $V(C_{tj}) = 1, 1 \leq t \leq m$. Além disso, $V(C_{i,n+1}) = 0$ e $V(C_{i,n+2}) = 1$ ou vice-versa. Logo, um entre R_{n+1} e R_{n+2} é selecionável em T' .

Por outro lado, se R_{n+1} ou R_{n+2} é selecionável em T' , R_j é selecionável em T , pois $C_{ij} = \text{“–”}$ e as demais entradas de condição da regra R_j são idênticas às das regras R_{n+1} e R_{n+2} . $\square$

Corolário 27 *Sejam T uma tabela de decisão limitada não trivial e não ambígua com condições $C_1, C_2, \dots, C_m$ e regras $R_1, R_2, \dots, R_n$ e T' a tabela obtida a partir de T com um número arbitrário de desdobramentos das entradas de condição “-”. T e T' são equivalentes.*

Dem.: Segue imediatamente do lema 26 acima. Considere as tabelas intermediárias por desdobramentos um a um das entradas “-”. Assim, seja

$$T \longrightarrow T_1 \longrightarrow T_2 \longrightarrow T_3 \longrightarrow \dots \longrightarrow T'$$

a sequência de aplicações de operação de desdobramento do símbolo “-”, até atingir a tabela T' a partir de T . Pelo lema 26, as mesmas ações são executadas entre T e T_1 , entre T_1 e T_2 , entre T_2 e T_3 e assim sucessivamente. Logo as mesmas ações são executadas por T e T' . Portanto, elas são equivalentes. $\square$

Corolário 28 *Sejam T uma tabela de decisão limitada não trivial e não ambígua com condições $C_1, C_2, \dots, C_m$ e regras $R_1, R_2, \dots, R_n$ e T' a tabela obtida a partir de T , substituindo-se p símbolos “*” por “N”, q símbolos “\$” por “Y” e r desdobramentos de “-”. T e T' são equivalentes.*

Dem.: Seja W a tabela obtida a partir de T com a substituição de p símbolos “*” por “N” e q símbolos “\$” por “Y”. Como vimos em 1.6, o valor $V(C_{ij})$ é o mesmo se $C_{ij} = “N”$ ou “*”. Assim os símbolos “N” e “*” são equivalentes um ao outro do ponto de vista de seleção de uma regra. O mesmo se aplica aos símbolos “Y” e “\$”. Logo, W e T são equivalentes. Pelo corolário 27, W e T' são equivalentes. Portanto, T e T' executam as mesmas ações para uma mesma transação e conseqüentemente são tabelas equivalentes. $\square$

O corolário mostra que estas operações produzem tabelas equivalentes, mas a demonstração supôs que primeiramente são feitas as substituições de “\$” e “*” por “Y” e “N” respectivamente, e depois as operações de desdobramentos de “-”. Mas o resultado vale para a aplicação destas operações em qualquer ordem e para um número arbitrário destas operações. Para demonstrar esta afirmação, basta aplicar o último corolário 28 sobre cada uma das tabelas intermediárias que são produzidas antes de se chegar na tabela final.

Teorema 29 *Sejam T uma tabela de decisão limitada e não trivial com linhas de condição $C_1, C_2, \dots, C_m$, C_i uma linha de condição irrelevante em T e T^i a tabela obtida a partir de T com a remoção de C_i . T não é ambígua se e somente se T^i não é ambígua.*

Dem.: Como fizemos na seção 1.14, denotamos por R_j^i a regra de T^i correspondente à regra R_j de T .

$\Rightarrow$. Por contradição. Suponha que existam regras R_j^i e R_k^i ambíguas em T^i . Como não são ambíguas em T e C_i é uma linha de condição irrelevante, $C_{ij} = “*”$ e $C_{ik} = “$”$ ou vice-versa. Sem perda de generalidade, vamos supor que $C_{ij} = “*”$ e $C_{ik} = “$”$. Existem duas possíveis utilizações para estes símbolos.

caso 1. Um deles (ou ambos) foi utilizado como eliminador de ambigüidades na tabela T . Neste caso, existe um erro de construção da tabela, pois deveria existir uma entrada de condição relevante na linha C_i da tabela T , como visto na seção 1.10. Suponhamos agora que $C_{ij} = “*”$

foi utilizado como eliminador de ambigüidade na tabela $T' \neq T$, com respeito a regras R'_j e R'_p em T' e que T é produzida por divisões sucessivas a partir da tabela original T' . R_j e R_p são as regras de T correspondentes a R'_j e R'_p de T' , após estas divisões. Elas de fato estão em T , pois as divisões sucessivas foram efetuadas por linhas de condição diferentes de C_i e a menos desta linha, elas não têm entradas de condição capazes de separá-las em sub-tabelas distintas. Logo C_{ij} e C_{ip} estão na linha de condição C_i em T . Mas pelo critério de utilização do símbolo “*” como eliminador de ambigüidade, a entrada C_{ip} deveria conter a entrada “Y” e isto contradiz o fato de que a linha de condição C_i é irrelevante em T .

De forma análoga, suponha que C_{ik} = “\$” é a entrada utilizada para este fim e se obtém que C_{ip} deve conter o símbolo “N”, o que contradiz a irrelevância da linha C_i .

caso 2. C_{ij} e C_{ik} são utilizados como entradas de condição implícita em T .

Se a tabela T é a tabela originalmente escrita pelo usuário, obtenha uma tabela W a partir de T , substituindo somente nestas duas regras as entradas “*” e “\$” por “N” e “Y”, respectivamente, com exceção de C_{ij} e C_{ik} e todas as entradas “-” com operações de desdobramentos. Como T não é ambígua, pelo corolário 28, W e T são equivalentes, ou seja, elas são representações distintas de um mesmo problema. A linha de condição C_i é irrelevante em W , por construção. Sejam $R_{j1}, R_{j2}, \dots, R_{jp}$ e $R_{k1}, R_{k2}, \dots, R_{kq}$, as regras de W derivadas de R_j e R_k , respectivamente. Como R_j e R_k são ambíguas em T' , existem R_{js} e R_{kt} em W , com todas as entradas de condição idênticas a menos daquelas que estão na linha C_i , onde possuem, respectivamente, entradas “*” e “\$”. Pela utilização de “*”, como entrada implícita o conjunto de condições $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_m$, com a valoração requerida pela regra R_{js} determinam implicitamente que C_i seja falsa. Como “\$” também é utilizado como entrada implícita, então exatamente a mesma valoração para as condições determina implicitamente que C_i seja verdadeira na regra R_{kt} , o que é uma contradição.

Para o caso dos símbolos terem sido usados como entradas implícitas numa tabela $T' \neq T$, sendo T o resultado de um certo número de fatorações de T' , basta considerar a tabela W obtida com as operações efetuadas sobre T' e não sobre T . A existência de regras R_{js} e R_{kt} é garantida pelo mesmo argumento acima somado ao fato de que nas divisões sucessivas a partir de T' até a obtenção de T , não houve nenhuma condição com entradas de condição diferentes o bastante para separá-las em sub-tabelas distintas, pois R_j e R_k estão em T . Daí o mesmo raciocínio se aplica.

⇐. Sejam R_j e R_k duas regras quaisquer em T . Como T^i não é ambígua, existe uma linha de condição C_i em T^i tal que C_{ij} = “Y” ou “\$” e C_{ik} = “N” ou “*” ou vice-versa. Por construção de T^i , C_{ij} e C_{ik} estão nas regras R_j e R_k , respectivamente. Portanto, R_j e R_k não são ambíguas em T . □

Parece imediato que nas tabelas intermediárias produzidas pela divisão, as linhas de condição que contenham somente “-” sejam sumariamente eliminadas, pois estas condições são totalmente irrelevantes para a tabela. Igualmente com as linhas que contenham somente “*” ou somente “\$”. O teorema 29 é importante, pois garante que sendo a tabela construída corretamente e não ambígua, as linhas com entradas irrelevantes podem também ser eliminadas, mesmo que contenham os dois símbolos “*” e “\$”, sem que estejamos introduzindo ambigüidades nas sub-tabelas. Esta propriedade pode ser utilizada pelos pré-processadores baseados no método de fatorações sucessivas.

2.4.4 Descrição do Algoritmo

Antes de apresentar o algoritmo de construção de uma STP associada a uma tabela de decisão, vamos descrever o algoritmo que faz a divisão de uma tabela por uma linha de condição.

Algoritmo: Divisão de uma tabela por uma linha de condição.

Importação: Tabela de decisão T representada pelos vetores $C = (C_{i_1}, C_{i_2}, \dots, C_{i_p})$ e $R = (R_{j_1}, R_{j_2}, \dots, R_{j_q})$ e a linha de condição C_{i_k} pela qual a tabela será dividida.

Exportação: Tabelas T_Y e T_N , representados pelos vetores C_Y, R_Y, C_N e R_N .

início

$C_Y, C_N, R_Y, R_N \leftarrow$ vazio;

para $t \leftarrow 1$ **a** p **faça**

se $t \neq k$ **insere** C_{i_t} em C_Y e em C_N ;

para $t \leftarrow 1$ **a** q **faça**

se $C_{i_k j_t} = \text{"-"} \text{ ou "Y" ou "$"} \text{ insere } R_{j_t} \text{ em } R_Y$;

se $C_{i_k j_t} = \text{"-"} \text{ ou "N" ou "*" } \text{ insere } R_{j_t} \text{ em } R_N$;

fim

A complexidade de tempo do algoritmo é $O(p + q)$, onde p é o número de condições e q o número de regras de T . Podemos observar que o tamanho da entrada é da ordem de $p + q$.

Neste ponto, pode-se eliminar as linhas de condição que contenham somente "-", "\$" ou "*", o que elevaria a complexidade para $O(p \cdot q)$. Como este algoritmo é parte do algoritmo de produção de uma STP, isto é melhor realizado durante a escolha da "próxima condição"² para a divisão, pois para isto é necessário pesquisar cada linha de condição. Feita esta nota, fica subentendido que esta tarefa é realizada em algum ponto adequado, mesmo que não mencionado explicitamente nos algoritmos.

Agora, vamos dar um esboço de algoritmo que produz uma STP. Tal algoritmo constrói todos os passos para a geração dos comandos que executam a tabela de decisão.

Algoritmo: Decompõe Tabela.

Importação: Tabela de decisão T , limitada e não ambígua, representada pelos vetores $C = (C_{i_1}, C_{i_2}, \dots, C_{i_p})$ e $R = (R_{j_1}, R_{j_2}, \dots, R_{j_q})$ e o vértice v onde deve ficar a raiz da STP de T .

Exportação: STP

início

1. **se** T não é trivial

2. **então** escolhe linha de condição C_{i_k} ;

3. **se** T é trivial (* T é folha da STP *)

4. **então** coloca a tabela trivial em v

5. **senão**

6. coloca a condição C_{i_k} em v ;

7. divide a tabela T por C_{i_k} , obtendo T_Y e T_N ;

²Vamos usar este nome para referir-nos à linha de condição que será usada na próxima operação de divisão da tabela.


```

8.      se  $T_Y \neq \emptyset$  então
9.          cria o filho esquerdo  $v_e$  de  $v$  ligado com aresta  $Y$ ;
10.         decompõe Tabela  $T_Y$  com raiz em  $v_e$ ;      (* chamada recursiva *)
11.      se  $T_N \neq \emptyset$  então
12.          cria o filho direito  $v_d$  de  $v$  ligado com aresta  $N$ ;
13.         decompõe Tabela  $T_N$  com raiz em  $v_d$ ;      (* chamada recursiva *)
fim

```

Os comandos das linhas 1 e 2 poderiam estar após a linha 5, mas isto é feito da forma acima, supondo que o comando da linha 2, que escolhe a linha de condição C_{i_k} também faz a busca de linhas de condição irrelevantes e neste processo, pode transformar T numa tabela trivial.

A linha 7 que contém o comando que faz a divisão de T pela linha de condição C_{i_k} produz duas tabelas T_Y e T_N . A primeira delas é utilizada imediatamente, mas a segunda fica guardada por muito tempo até a final decomposição recursiva da primeira e durante este tempo ela não é utilizada. Assim, em lugar do algoritmo de divisão poderíamos usar um algoritmo que constrói somente T_Y na linha 7 e entre as linhas 10 e 11, um que constrói somente T_N . Deste modo, haveria a garantia de que elas seriam construídas no momento de serem usadas. Isto economizaria espaço, mas prejudicaria o tempo de execução do algoritmo, pois a geração de T_Y e T_N pode ser feita com uma só passada em T .

Teorema 30 *O algoritmo “Decompõe Tabela” descrito anteriormente constrói uma STP para a tabela de decisão T , limitada, não ambígua e com linhas de condição $C_1, C_2, \dots, C_m$.*

Dem.: Por indução em m .

Base da indução: $m = 0$. O algoritmo executa as linhas 3 e 4, produzindo a STP trivialmente. Suponhamos que o algoritmo constrói corretamente STP's para tabelas com menos que $m > 0$ condições. Se a tabela tem linhas de condição irrelevantes, obtenha a tabela T' com a sua remoção. Claramente, T' tem menos que m linhas de condição e assim, a hipótese indutiva garante a solução correta do algoritmo. Vamos então considerar o caso em que T não tem linhas de condição irrelevantes. Sem perda de generalidade, vamos supor que C_1 é a linha de condição escolhida pelo comando 2 do algoritmo. Assim, o algoritmo produz a seguinte árvore de fatoração de T .

Se $T_Y \neq \emptyset$, o algoritmo constrói o vértice v_e e a aresta Y que liga os vértices v_e e v e chama recursivamente o algoritmo para a sub-tabela T_Y enraizada em v_e .

Se $T_N \neq \emptyset$, o algoritmo constrói o vértice v_d e a aresta N que liga os vértices v_d e v e chama recursivamente o algoritmo para a sub-tabela T_N enraizada em v_d .

As tabelas T_Y e T_N têm exatamente $m - 1$ condições e são limitadas. Além disso, pelo teorema 7, não são ambíguas. Logo, podemos aplicar a hipótese indutiva sobre as duas sub-tabelas. Assim, o algoritmo constrói corretamente a sub-árvore à esquerda de v para T_Y e a sub-árvore à direita de v para T_N . Estas duas sub-árvores juntadas à árvore parcial da figura anterior, é uma STP de T . Portanto, o algoritmo constrói corretamente uma STP para T . $\square$

Complexidade do Algoritmo

Vamos agora discutir a complexidade deste algoritmo, assumindo que os comandos:

- “cria um vértice v_e (v_d) e uma aresta Y (N)” consome tempo constante.
- “coloca a tabela trivial no vértice v ” consome tempo proporcional ao número de ações, pois uma tabela trivial tem somente a parte de ações. Se a tabela tem l ações, consome tempo $O(l)$.
- operação de comparação consome tempo constante.
- “divide T por C_{i_k} ” consome tempo $O(p + q)$.
- “escolhe a linha de condição C_{i_k} ” consome tempo $O(p \cdot q)$. Por enquanto, vamos adotar esta suposição sem maiores discussões.

Assim, temos facilmente a seguinte fórmula de recorrência para o algoritmo de construção de uma STP correspondente a uma tabela de decisão T , limitada, não ambígua, com m condições, n regras e l ações. Seja $f(m, n, l)$ a complexidade de tempo relativa à tabela T .

$$f(m, n, l) = k_1 \cdot m \cdot n + k_2(m + n) + f(m - 1, n_Y, l) + f(m - 1, n_N, l) + C,$$

onde n_Y e n_N são os números de regras em T_Y e T_N , respectivamente, e k_1, k_2 e C são constantes. Para alguma constante k , podemos reescrever a fórmula acima da seguinte maneira:

$$f(m, n, l) \leq k \cdot m \cdot n + f(m - 1, n_Y, l) + f(m - 1, n_N, l) + C \quad (2.3)$$

$$f(0, 1, l) \leq k \cdot l + C. \quad (2.4)$$

A fórmula 2.4 dá o custo de execução de um tabela trivial com $m = 0$, pois neste caso, o algoritmo executa o comando da linha 4. Vamos considerar que a tabela T não tem entradas de condição “-”. Isto pode ser conseguido com o desdobramento destes símbolos. Nesse caso, $n_Y + n_N = n$.

Teorema 31 *Seja T uma tabela de decisão com m linhas de condição, n regras, l linhas de ação e sem entradas de condição com símbolos “-”. A execução do algoritmo de construção de uma STP para T consome tempo*

$$f(m, n, l) \leq k \cdot m^2 \cdot n + k \cdot n \cdot l + (2 \cdot n - 1) \cdot C. \quad (2.5)$$

Dem.: Vamos provar que a expressão é solução da fórmula de recorrência acima, por indução em m .

Tome por base da indução, $m = 0$. Neste caso, pela fórmula 2.4, $f(0, 1, l) \leq k \cdot l + C$. Por outro lado, utilizando a fórmula 2.5, com $n = 1$, pois a tabela T é trivial e não ambígua, obtemos

$$\begin{aligned} f(0, 1, l) &\leq k \cdot m^2 \cdot n + k \cdot n \cdot l + (2 \cdot n - 1) \cdot C \\ &\leq k \cdot (0) \cdot 1 + k \cdot 1 \cdot l + (2 \cdot 1 - 1) \cdot C \\ &\leq k \cdot l + C. \end{aligned}$$

Portanto, $f(0, 1, l) \leq k \cdot m^2 \cdot n + k \cdot n \cdot l + (2 \cdot n - 1) \cdot C$.

Suponha a expressão verdadeira para tabelas com menos que m condições, $m \geq 1$. Pela fórmula de recorrência 2.3

$$f(m, n, l) \leq k \cdot m \cdot n + f(m - 1, n_Y, l) + f(m - 1, n_N, l) + C.$$

Aplicando a hipótese indutiva em $f(m - 1, n_Y, l)$ e $f(m - 1, n_N, l)$, obtemos,

$$\begin{aligned} f(m, n, l) &\leq k \cdot m \cdot n + k \cdot (m - 1)^2 \cdot n_Y + k \cdot n_Y \cdot l + \\ &\quad (2 \cdot n_Y - 1) \cdot C + k \cdot (m - 1)^2 \cdot n_N + k \cdot n_N \cdot l + \\ &\quad (2 \cdot n_N - 1) \cdot C + C \\ &= k \cdot m \cdot n + k \cdot (m - 1)^2 \cdot (n_Y + n_N) + \\ &\quad k \cdot l \cdot (n_Y + n_N) + (2 \cdot n_Y + 2 \cdot n_N - 1 - 1 + 1) \cdot C \\ &= k \cdot m \cdot n + k \cdot (m - 1)^2 \cdot n + k \cdot n \cdot l + (2 \cdot n - 1) \cdot C, \\ &\quad \text{pois } n_Y + n_N = n \\ &= k \cdot m \cdot n + k \cdot m^2 \cdot n - 2 \cdot k \cdot m \cdot n + k \cdot n + \\ &\quad k \cdot n \cdot l + (2 \cdot n - 1) \cdot C. \end{aligned}$$

Como $m \geq 1$, podemos estabelecer a seguinte relação

$$k \cdot n \leq k \cdot m \cdot n.$$

Portanto,

$$f(m, n, l) \leq k \cdot m^2 \cdot n + k \cdot n \cdot l + (2 \cdot n - 1) \cdot C. \quad \square$$

Pela fórmula encontrada, vemos que a complexidade do algoritmo é quadrática em relação ao tamanho da entrada, pois a parte de condições da tabela tem $(m \cdot n)$ entradas e a parte de ações $(l \cdot n)$ entradas.

No cálculo da complexidade de tempo, assumimos a hipótese de que a tabela não contém entradas de condição “-”. Para o caso desta afirmação não ser verdadeira, usamos a desculpa de que sempre é possível desdobrá-las. Mas o problema não se trata de poder obter ou não uma tabela sem tais entradas. Nesta situação a complexidade obtida seria em relação ao comprimento da tabela desdobrada e não da tabela original de entrada.

Podemos ver facilmente que a profundidade da recursão p do algoritmo para uma tabela trivial é $p(0) = 1$. Também podemos observar que as linhas 10 e 13, chamam recursivamente o algoritmo para tabelas com $m - 1$ condições e neste instante já estamos no primeiro nível de

recursão. Portanto, $p(m) = p(m - 1) + 1$. Resolvendo esta recorrência extremamente simples, obtemos $p(m) = m + 1$.

Assim, as tabelas mantidas pelas chamadas recursivas do algoritmo consomem espaço $2 \cdot (m + 1) \cdot t$, onde t é o tamanho de espaço necessário para guardar os vetores de uma tabela de decisão. Vimos que no esquema de representação de tabelas proposta anteriormente em 2.4.2, precisamos de duas listas que somam $(m + n)$ elementos. A constante 2 se deve ao fato do algoritmo de divisão da tabela produzir duas sub-tabelas que ficam mantidas na pilha de execução até esgotar todas as chamadas posteriores. Para excluir este fator, basta adotar a sugestão feita após a descrição do algoritmo. A estrutura global dos dados, que guarda a tabela de decisão original, consome espaço necessário para guardar a imagem da entrada dos dados.

Finalmente, examinando-se o algoritmo *Decompõe Tabela*, nota-se que a chamada recursiva da linha 13 é o último comando executado (“tail recursion”); portanto, ele pode ser substituído pelas atribuições $T \leftarrow T_N$; $v \leftarrow v_d$ e por um desvio incondicional ao início do algoritmo. A chamada recursiva da linha 10 pode ser substituída usando-se um esquema de eliminação de recursão com pilha como em [Vi 79].

Conversão da tabela e construção da STP

O algoritmo apresentado para a construção da STP é exatamente o mesmo para a conversão da tabela em linguagem algorítmica de programação. Basta trocar alguns comandos por outros:

- *coloca a tabela trivial no vértice v* da linha 7, seria trocado por *liste as ações da tabela trivial*.
- *coloca a condição C_{i_k} em v* seria trocado por *gere o comando “if C_{i_k} ”*.
- *cria o filho esquerdo v_e de v ligados com aresta “Y”* da linha 9 seria trocado por *gere a palavra “then”*.
- *cria o filho direito v_d de v ligados com aresta “N”* da linha 11 seria trocado por *gere a palavra “else”*.

Outros pequenos detalhes são triviais como colocar os delimitadores de comandos compostos e terminadores de comandos onde são necessários.

Quando o algoritmo isola uma regra, isto é, a tabela que está sendo examinada contém uma só regra, é possível juntar num único teste todas as demais condições relevantes que faltam. Outra observação sobre o algoritmo é que entre as linhas 3 e 4, poderia se colocar um comando que verifica a ambigüidade da tabela. Assim, não haveria a necessidade de fazer restrições a estas tabelas e tal problema seria detectado neste ponto. Basta contar se existe uma só regra na tabela trivial, conforme o teorema 22. Outro aspecto importante é sobre a “else rule”. A linha 7 do algoritmo verifica se $T_Y \neq \emptyset$. Para que o algoritmo trate a “else rule” basta uma pequena alteração. Em lugar dos comandos se das linhas 7 e 11 fecharem sem a parte *senão*, como está feito, coloca-se a chamada para as ações da “else rule”. Em outras palavras, se existe a “else rule”, deverá ser colocado um comando que lista as suas ações caso T_Y ou T_N sejam vazias. Esta alteração não prejudica a complexidade de tempo calculada, pois em seu lugar foi contabilizada uma chamada recursiva cuja complexidade é maior que o tempo necessário para listar as ações da “else rule”.

Se uma sub-tabela é completa e todas as suas regras executam a mesma sequência de ações, pode-se ignorar todos os testes de condição que estão nesta sub-tabela e colocar a tabela trivial [Ki-Jo 75].

2.4.5 Escolha da Condição para Teste

Como vimos, existe um número explosivo de STP's para uma mesma tabela. Em particular, o exemplo da tabela de decisão apresentada na figura 2.8, página 79, vem seguido de duas STP's que possuem diferentes número de vértices. Dois programas³ correspondentes gerados a partir da tabela teriam códigos com diferentes números de comparações, ou seja, um teria mais operações de comparação que outro. Assim, podemos comparar as diferentes STP's pelo número de comparações do código objeto. Este é um critério para a escolha de um programa dentre os possíveis. Outro critério seria o de escolher aquele que executa a tabela sobre um conjunto de dados em menos tempo que outros. Assim, temos dois critérios para escolha de um entre vários programas correspondentes a uma tabela de decisão: o que otimiza espaço e o que otimiza tempo de execução. Se os espaços ocupados pelas condições são iguais, o critério de otimização de memória corresponde exatamente à otimização do número de vértices da STP. Escolher entre programas é análogo a escolher entre STP's e, conseqüentemente, análogo a escolher entre métodos de construção de STP's.

As diferenças entre STP's de uma mesma tabela (entre os diferentes programas obtidos pela conversão de uma mesma tabela) decorrem da ordem de escolha da condição a ser testada [Da 71]. Pelo exemplo da tabela da figura 2.8, página 79, podemos observar que começar por uma linha de condição que tem símbolos “-” não é uma boa opção, pois estamos produzindo comparações de expressões nas regras que as consideram irrelevantes. Analogamente, o mesmo raciocínio leva-nos a imaginar que não é uma boa estratégia iniciar com uma linha que contenha muitos símbolos “*” e “\$”. Assim, o comando “*escolha a linha de condição C_{i_k}* ” é um sub-algoritmo de escolha. Um método elementar seria o de experimentar, uma a uma, todas as STP's possíveis, a fim de escolher a melhor delas, mas pelo seu número explosivo, esta alternativa está totalmente afastada. Restam assim, algumas heurísticas que se propõem a fazer uma boa escolha. Além disso, como já observamos, mesmo a pior STP não é uma opção condenável, pois para ela existe a garantia de que a execução de uma transação avaliará cada condição não mais que uma vez, conforme o corolário 24.

O algoritmo geral de construção de uma STP é chamado também de VSM (variable selection method) e o processo básico de divisão em sub-tabelas de refinamento [Mi 89].

Simbolos irrelevantes

Podemos observar que se a tabela não contém o símbolo “*”, “\$” e “-” nas entradas de condição, qualquer caminho da raiz a uma folha numa STP arbitrária precisa testar todas as condições. Isto é verdade, pois uma regra sem entradas irrelevantes para ser selecionada, depende dos resultados de todas as condições. Um caminho da raiz à folha reflete exatamente a seleção de uma regra. Se ocorrem símbolos “-” em alguma linha de condição para uma certa regra,

³Chamamos de programa ou código objeto ao texto em linguagem algorítmica de programação obtido pela conversão da tabela de decisão.

a princípio, o caminho da STP que seleciona esta regra, não deveria ter vértices contendo estas condições. Mas nem sempre isto é possível, embora algumas STP's produzam melhores resultados que outras. A presença de símbolos “-” é indicadora de que se deve ter cuidado na escolha da linha de condição a ser tomada para o próximo teste. O exemplo de tabela que está na página 79 mostra o efeito de um símbolo “-” na tabela sobre as STP's. Naquele mesmo exemplo, substituindo-se “-” por “\$” ou “*”, observamos o mesmo efeito, embora em outros casos a situação se modifique, dependendo de qual destes símbolos está presente. De qualquer forma, estas entradas de condição devem ser analisadas na escolha da condição para o próximo teste.

Considere agora uma tabela de decisão mostrada na figura seguinte.

	R_1	R_2	R_3
C_1	-	N	Y
C_2	-	N	Y
C_3	Y	N	-
C_4	Y	-	N

Se for escolhida a linha C_1 para o próximo teste para divisão da tabela, é claro que a regra R_1 estará em T_Y e T_N , sendo que esta regra tem um símbolo “-” na condição C_2 . Assim, se a condição C_2 for a escolhida no passo seguinte, novamente a regra R_1 estará presente nas quatro sub-tabelas obtidas de T_Y e T_N . Por outro lado, se a linha inicialmente escolhida for C_3 a regra R_1 estará somente em T_Y e assim, se a próxima condição escolhida for C_1 ou C_2 , a regra R_1 estará nas duas sub-tabelas obtidas somente de T_Y . Esta observação informal nos dá conta de que várias linhas de condição com igual número de símbolos indiferentes (“-”), escolhidas para o próximo teste, produzem STP's com número de vértices diferentes. Assim, é importante verificar não só os símbolos “-” das linhas de condição, mas os símbolos “-” que estão na mesma regra.

Outra forma de observar a influência deste critério é pensar que devemos evitar a propagação de regras que têm símbolos “-”, pois já vimos o cuidado que eles requerem. Um grande número desses símbolos, numa mesma regra, pode provocar a propagação em cadeia de tal regra, por várias sub-tabelas.

Também podemos entender que as regras com símbolos “-” representam diversas regras. É desejável que as sub-tabelas produzidas sejam as menores possível, pois intuitivamente, quanto maior a tabela, maior será o número de vértices esperados para a STP. Assim, é prudente evitar que regras complexas equivalentes a inúmeras outras regras simples sejam repetidas nas duas sub-tabelas resultantes da divisão [Hu 73]. Para isto, precisamos evitar condições que tenham símbolos “-” nestas regras. Os algoritmos de Pollack [Po 65] oferecem uma forma interessante de levar em conta simultaneamente este aspecto e o número de “-” na linha de condição que está sendo investigada como a melhor candidata para o próximo teste. Em lugar de contar o número de símbolos “-” na linha de condição, conta-se o número de regras simples representadas por aquelas que têm o símbolo “-” na linha de condição.

Balanceamento entre “Y” e “N”

Um fator considerado na literatura [Hu 73,Pr 65] para o desempate em caso de diversas linhas apresentarem as mesmas propriedades com relação ao símbolo “-” é o valor absoluto da diferença entre o número de “Y” e “N”. É claro que quanto menor esta diferença, as sub-tabelas produzidas terão tamanhos menos diferentes, de modo a propiciar árvores mais balanceadas para as STP's. Como o critério de otimização do tempo de execução do programa objeto tem a ver com o comprimento médio dos caminhos da raiz às folhas da STP, o balanceamento influi no desempenho do código associado à STP. Logo, quando se deseja obter programas priorizando o tempo de execução, é conveniente utilizar como um sub-critério de desempate o mínimo valor absoluto da diferença entre o número de “Y” e “N”. Diversos algoritmos utilizam o sub-critério da diferença máxima, em lugar da mínima, para programas com prioridade para o espaço, mas existem dúvidas [Hu 73] com relação à sua efetividade, exceto no caso em que este critério é usado para verificar se uma linha de condição contém todos “Y” ou todos “N”. O exemplo abaixo mostra a validade deste critério.

	R ₁	R ₂
C ₁	Y	N
C ₂	Y	Y

O número de vértices da STP corresponde ao número de comparações produzidas no programa traduzido. Logo, a STP à direita na figura anterior é mais econômica que a da esquerda com relação ao espaço ocupado. Portanto, é um critério de desempate viável considerar as linhas que só tenham unicamente “Y” ou unicamente “N”. Pollack [Po 65] utiliza uma forma de contagem, considerando o número de regras simples representadas por cada regra que possui entrada “Y” e “N” na linha de condição que está sendo investigada.

“Split-Count”

É desejável que escolhamos linhas com menos “-” e dentre elas, as que propiciam a menor propagação destes símbolos pelas sub-árvores. Igualmente, como fator de desempate, é desejável que escolhamos linhas com somente “Y” ou somente “N”, pois isto pode ajudar a diminuir o

número de vértices da STP, como vimos no item precedente. Num refinamento maior, como fator de desempate, podemos fazer uma escolha que propicie futuras sub-tabelas com linhas que contenham somente "Y" ou somente "N". Isto é possível se as regras com entradas "Y" e "-" na linha escolhida têm somente "Y" ou somente "N" numa outra linha de condição. Ou, de forma análoga, se as regras com entradas "N" e "-" na linha escolhida têm somente "Y" ou somente "N" numa outra linha de condição.

Para cada linha de condição C_i , considere uma ordenação no conjunto de regras, de modo que

$$\begin{aligned} C_{i1}, C_{i2}, \dots, C_{ik} &= \text{"Y"}, \\ C_{i,k+1}, C_{i,k+2}, \dots, C_{il} &= \text{"-" e} \\ C_{i,l+1}, C_{i,l+2}, \dots, C_{in} &= \text{"N"}. \end{aligned}$$

Se para $j \neq i$, $C_{j1} = \dots = C_{jl} = \text{"Y"}$ ou $C_{j1} = \dots = C_{jl} = \text{"N"}$, dizemos que C_i é *Y-complementado* por C_j . Analogamente, se $C_{j,k+1} = \dots = C_{jn} = \text{"Y"}$ ou $C_{j,k+1} = \dots = C_{jn} = \text{"N"}$, dizemos que C_i é *N-complementado* por C_j . Dizemos que C_i é *complementado* por C_j se C_i é Y-complementado ou N-complementado por C_j .

Define-se CC_{ij} , a *complementaridade* de C_i em relação a C_j [Pr 65] como sendo

$$CC_{ij} = \begin{cases} l, & \text{se } C_i \text{ é Y-complementado e não} \\ & \text{N-complementado por } C_j \\ n - k, & \text{se } C_i \text{ é N-complementado e não} \\ & \text{Y-complementado por } C_j \\ n, & \text{se } C_i \text{ é Y-complementado e} \\ & \text{N-complementado por } C_j \\ 0, & \text{se } C_i \text{ não é complementado por } C_j \end{cases}$$

O "split-count" SC_i [Po-Hi-Ha 71] para a linha de condição C_i é definido como

$$SC_i = \sum_{j=1, i \neq j}^m CC_{ij}.$$

Na verdade, busca-se a linha de condição que de uma certa forma propicia sub-tabelas futuras com maior número de linhas que tenham somente "Y" ou "N". Existem variações na literatura, onde se minimiza o "split-count" SC , e neste caso, SC contabilizaria o número de linhas que não satisfazem tal propriedade. Outras, em lugar de contabilizar as linhas, acumulam valores 0, 1, ou 2, por cada linha, caso não satisfaça, caso satisfaça com relação somente a uma das propriedades e caso satisfaça a ambas as possibilidades, respectivamente. Para simplicidade da descrição da influência do fator "split-count" na estrutura de uma STP não incluímos, entre os símbolos de entrada de condição, "\$" e "*", mas eles podem ser considerados, bastando agrupá-los convenientemente com "Y" e "N", respectivamente.

Retornando ao problema geral da escolha da condição para o próximo teste, fica a observação de que as heurísticas podem produzir a solução ótima em muitos casos, muito boas em outra considerável parte dos casos, mas sempre poderá existir situações para as quais uma outra estratégia produziria melhores resultados.

Sequência de execução

As linguagens de programação utilizadas normalmente são escritas de forma que o programador determina a ordem de execução dos comandos pela sequência em que os escreve no programa fonte. Para qualificar este fato, muitas vezes, são chamadas de linguagens “procedimentais” (procedural languages) [Hu 73]. Neste sentido, podemos afirmar que as tabelas de decisão fornecem um mecanismo parcialmente não “procedimental” para as linguagens, visto que as várias STP’s, correspondentes a uma mesma tabela, produzem diferentes sequências de testes das condições, que correspondem aos comandos de comparação que são produzidos nos programas. De um modo geral a tabela de decisão é insensível a alterações na ordem de suas regras e também das linhas da parte de condição [Re-Sa-Ya 87, Me-Ba 77, Hu 73]. Assim, fica difícil ao usuário saber, a priori, qual a ordem que um determinado algoritmo adotará para as condições de sua tabela de decisão. Se por um lado, é interessante supor que dado um conjunto de condições, é possível determinar automaticamente uma sequência de testes mais adequada para o problema economizando trabalho humano, por outro lado, pode tornar delicadas as situações em que uma ordem particular é relevante. Para estes casos, uma possível solução é fatorar uma tabela em sub-tabelas.

2.4.6 Método trivial

Pelo exposto, não é uma alternativa indecente, a política de escolher a primeira linha de condição da tabela, como a próxima a ser testada. Tem ainda a vantagem de ser uma tarefa simples e de baixíssimo custo de conversão. Basta verificar se a primeira condição é irrelevante, eliminando-a, em caso afirmativo. Tendo entradas de condição relevantes, ela é a escolhida como a próxima condição a ser testada na construção da STP e, portanto, no programa gerado. Em outras palavras, é a condição pela qual será feita a divisão da tabela. Se considerarmos que esta tarefa consiste apenas na escolha da próxima linha de condição, o custo seria constante, mas como incluímos a eliminação de linhas irrelevantes, o pior caso consiste de todas elas serem irrelevantes, o que produziria um custo computacional de tempo $O(p \cdot q)$, onde p e q são os números de condições e regras da tabela, respectivamente. Este valor para a complexidade está de acordo com o valor utilizado no estudo da complexidade do algoritmo de construção de uma STP.

2.4.7 Método I de Pollack

Esta heurística para escolha da “próxima condição” foi proposta por Pollack [Po 65] e tem o objetivo de produzir uma STP com número pequeno de vértices. Em termos de conversão de tabelas para programas, tenta produzir programas com menos comandos de comparação. Logo, este método utiliza o critério de otimização do espaço para a escolha da próxima condição.

Seja T uma tabela de decisão com condições $C_1, C_2, \dots, C_m$ e regras $R_1, R_2, \dots, R_n$ e seja RC_j , $1 \leq j \leq n$, o número de regras simples (só contêm “Y”, “N”, “\$” ou “*”) representadas pela regra R_j . RC é chamado de *Column-count* ou *Rule-count*. Assim, $RC_j = 2^t$ se a regra R_j possui t símbolos “-”. Seja DC_i , $1 \leq i \leq m$, chamado de “dash-count”, a soma dos “Column-

Count” das regras que têm entradas “-” na linha de condição C_i . Portanto,

$$DC_i = \sum_{j=1, C_{ij}=-}^n RC_j.$$

O DC_i é a quantidade a ser utilizada para a seleção da próxima condição, levando em conta o número de símbolos “-” não somente na linha, mas também o número deles nas colunas em que este mesmo símbolo ocorre na linha que está sendo investigada. Analogamente, YC_i , $1 \leq i \leq m$, chamado *Y-Count*, é a soma dos “Column-Count” correspondentes às regras com símbolos “Y” na respectiva linha C_i . Ou seja,

$$YC_i = \sum_{j=1, C_{ij}=Y}^n RC_j.$$

Analogamente, temos a seguinte expressão para *N-Count*:

$$NC_i = \sum_{j=1, C_{ij}=N}^n RC_j.$$

O valor absoluto da diferença entre YC_i e NC_i é denotado por Δ_i . Assim, $\Delta_i = |YC_i - NC_i|$.

Para exemplificar estas grandezas, vejamos a tabela seguinte, contendo somente a parte de condições.

	R_1	R_2	R_3	R_4	R_5	DC	YC	NC	Δ
C_1	Y	-	N	Y	N	4	8	9	1
C_2	N	N	-	Y	N	8	4	9	5
C_3	-	-	Y	N	N	8	8	5	3
C_4	Y	N	Y	N	N	0	12	9	3
C_5	-	N	-	-	N	16	0	5	5
C_6	Y	Y	-	-	N	12	8	1	7
RC	4	4	8	4	1				

O algoritmo de Pollack para obtenção de uma STP com menos vértices utiliza a seguinte estratégia para escolha da próxima condição: Escolha a linha de condição com mínimo DC . Havendo mais que uma linha nesta situação, escolha entre elas, a de máximo Δ . Na prática, pode-se calcular YC , NC e Δ somente para as linhas em que houver o empate no valor mínimo de DC , mas tem a desvantagem de varrer novamente estas linhas para fazer o cálculo, enquanto que se calcular estas quantidades para todas as linhas, isto pode ser feito numa única varredura pelas entradas de condição da tabela. A maneira mais econômica é guardar somente o mínimo DC com o máximo Δ . Assim, pesquisa-se, linha por linha, comparando-a com aquela que é a melhor candidata entre as que já foram analisadas. De qualquer forma, é preciso uma varredura inicial para o cálculo de RC . Pode-se observar que a complexidade desta heurística é $O(m \cdot n)$, que está de acordo com o que assumimos no estudo da complexidade do algoritmo de decomposição de tabelas de decisão. Esta heurística é encontrada com o nome de “Quick rule method” [Po 74] e se deve ao fato de que se tenta isolar uma regra o mais rápido possível, de modo

a diminuir as chances de se fazer comparações desnecessárias para elas. Isto é implementado pela escolha de linhas com menos entradas irrelevantes, considerando que poucas dessas linhas são suficientes para isolar regras com tais entradas. Esta estratégia que é utilizada no sub-algoritmo de escolha da próxima condição do algoritmo de fatorações sucessivas na versão experimental do Gerador I-M-E (capítulo 5), tende a fazer com que as regras que necessitam de menos testes sejam verificadas em primeiro lugar, embora este objetivo nem sempre seja atingido [Sp 66].

Dial [Di 70] descreve com detalhes um algoritmo para conversão de tabelas de decisão, onde a única diferença com o método I de Pollack é o sub-critério de desempate que leva em conta a diferença entre os números de “Y” e “N” nas linhas de condição.

2.4.8 Método II de Pollack

Esta heurística [Po 65, Po-Hi-Ha 71] tem por objetivo escolher uma sequência de teste para as condições de modo a produzir programas com tempo mais curto de execução. Pretende-se obter programas que sejam rápidos no processamento das transações. É encontrado na literatura como o nome de “Delayed rule method” [Po 74], em razão de poder provocar um retardo na seleção de uma regra que eventualmente necessite de poucas avaliações, cedendo lugar a outra regra que precise de mais requisitos, por uma questão de frequência com que elas ocorrem. Este método de escolha da próxima condição emprega as variáveis definidas no método I de Pollack, além das seguintes.

Seja f_j , $1 \leq j \leq n$, a frequência com que a regra R_j ocorre no universo do problema, ou seja, f_j dá uma medida da probabilidade com que a regra R_j será selecionada por uma transação na execução da tabela de decisão.

Seja WDC_i (“weighted dash-count”), $1 \leq i \leq m$, a soma dos produtos de RC_j pela frequência R_j , das regras que contêm “-” na linha de condição C_i . Isto é,

$$WDC_i = \sum_{j=1, C_{ij}='-' }^n RC_j \cdot f_j.$$

Para exemplificar, retomemos a tabela da seção anterior.

	R_1	R_2	R_3	R_4	R_5	DC	YC	NC	Δ	WDC
C_1	Y	-	N	Y	N	4	8	9	1	0,8
C_2	N	N	-	Y	N	8	4	9	5	2,4
C_3	-	-	Y	N	N	8	8	5	3	1,2
C_4	Y	N	Y	N	N	0	12	9	3	0,0
C_5	-	N	-	-	N	16	0	5	5	3,2
C_6	Y	Y	-	-	N	12	8	1	7	2,8
freq.	0,1	0,2	0,3	0,1	0,3					
RC	4	4	8	4	1					

A estratégia II de Pollack estabelece o seguinte procedimento para seleção da próxima condição: Escolha a condição com mínimo WDC . Dentre as que têm este valor igual, aquelas

com o mínimo Δ . Persistindo o empate, escolha uma linha com o mínimo DC . Os comentários feitos sobre a complexidade de tempo e armazenamento de variáveis auxiliares para o método I de Pollack servem totalmente para este método, inclusive a estimativa de custo feita para o algoritmo de fatorações sucessivas. Kulikov & Chervenchuk [Ku-Ch 84] descrevem um método muito semelhante para a escolha da próxima condição para teste.

2.4.9 Método Esquerda para Direita

Os tradutores de tabelas de decisão, de um modo geral, esperam que elas sejam não ambíguas do ponto de vista da sintaxe e também que ambigüidades semânticas não sejam cometidas pelos usuários. No caso de linhas de condição estendida, existe a exigência adicional de *restrição de exclusão* (vide 1.11) sobre as entradas de condição, requerendo que uma entrada seja disjunta de todas as outras ou igual a uma delas. Uma tabela sintaticamente correta contendo ambigüidades semânticas pode produzir resultados completamente inesperados. Considere a tabela T abaixo.

	R_1	R_2
X	≥ 2	≥ 5
Y	≤ 0	≤ 9

Tabela T

	R_1	R_2
$X \geq 2$	Y	*
$X \geq 5$	*	Y
$Y \leq 0$	Y	*
$Y \leq 9$	*	Y

Tabela T'

Claramente as linhas de condição da tabela T são estendidas e contrariam a restrição de exclusão. Por exemplo, para os conjuntos da primeira linha, $\{2, 3, \dots\} \cap \{5, 6, \dots\} = \{5, 6, \dots\}$. Convertendo esta tabela em limitada conforme 1.12 obtemos a tabela T' da figura anterior.

Existe um conjunto de valores para X e Y ($X \geq 5$ e $Y \leq 0$) que satisfaz tanto à regra R_1 como a R_2 em T , embora estruturalmente a tabela T' não contenha ambigüidades. A tabela limitada T' mostra um problema na sua interpretação. Na regra R_1 existe a informação de que se $X \geq 2$, então obrigatoriamente $X < 5$, o que não é verdade, e a regra R_2 diz que se $X \geq 5$ então obrigatoriamente $X < 2$, o que também não é verdade. Problemas semelhantes ocorrem nas linhas de condição que envolvem a variável Y em T' . Este erro incorporado na tabela limitada se deve ao fato de termos convertido uma tabela estendida, transformando as linhas de condição estendidas que não satisfazem à restrição de exclusão usando um método de conversão (vide 1.12) proposto somente para tabelas que satisfazem aquela restrição.

É difícil saber qual a ordem que um determinado conversor seguirá na escolha das condições. Supondo que o usuário propôs a tabela acima ao conversor, sem a necessária cautela na utilização de linhas estendidas e como na tabela limitada correspondente, todas as linhas têm exatamente os mesmos símbolos nas entradas de condição, apenas na ordem invertida, provavelmente as condições serão escolhidas para teste na sequência em que aparecem na tabela. Assim, insistindo na impropriedade do usuário na elaboração da tabela, se o conversor escolher a condição $X \geq 2$ antes de $X \geq 5$, a regra R_2 nunca será executada. A STP da figura 2.10 mostra mais claramente o problema:

Figura 2.10: STP para T'

Pelo esquema da STP, vemos que a sub-árvore direita da raiz jamais será percorrida com possibilidades de satisfazer a regra R_2 . A execução da STP acima, seleciona somente a regra R_1 e quando $X \geq 5$ e $0 < Y \leq 9$ nenhuma das regras é selecionada. Assim, em casos de violação à restrição de exclusão, resultados inesperados podem ser produzidos na execução de uma tabela de decisão. Além disso, os resultados são difíceis de serem analisados em razão de não se saber qual o mecanismo de escolha da próxima condição utilizado pelo algoritmo. Do ponto de vista teórico, o problema fica por conta da falta de cuidado na construção da tabela. Mas do ponto de vista prático, seria altamente positivo se além de estipular todas as restrições na confecção de uma tabela, fosse possível adiantar o comportamento do algoritmo no caso de infração aos requisitos de boa construção.

H. Takaoka [Ta 76] apresentou uma heurística que, com relação à ambigüidade, deveria seguir a estratégia de selecionar sempre a regra mais à esquerda da tabela de decisão e aceitar linhas de condição sem restrições, mas infelizmente esta não é uma tarefa fácil, pois o algoritmo deve ser capaz de reconstruir cada linha de condição estendida, particionando os conjuntos de valores nela presentes em sub-conjuntos disjuntos como propõem King & Johnson [Ki-Jo 75]. O problema se agrava se as entradas de condição contêm expressões com variáveis cujos valores são desconhecidos em tempo de tradução e são permitidas várias formas de expressão de conjuntos, pois neste caso não é possível analisar as entradas de condição e determinar o tipo de dependência existente entre elas. Como vemos, a heurística enfrenta dificuldades quando as linhas de condição estendidas violam a restrição de exclusão como no último exemplo. Mesmo a tentativa de superar o problema com uma análise dos valores das entradas de condição seria barrada pela simples substituição das constantes por variáveis nas entradas de condição. De qualquer modo existe uma indicação de como o algoritmo escolheu as regras e condições na solução da tabela.

Deixando de lado o objetivo de resolver o problema da construção de tabelas de decisão sem seguir as regras de “boa construção”, vamos apresentar uma heurística que chamaremos *Esquerda para Direita* que é uma pequena modificação da heurística de Takaoka com a finalidade de torná-la uma mistura dos algoritmos I e II de Pollack e que é implementada no pré-processador de tabelas de decisão (capítulo 4).

Esta heurística de escolha da próxima condição leva em conta os dois fatores abaixo, sem prejuízo de outros que podem ser considerados e incorporados como sub-critérios de desempate.

- O número de “–” na linha de condição.
- O número de entradas relevantes (“Y” e “N”).

Com respeito ao tempo de execução do programa objeto, a heurística leva em conta o critério geral de que é desejável que as regras mais freqüentes sejam testadas prioritariamente, pois assim, torna-se menos possível a realização de testes desnecessários para elas. Neste enfoque, considerando-se que nem sempre se conhece a freqüência exata das regras, este método supõe que as regras estão ordenadas da esquerda para a direita em ordem decrescente de freqüências, sem uma preocupação pelos valores exatos de cada uma delas. Assim, o usuário simplesmente coloca as regras na ordem decrescente da freqüência com que elas ocorrem. Temos, portanto, uma forma para seleção da próxima condição, que é simples e que pode ser considerada como uma mistura dos aspectos enfocados pelos métodos que otimizam espaço e tempo de execução do programa objeto (“Quick rule method” e “Delayed rule method”). Esta heurística seleciona a linha de condição que tenha o melhor resultado, segundo os fatores acima citados (mínimo número de “–” e, em caso de empate, máximo número de entradas relevantes) *dentre aquelas que possuem uma entrada relevante (“Y” ou “N”) na primeira regra*. Se a sub-tabela que está sendo investigada pelo algoritmo só tem uma regra e ela não possui entradas relevantes, a sub-tabela é reduzida a uma tabela trivial, conforme o teorema 29. Se nenhuma entrada é relevante na primeira regra e a tabela não pode ser reduzida a uma tabela trivial, a heurística seleciona a melhor linha, dentro dos fatores acima considerados. A pergunta neste ponto é se existe uma sub-tabela não redutível a uma tabela trivial, cuja primeira regra não tem entradas relevantes. Parece difícil, mas para mostrar que esta possibilidade existe, vamos retomar a tabela da figura 1.5(b) que apresenta o símbolo “*” utilizado como eliminador de ambigüidade com as regras em outra ordem.

	R ₁	R ₂
C ₁	*	Y
C ₂	Y	–

Fatorando esta tabela pela linha de condição C₂, teríamos a árvore de fatoração da figura seguinte com sub-tabelas T_Y e T_N, sendo que na primeira, a regra mais a esquerda não tem entrada relevante.

Geralmente é possível apresentar um exemplo para o qual uma determinada heurística não apresenta bons resultados. A tabela do exemplo anterior, que não é comum em virtude da utilização do símbolo “*” como eliminador de ambigüidade semanticamente válida e não como entrada implícita, é um caso em que a presente heurística faz uma escolha que não é a melhor alternativa.

Para a seleção da próxima condição, a heurística *esquerda para a direita* pesquisaria cada linha de condição, como está representada pela tabela abaixo. Nela, usamos a seguinte notação:

- o índice i refere-se à condição sendo investigada;
- o índice k é o índice da melhor condição até o momento;
- L é o número de “–” na linha de condição C_i ;
- J é o número de “Y” e “N” na linha de condição C_i ;
- Min é o número de “–” na linha de condição C_k ;
- Max é o número de “Y” e “N” nas linhas de condição C_k .

Linha C_i é irrelevante	Y	N	N	N	N	N
$C_{k1} = \text{“Y” ou “N”}$	–	N	N	N	Y	Y
$C_{i1} = \text{“Y” ou “N”}$	*	Y	N	N	Y	Y
$L < Min$	–	–	Y	N	Y	N
$L = Min$	–	–	*	Y	*	Y
$J > Max$	–	–	–	Y	–	Y
Remove Linha C_i	1					
$k := i$		1	1	1	1	1
$Max := J$		2	2	2	2	2
$Min := L$		3	3		3	

Nesse método de escolha podem ser considerados outros fatores, como o número de símbolos “–” nas colunas, para diminuir o seu desdobramento pelas sub-tabelas e outros que estão presentes nas heurísticas, por exemplo, de Pollack. Todas as propriedades citadas anteriormente são mantidas, conservando o aspecto básico deste método, que é tornar estes fatores determinantes na escolha da próxima condição, como subsídio para desempate entre as linhas de condição que têm entrada relevante na regra mais à esquerda. A complexidade de tempo desta heurística também está de acordo com a estimativa feita no algoritmo de fatorações sucessivas.

2.4.10 Heurísticas baseadas na Teoria da Informação

Foram propostas heurísticas que determinam a próxima condição de teste com base no conceito de Entropia da Teoria da Informação [Ga-Ra 73, Sh 71, Sh 74]. Elas consistem basicamente em calcular o valor da entropia associada a cada linha de condição e escolher aquela que proporciona o maior resultado. Levando em conta que a entropia dá uma medida da incerteza do problema

[Mo 82], que pode ser associado ao grande número de possibilidades da tabela inicial, as sub-tabelas têm as suas incertezas diminuídas. Como o objetivo final de uma STP é atingir uma única regra, quando então não temos mais incerteza, as heurísticas escolhem as condições que contribuem com a diminuição mais drástica das possibilidades nas sub-tabelas seguintes, o que intuitivamente é conseguido fazendo com que entropias maiores sejam relacionadas com linhas que possuem maior número de entradas de condição relevantes. A título de exemplo, vamos apresentar a heurística de Shwayder [Sh 71, Sh 74] que tenta minimizar o número de operações do programa objeto. Seja T uma tabela de decisão com condições $C_1, C_2, \dots, C_m$, regras $R_1, R_2, \dots, R_n$, com frequências relativas $f_1, f_2, \dots, f_n$. Para cada linha de condição C_i , calculam-se os seguintes valores:

$$\begin{aligned} F_i &= 1 - \sum_{j=1, C_{ij}=-}^n f_j \\ F_i^Y &= \sum_{j=1, C_{ij}=Y}^n f_j / F_i \\ F_i^N &= 1 - F_i^Y. \end{aligned}$$

F_i representa a soma das frequências das regras que não possuem o símbolo “-” na linha de condição C_i . F_i^Y e F_i^N representam as somas das frequências das regras que têm entradas “Y” e “N”, respectivamente na linha de condição C_i em relação a P_i . A entropia E_i da linha C_i é definida como

$$E_i = -F_i \cdot (F_i^Y \cdot \log F_i^Y + F_i^N \cdot \log F_i^N).$$

A próxima condição a ser testada é C_k tal que $E_k = \max\{E_i, 1 \leq i \leq m\}$ [Sh 71].

Shwayder [Sh 74] propôs incorporar informações sobre o custo de teste das condições à proposta anterior o que levaria a selecionar a próxima condição pela entropia por unidade de custo.

Ganapaty & Rajaraman [Ga-Ra 73], além de apresentarem uma heurística baseada na teoria da informação, utilizam a entropia para estimar a qualidade da STP produzida, através de uma comparação com uma cota inferior, embora uma medida exata deste valor seja difícil, pois não conhecemos o custo da STP ótima.

2.4.11 Heurísticas com “Look-Ahead”

As heurísticas analisam localmente as tabelas de decisão e, com base em suas propriedades, escolhem uma linha de condição para o próximo teste sem levar em conta a influência desta escolha nos custos das sub-árvores resultantes, o que acaba por produzir uma STP que nem sempre é ótima com relação a algum critério. Como uma solução que testa todas as possibilidades é altamente custosa, pode-se adotar uma solução intermediária, escolhendo a próxima condição com base nas informações locais e também de seus futuros desdobramentos, mas limitando a profundidade do “look-ahead”. Sethi & Chatterjee [Se-Ch 80] propuseram uma heurística enfocando a minimização do tempo de execução do código gerado com profundidade 1, ou seja, para escolher a próxima condição, são analisadas a tabela corrente e as sub-tabelas produzidas pela sua divisão por cada uma das condições.

Os autores introduzem uma forma de calcular o custo esperado para uma tabela de decisão T com condições $C_1, C_2, \dots, C_m$, com custos de avaliação $c_1, c_2, \dots, c_m$ e regras $R_1, R_2, \dots, R_n$, com frequências $f_1, f_2, \dots, f_n$, como sendo

$$\bar{c}_T = f_T \cdot \sum_{j=1}^{m-1} c_j + f_l \cdot c_m,$$

onde f_l é a soma das frequências das regras que têm o maior número de entradas relevantes, o que intuitivamente, pode corresponder a caminhos mais longos nas STP's e f_T é a soma das frequências de todas as regras de T e vale 1 se T é a tabela inicial.

A heurística considera as sub-tabelas T_Y^i e T_N^i , $1 \leq i \leq m$, que, à medida do possível, devem ter as suas regras intercaladas priorizando aquelas que produzem maior diminuição de custo e as condições ordenadas de forma que regras mais freqüentes tenham entradas de condição relevantes entre as linhas C_1 e C_{m-1} . A condição escolhida para o próximo teste é C_k tal que

$$\bar{c}_{T_Y^k} + \bar{c}_{T_N^k} + f_T \cdot c_k = \min\{\bar{c}_{T_Y^i} + \bar{c}_{T_N^i} + f_T \cdot c_i, 1 \leq i \leq m\}.$$

2.4.12 Outras heurísticas

Diversas estratégias para escolha da próxima condição para teste no algoritmo de fatorações sucessivas são encontradas na literatura, além das mencionadas neste texto. Uma dessas é utilizar o conceito de *atividade de uma condição ou variável*, como às vezes as condições são chamadas nos artigos [Mo-Th-Go 80] [Mi 89].

Seja T uma tabela de decisão completa com 2^m regras simples. De acordo com a nossa notação para divisão de tabela de decisão,

$$T_{x_1 \dots x_{j-1} x_{j+1} \dots x_m}^{1 \dots j-1 \ j+1 \dots m},$$

onde $x_i = \text{"Y"}$ ou "N" , é uma tabela obtida pela divisão sucessiva de por $C_1, C_2, \dots, C_{j-1}, C_{j+1}, \dots, C_m$, tomando-se a sub-tabela que agrupa as regras pelo valor x_i em cada operação de divisão por C_i . Uma sequência $x = (x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_m)$ é *ativa* [Mi 89] com relação a C_j se

$$(T_{x_1 \dots x_{j-1} x_{j+1} \dots x_m}^{1 \dots j-1 \ j+1 \dots m})_Y^j \neq (T_{x_1 \dots x_{j-1} x_{j+1} \dots x_m}^{1 \dots j-1 \ j+1 \dots m})_N^j,$$

e seja

$$a_T(C_j) = \sum_{x \text{ ativo}} f(T_{x_1 \dots x_{j-1} x_{j+1} \dots x_m}^{1 \dots j-1 \ j+1 \dots m}) / f(T)$$

a *atividade* da condição C_j na tabela T , onde $f(T)$ é a soma das frequências das regras de T . Se T é a tabela inicial $f(T) = 1$. Intuitivamente, $a_T(C_j)$ dá uma medida da relevância da condição C_j na tabela T enquanto $1 - a_T(C_j)$ dá uma medida de sua não essencialidade. Define-se

$$perda_T(C_j) = c_j \cdot (1 - a_T(C_j)),$$

onde c_j é o custo de avaliação da condição C_j . A $perda_T(C_j)$ dá uma medida do custo da não essencialidade da condição C_j na tabela T . Assim, a heurística que escolhe a condição C_j tal que

$$perda_T(C_j) = \min\{perda_T(C_k), 1 \leq k \leq m\},$$

pode produzir uma boa STP para a tabela de decisão T . Segundo Miyakawa [Mi 89], o algoritmo de fatorações sucessivas com esta heurística tem complexidades de tempo e espaço de $O(m^2 \cdot 2^m)$ e $O(m \cdot 2^m)$, respectivamente, onde neste caso, $n = 2^m$.

Miyakawa [Mi 89] define outras funções a partir da atividade de uma condição que satisfazem a certas propriedades, chamando-as de *potenciais* e propõe outros critérios heurísticos para a escolha da próxima condição.

Zaki [Za 84] propôs uma heurística simples para a seleção da próxima condição. Primeiramente, deve-se combinar as regras. Em caso da necessidade de escolher entre diferentes combinações, deve-se executar a operação que maximiza o produto $c_k \cdot f_{ij}$, onde c_k é o custo de avaliação da condição em cuja linha a combinação produz uma entrada indiferente e f_{ij} é a soma das frequências das regras que estão sendo combinadas. Existindo linhas de condição que tenham somente entradas relevantes, uma delas deve ser escolhida. Na sua ausência, calcula-se para cada condição C_i a seguinte expressão

$$G_i = \frac{1}{c_i} \sum_{\substack{j=1, \dots, n \\ C_{ij}=Y, N}} f_j$$

e escolhe-se a linha de condição que maximiza o valor de G_i entre as que não contém o símbolo “-”. Se todas as linhas contêm este símbolo então escolhe-se uma linha que maximiza o valor de G_i .

Press [Pr 65] propôs uma heurística para escolha da próxima condição com o objetivo de minimizar o número de desvios no programa gerado que seleciona uma linha que não tenha entradas “-”, de preferência uma que tenha todas as entradas “Y” ou todas “N”. Se todas as linhas têm uma entrada “-”, escolhe-se aquela que maximiza o “split-count” (vide 2.4.5).

Rabin [Ra 71] e ChinnaSwamy [Ch 79] propuseram a utilização de “Karnaugh Maps” [Mi 65] para a conversão de tabelas de decisão.

2.4.13 Outros Aspectos do Algoritmo de Fatorações Sucessivas

Existem heurísticas que combinam regras [Da 75, Za 84] antes de efetuar a escolha da linha de condição a ser passada para o algoritmo de fatorações sucessivas para o próximo teste. Isto requer uma estratégia, porque em geral é difícil determinar se uma sequência de operações de combinações produz uma tabela ótima segundo algum critério.

Embora as linhas de condição estendidas possam ser convertidas em limitadas, é possível, sem grande esforço, adaptar o algoritmo de fatorações sucessivas para tratar diretamente estas condições [Wo-Ha 71]. Neste caso, em lugar de uma STP binária, teríamos uma árvore k -ária, onde k é o número máximo de estados distintos nas linhas de condição estendidas da tabela [Ki-Jo 75]. Assim, em cada vértice da STP que contém uma condição C_i estendida com k estados distintos teríamos k arestas saindo deste vértice.

Outro aspecto que pode ser abordado é o compartilhamento de ações comuns por diversas regras. Existem ações que estão presentes em mais de uma regra. Podemos tratar este problema de várias formas, mas uma possível solução é isolar em cada vértice da STP as ações que são executadas para todas as regras presentes na sub-tabela associada ao vértice e executá-las já neste ponto [Me-Ba 77].

2.5 Algoritmos Ótimos

Existem algoritmos que produzem a STP ótima para uma tabela de decisão, através de duas técnicas: Busca Exaustiva e Programação Dinâmica. Vamos selecionar entre as várias propostas existentes, uma de cada técnica para exemplificar estes algoritmos. Nesta parte, vamos nos referir à sequência de ações de uma regra como sendo uma única ação.

2.5.1 Método de Busca Exaustiva Parcial

Os algoritmos que empregam este método para a obtenção da STP ótima correspondente a uma tabela de decisão, fazem uma enumeração exaustiva de todas as STP's possíveis, na esperança de que o espaço de busca possa ser significativamente diminuído por critérios de custo ("branch and bound method"). Reinwald & Soland [Re-So 66, Re-So 67] descreveram um algoritmo para a obtenção da árvore ótima pelo critério de tempo de execução e outro para minimização do espaço ocupado pela árvore de decisão. Estes dois algoritmos particionam o conjunto das STP's em sub-conjuntos de árvores que começam pelo mesmo teste de condição e calculam um limite inferior para cada bloco da partição [Br-Re 75a]. A busca é direcionada na esperança de que a melhor árvore se encontre no bloco de menor cota inferior. Como isto nem sempre ocorre, o arrependimento leva a analisar outros blocos. O algoritmo 2 de Verhelst [Ve 72], apresentado inicialmente como um algoritmo ótimo, teve a sua capacidade contestada por King & Jonhson [Ki-Jo 74], mas restringindo a sua aplicação a tabelas com condições independentes e assumindo que todo desdobramento de regras complexas em simples pode ser feita pela divisão das frequências das regras complexas, o algoritmo fornece uma forma de utilização desta técnica baseada em cotas superior e inferior, calculadas para cada bloco da partição, sendo a cota superior obtida pela aplicação de uma boa heurística. A comparação desses dois valores pode eliminar definitivamente do espaço de busca alguns blocos da partição cuja cota inferior seja maior que o custo da STP obtida pela heurística.

Apresentaremos o algoritmo de Breitbart & Reiter [Br-Re 75b] que exige uma tabela que especifique todas as 2^m regras e, segundo os próprios autores, trata-se de um refinamento do algoritmo de Reinwald & Soland [Re-So 66], que minimiza o tempo de execução do programa gerado. Ele produz várias STP's, simultaneamente, na esperança que nem todas tenham que ser concluídas. O algoritmo original soluciona o problema da árvore de decisão ótima, independentemente de estar associada a uma tabela de decisão. Algumas pequenas mudanças são feitas para adaptá-lo ao problema específico de tabelas de decisão e à nomenclatura utilizada até o momento.

Seja uma STP S correspondente a uma tabela de decisão T . O custo H de execução de S corresponde ao comprimento médio ponderado dos caminhos da raiz às folhas de S . Isto pode ser escrito da seguinte forma:

$$H = \sum_{v_j \in VS} p_j \cdot c_j$$

onde c_j é o custo para o teste da condição que está presente no vértice v_j , p_j é a soma das probabilidades das regras que estão presentes na sub-tabela correspondente à STP com raiz em v_j e VS é o conjunto de vértices de S . Assim, temos associado a cada vértice v_j um custo parcial $h_j = p_j \cdot c_j$, de modo que, tendo a árvore S e os custos parciais de seus vértices, podemos

obter o custo H de toda a STP .

O algoritmo produz conjuntos de STP 's parciais, um nível de cada vez, que foram chamadas de PET 's ("partially constructed evaluation tree") e em razão disto, precisamos descrever um modo de representar uma floresta de PET 's.

Considere inicialmente uma tabela de decisão T completa com três condições C_1, C_2 e C_3 e que contém somente regras simples. Pelo visto em 2.4.1, existem 9 STP 's distintas equivalentes a T . Na figura seguinte estão as três STP 's que contêm C_1 na raiz.

Analogamente, existem outras três para C_2 na raiz e mais três para C_3 na raiz. Nesta descrição não usamos as tabelas triviais nas folhas, para simplicidade. Vamos utilizar a estrutura de dados proposta por Breitbart & Reiter [Br-Re 75b] para representar uma floresta de PET 's e que é sugerida pela figura 2.11. Ela mostra a floresta de STP 's correspondentes à tabela T acima citada. Na verdade, esta floresta representa mais que as 9 STP 's distintas; ela representa também as outras três STP 's isomorfas, num total de 12 STP 's.

Figura 2.11: Representação de uma floresta

Cada vértice da floresta contém os seguintes campos:

- Tabela: A sub-tabela de decisão associada ao vértice.

- Cond: A condição que está no vértice.
- Esq: O apontador para seu filho esquerdo.
- Dir: O apontador para seu filho direito.
- Prox: O apontador para o próximo irmão.
- Custo: O custo parcial do vértice.

Convencionamos os seguintes valores para um apontador dentro desta estrutura:

0, o apontador pertence a uma folha de uma STP;
 < 0 , se o apontador está indefinido;
 $x > 0$, aponta para o vértice x da floresta

Esta representação mantém toda a floresta, mas precisamos conseguir destacar uma única *PET* dentro da coleção. Os autores descrevem um algoritmo recursivo de nome *MinCost* que de posse de uma floresta e de uma raiz v , devolve uma *PET* de custo mínimo com raiz em v . Os nós da árvore construída por *MinCost* contêm apontadores *LS* e *RS* para os filhos esquerdo e direito respectivamente e um outro apontador *PetNode* para um vértice da floresta, constituindo-se numa árvore de apontadores para os nós da floresta. No que segue, vamos denotar um campo C de um vértice p por $c(p)$ e vamos considerar que todas as 2^m regras estão especificadas na tabela de entrada como regras simples.

O algoritmo constrói a floresta começando com diversas *PET*'s, cada uma delas possuindo somente a raiz. Cada raiz contém uma das condições da tabela de decisão e também o seu respectivo custo parcial. Como vimos em 1.15, em cada vértice v de uma *STP*, temos uma sub-tabela associada, correspondente à sub-árvore enraizada em v . Assim, vamos denotar por T_v a sub-tabela associada ao vértice v da floresta construída pelo algoritmo, em lugar de $Tabela(v)$. Numa iteração genérica seja S a *PET* de custo mínimo na floresta com vértices terminais que não correspondem a tabelas triviais $v_1, v_2, \dots, v_k$. Divide-se a sub-tabela associada T_{v_i} pela linha de condição $Cond(v_i)$, obtendo as sub-tabelas T_Y e T_N . Se T_Y contiver uma única ação, as condições não precisam ser avaliadas, e assim, cria-se um vértice como o filho esquerdo de v_i e coloca-se nele a ação correspondente, constituindo-se num nó terminal que não será mais expandido no desenvolvimento da floresta. Caso contrário, para cada condição relevante de T_Y , cria-se um novo vértice como filho esquerdo de v_i . Faz-se o mesmo para a sub-tabela T_N . Repete-se este processo até que a *PET* de custo mínimo não tenha nenhum vértice terminal a ser expandido, ou seja, todas as folhas contêm ações. A seguir está descrito o algoritmo que constrói a *PET* de custo mínimo dada a floresta de *PET*'s.

Algoritmo: MinCost

Importação: Apontador a para um nó da floresta que é a raiz da *PET* ótima que se deseja construir.

Exportação: O Custo c mínimo e o apontador p para a raiz da *PET* construída.

Temporários: Inteiros min e v e apontadores q , g , e r .

Início

aloca(p); PetNode(p) := a ; c := custo(a);


```

se Esq(a) <= 0      (* Vértice a Não tem filho esquerdo *)
  então LS(p) := 0;
senão
  min := ∞; g := 0; q := Esq(a);
  enquanto q > 0 faça      (* Procura a melhor sub-árvore esquerda *)
    MinCost(q,v,r);
    se v < min
      então min := v;
      se g > 0 então libera(g);
      g := r;
    senão libera(r);
    q:=Prox(q);
  c := c + min;
  LS(p) := g;
se Dir(a) <= 0      (* Vértice a Não tem filho direito *)
  então RS(p) := 0;
senão
  min := ∞; g := 0; q := Dir(a);
  enquanto q > 0 faça      (* Procura a melhor sub-árvore direita *)
    MinCost(q,v,r);
    se v < min
      então min := v;
      se g > 0 então libera(g);
      g := r;
    senão libera(r);
    q:=Prox(q);
  c := c + min;
  RS(p) := g;
fim

```

Suponhamos um vértice genérico a da floresta de PET's. Para cada filho esquerdo q de a é executado o algoritmo *MinCost* que devolve a melhor sub-árvore com raiz em q , retornando o seu custo na variável v e sua raiz no apontador r . O apontador g indica o filho esquerdo que produziu o melhor resultado entre os que já foram consultados e min o seu custo associado. Isto também é feito com os filhos direitos de a .

Embora o algoritmo possa trabalhar sem problemas com a representação da floresta apresentada, existe a necessidade de se calcular o custo das STP's com as várias raízes possíveis utilizando o algoritmo *MinCost* e entre elas escolher a de menor custo geral. Para facilitar este trabalho, é conveniente introduzir um vértice fictício v_0 , com custo parcial zero sem filho direito e cujos filhos esquerdos são os vértices $v_1, v_2, \dots, v_m$, cada um contendo uma das condições da tabela de decisão. Esta providência faz com que uma única chamada a *MinCost* produza o retorno da melhor PET construída até o momento, como pode ser observado no esquema do algoritmo que segue, onde denotaremos por f_Y e f_N a soma das frequências das regras em T_Y e T_N , respectivamente. Também vamos assumir que a operação de alocação de um vértice faz com que os seus apontadores conttenham valores negativos.

Algoritmo: Produz uma floresta de PET's e uma STP de custo mínimo através de chamada ao procedimento MinCost

Importação: Tabela de Decisão T , completa com condições $C_1, C_2, \dots, C_m$, custos de avaliação $c_1, c_2, \dots, c_m$, regras $R_1, R_2, \dots, R_n$ ($n = 2^m$), que executam ações $A_1, A_2, \dots, A_k$ e com frequências $f_1, f_2, \dots, f_n$.

Exportação: O Custo c mínimo e o apontador p para a raiz da STP construída.

Início

se T contém uma única ação A_1 então

 produza uma STP com um único vértice contendo a ação A_1 e pare

aloca(x_i), $0 \leq i \leq m$;

custo(x_0) := 0; Dir(x_0) := 0; Esq(x_0) := x_1 ;

Prox(x_i) := x_{i+1} , $1 \leq i < m$;

Custo(x_i) := c_i , $1 \leq i \leq m$;

repita

 MinCost(x_0, c, p);

 seja S a árvore de custo mínimo com raiz em p e folhas $v_1, v_2, \dots, v_k$.

 para cada v_i faça

 se $T(v_i)$ não é trivial então

 Divida T_{v_i} pela condição em Cond(v_i), obtendo T_Y e T_N

 se T_Y contém uma única ação A_p

 então aloca(x); Esq(v_i) := x ; Cond(x) := 0; T_x := A_p ;

 Esq(x) := 0; Dir(x) := 0; Custo(x) := 0;

 senão sejam $C_{i_1}, C_{i_2}, \dots, C_{i_l}$ as condições de T_Y

 aloca(x_{i_j}), $1 \leq j \leq l$

 Cond(x_{i_j}) := C_{i_j} , $1 \leq j \leq l$;

$T_{x_{i_j}}$:= T_Y , $1 \leq j \leq l$;

 Custo(x_{i_j}) := $c_{i_j} \cdot f_Y$, $1 \leq j \leq l$;

 Esq(v_i) := x_{i_1} ;

 Prox(x_{i_j}) := $x_{i_{j+1}}$, $1 \leq j < l$;

 se T_N contém uma única ação A_p

 então aloca(x); Dir(v_i) := x ; Cond(x) := 0; T_x := A_p

 Esq(x) := 0; Dir(x) := 0; Custo(x) := 0;

 senão sejam $C_{i_1}, C_{i_2}, \dots, C_{i_l}$ as condições de T_N

 aloca(x_{i_j}), $1 \leq j \leq l$

 Cond(x_{i_j}) := C_{i_j} , $1 \leq j \leq l$;

$T_{x_{i_j}}$:= T_N , $1 \leq j \leq l$;

 Custo(x_{i_j}) := $c_{i_j} \cdot f_N$, $1 \leq j \leq l$;

 Dir(v_i) := x_{i_1} ;

 Prox(x_{i_j}) := $x_{i_{j+1}}$, $1 \leq j < l$;

 até que nenhum novo nó tenha sido alocado

fim

O que garante que o custo c da árvore produzida é mínimo sem a necessidade de concluir as PET's que ficaram incompletas é que qualquer outra PET S' tem custo $c' \geq c$. Além disso, por

construção, c' é uma cota inferior para o custo de todas as árvores que podem ser obtidas com a expansão de S' . Embora o algoritmo produza a STP ótima ele é totalmente inviável para tabelas com mais que 6 condições, pois no pior caso, o número de PET's construídos em cada nível x da floresta é $(m - (x - 1))^{2^{x-1}}$ [Br-Re 75b]. O exemplo do pior caso para o algoritmo é uma tabela simples. Basta considerar uma tabela completa sem entradas irrelevantes, com custos de avaliação iguais para as condições, iguais frequências de ocorrência das regras e diferentes ações para cada regra. Esta tabela seria convertida numa STP ótima pela mais simples das heurísticas que seleciona seqüencialmente as condições para a divisão da tabela, mas o algoritmo apresentado produziria todas as STP's possíveis, pois qualquer PET parcial tem custo inferior a qualquer PET completa. O cálculo da cota inferior de Reinwald & Soland [Re-So 66] é melhor que o apresentado neste algoritmo, levando em conta não somente a condição que está sendo testada, mas também uma porcentagem do custo das outras condições e armazena em cada nó a cota inferior de todas as STP's enraizadas em cada vértice.

2.5.2 Método de Programação Dinâmica

Diversos trabalhos foram desenvolvidos para a obtenção da STP ótima utilizando a técnica de programação dinâmica [Ba 73, Sc-Se 76, Mi 85, Ma-Mo 78]. Schumacher & Sevcik [Sc-Se 76] apresentam o algoritmo, associando uma tabela completa a um *k-cubo* [Bo-Mu 77, Ya 71]. Vamos descrever o algoritmo de Bayes [Ba 73] com ligeira modificação, para adaptá-lo à notação do nosso trabalho.

Os algoritmos que utilizam essa técnica encontram a solução construindo soluções ótimas para sub-problemas cada vez maiores do problema inicial, usando soluções ótimas já conhecidas para sub-problemas menores. Assim, a construção da solução começa pelos sub-problemas elementares, aumentando progressivamente o tamanho deles de modo que nenhum sub-problema ocorre antes de seus próprios sub-problemas [Ga 72].

Consideremos novamente uma tabela de decisão com condições C_1, C_2 , e C_3 , completa e sem entradas irrelevantes. O conjunto de STP's possíveis, como vimos no método de Busca Exaustiva, pode ser representada pela floresta da figura 2.11. De cima para baixo, temos inicialmente um vértice com todas as condições indefinidas, no nível abaixo um conjunto de vértices com uma condição definida e as demais indefinidas. Assim, sucessivamente, até que, por último, naquela figura, temos um conjunto de vértices onde apenas uma condição está indefinida e que deverá ser testada nesses nós. Se completarmos a figura com a representação das tabelas triviais, as folhas serão vértices com todas as condições definidas. A programação dinâmica consiste em construir a árvore a partir das folhas, mas aquela representação não é adequada, em razão do grande número de cálculos repetidos que deveríamos efetuar. Se rotularmos cada vértice v daquela figura com uma sequência de três símbolos (x_1, x_2, x_3) , tal que

$$\begin{aligned} x_i &= '-' && \text{se a condição está indefinida} \\ x_i &= 'Y' && \text{se a condição foi testada e tem valor verdadeiro} \\ x_i &= 'N' && \text{se a condição foi testada e tem valor falso} \end{aligned}$$

obtemos a seguinte figura que mostra a floresta da figura 2.11 com os vértices rotulados segundo o critério acima.

Nesta representação podemos ver que existem vários vértices com o mesmo rótulo e que os vértices com rótulos contendo o mesmo número de símbolos “-” estão no mesmo nível. Nesta seção, vamos dizer que os vértices estão no nível i , $0 \leq i \leq m$ se possuem i símbolos “-” nos seus rótulos. Isto corresponde a enumerar de 0 a m de baixo para cima as cotas em que os vértices estão desenhados, representando a floresta de STP's de uma tabela com m condições. Vamos apresentar a mesma floresta sem repetições de vértices na figura 2.12. Basta enumerar sistematicamente o vértice com m símbolos “-” no nível m e sucessivamente, todos os vértices com i destes símbolos no i -ésimo nível, finalizando com o nível zero onde são colocados todos os vértices sem nenhum destes símbolos, que correspondem às regras da tabela de decisão com todas as condições já avaliadas. Como o grafo resultante possui vértices com todas as combinações possíveis dos símbolos “Y”, “N” e “-”, ele contém 3^m vértices. No caso do exemplo, como $m = 3$, temos $3^3 = 27$ vértices. As arestas são produzidas sistematicamente da seguinte forma: Seja v um vértice do nível i , $1 \leq i \leq m$, com rótulo $(x_1, x_2, \dots, x_m)$. Para cada $x_j = “-”$, produza uma aresta ligando v com o vértice $v_Y = (x_1, \dots, x_{j-1}, Y, x_{j+1}, \dots, x_m)$ e outra aresta ligando v com $v_N = (x_1, \dots, x_{j-1}, N, x_{j+1}, \dots, x_m)$. Cada um destes pares (v_Y, v_N) é chamado um *par de predecessores de v* [Ba 73] e a condição C_j é chamada *condição de v com relação a (v_Y, v_N)* . Com este processo sistemático, constrói-se o grafo que chamaremos de *grafo de computação* para uma tabela de decisão com qualquer número de condições. Este mesmo grafo Martelli & Montanari [Ma-Mo 78] chamam de grafo “and/or” associado à tabela de decisão e utilizam noções de grafos no desenvolvimento do algoritmo.

O algoritmo de programação dinâmica opera sobre o grafo de computação, inicializando os valores dos vértices do nível zero, que são chamados de *vértices iniciais* [Ba 73], e iterativamente sobre os vértices de cada nível 1 a m , encontrando neles uma orientação sobre quais vértices do nível anterior devem ser consultados.

Em cada nó v do grafo precisamos armazenar:

- a frequência f_v da STP de custo mínimo enraizada em v , que é a soma das frequências das regras que estão na sub-tabela correspondente a esta STP.
- o custo c_v da STP de custo mínimo enraizada em v .
- a ação A_v que deve ser executada, se a STP enraizada em v é uma tabela trivial; contém o valor zero para indicar que a sub-tabela correspondente à STP enraizada em v tem mais que uma regra com ações distintas e, portanto, o teste da condição é necessário; Tanto o

Figura 2.12: Grafo de Computação do Algoritmo de Programação Dinâmica

algoritmo de Schumacher & Sevcik [Sc-Se 76] como o de Bayes [Ba 73] usam também o valor -1 para indicar que uma regra é logicamente impossível.

- um apontador P_v que indica qual o par de predecessores do vértice v minimiza o custo das STP's enraizadas em v . Basta indicar um índice k tal que $x_k = "-"$ em v e $x_k \neq "-"$ para o par de predecessores escolhido.

Seja T uma tabela de decisão limitada, completa, com condições $C_1, \dots, C_m$, com custos de avaliação $c_1, \dots, c_m$ e Regras $R_1, \dots, R_n$, com frequências $f_1, \dots, f_n$ que executam ações $A_1, A_2, \dots, A_k$ (várias regras podem executar um mesmo conjunto de ações). Se T não é completa, é possível expandi-la, indicando regras logicamente impossíveis com a ação A_{-1} . Havendo indicação da "else rule" deve-se desdobrá-la em regras simples com ação A_{k+1} [Sc-Se 76].

O algoritmo começa colocando em cada um dos 2^m nós iniciais a frequência e ação da regra correspondente e anulando os seus custo e apontador. Depois disso, efetua a computação para todos os vértices do nível 1 e, sucessivamente, para todos do nível 2, 3 ..., até atingir o vértice do nível m .

Para cada vértice v genérico dos níveis 1 a m sejam (v_Y, v_N) um de seus pares de predecessores e C_k a condição de v com relação a este par com custo de avaliação c_k . O algoritmo computa:


```

 $p := p_{v_Y} + p_{v_N}$ 
se  $A_{v_Y} = A_{v_N} > 0$  então
     $c := c_{v_Y} + c_{v_N}$ 
     $A_v := A_{v_Y}$ 
senão se  $A_{v_Y} < 0$  ou  $A_{v_N} < 0$  então
     $c := c_{v_Y} + c_{v_N}$ 
     $A_v := \max\{A_{v_Y}, A_{v_N}\}$ 
senão
     $c := c_{v_Y} + c_{v_N} + p \cdot c_k$ 
     $A_v := 0$ 

```

Repetindo este cálculo para cada par de predecessores de v , o algoritmo escolhe aquele que proporcionou o menor valor para c e os resultados computados para aquele par devem atualizar os campos do nó v . O campo P_v recebe o índice da condição de v em relação ao par de predecessores escolhido.

Quando o algoritmo executa a computação para o vértice v do nível m teremos a STP de custo c_v mínimo que pode ser recuperada percorrendo recursivamente o grafo de computação, começando pelo vértice v do nível m e seguindo sucessivamente pelos pares de predecessores escolhidos.

O rótulo dos vértices são usados numa implementação interpretando a sequência de símbolos do rótulo como um número na representação ternária.

Em cada nível i do grafo de computação são produzidos todos os rótulos contendo i símbolos “-”. Isto significa que para cada combinação destes i símbolos dentro da sequência de m símbolos que constituem o rótulo existem todas as possibilidades para “Y” e “N” nas $m - i$ posições, ou seja, temos $2^{m-i} \cdot \binom{m}{i}$ nós no nível i do grafo de computação do algoritmo. Como já dissemos, o número total de vértices é 3^m que equivale a [Ba 73]

$$\sum_0^m 2^{m-i} \cdot \binom{m}{i} = \sum_0^m 2^i \cdot \binom{m}{i} = 3^m.$$

De imediato temos que a complexidade de espaço do algoritmo é da ordem do número de vértices do grafo, ou seja $O(3^m)$. A implementação de Bayes ocupa $12 \cdot 3^m$ bytes. Cada vértice do nível i , $1 \leq i \leq m$, tem i pares de predecessores, pois por construção, no nível i , os rótulos dos vértices possuem i símbolos “-” e para cada um deles existe um par de predecessores e duas arestas. Logo o número de arestas do grafo de computação é [Ma-Mo 78].

$$\begin{aligned} 2 \cdot \sum_1^m i \cdot 2^{m-i} \cdot \binom{m}{i} &= 2 \cdot \sum_0^m i \cdot 2^{m-i} \cdot \binom{m}{i} = \\ 2 \cdot \sum_0^m (m-i) \cdot 2^i \cdot \binom{m}{i} &= 2 \cdot m \cdot 3^{m-1}. \end{aligned}$$

Em cada nó dos níveis 1 a m é executada uma operação de mínimo. Portanto, o número destas operações executadas pelo algoritmo somado o número de inicializações efetuadas sobre os vértices do nível zero produz o número de operações correspondentes ao número de vértices do grafo de computação, ou seja, a complexidade de tempo medida desta forma é $O(3^m)$ [Le 78].

Se levarmos em conta que para cada operação de mínimo num vértice v , o algoritmo efetua computações sobre cada um de seus pares predecessores, o tempo de computação pode ser calculado pelo número total de pares de predecessores dos vértices do nível 1 a m que é [Ba 73]

$$\sum_{i=1}^m i \cdot 2^{m-i} \cdot \binom{m}{i} = \sum_{i=0}^m (m-i) \cdot 2^i \cdot \binom{m}{i} = m \cdot 3^{m-1}.$$

Somando este número ao número de computações efetuadas para inicializar os 2^m vértices do nível zero, obtemos um número de operações mais realístico que é

$$2^m + m \cdot 3^{m-1}.$$

Vamos apresentar um esquema do algoritmo, substituindo os símbolos “-”, “Y” e “N” nos rótulos pelos números 0, 1 e 2, respectivamente. A representação do grafo de computação exige 3^m vértices, que pode ser implementada, por exemplo, por uma matriz de apontadores. Vamos usar uma função $g_3(x)$ que calcula o inteiro correspondente à sequência x na representação ternária e retorna o endereço do vértice correspondente.

Algoritmo: Constrói uma STP ótima para uma tabela de decisão.

Importação: Tabela de decisão T com condições $C_1, \dots, C_m$ com custos de avaliação $c_1, \dots, c_m$, regras $R_1, \dots, R_{2^m}$ com frequências $f_1, \dots, f_n$ e que executam as ações $A_1, \dots, A_k$.

Exportação: Grafo de computação contendo a STP ótima.

início

1. para cada regra R_j faça
 2. seja $x = (C_{1j}, C_{2j}, \dots, C_{mj})$ e A_k a ação da regra R_j
 3. $z := g_3(x)$; $f_z := f_j$; $c_z := 0$; $A_z := k$; $P_z := 0$;
 4. para $i := 1, 2, \dots, m$ faça
 5. para cada sequência $x = (x_1, x_2, \dots, x_m)$ de m elementos com i zeros faça
 6. para cada combinação de 1's e 2's nas $m - i$ posições que não contém zero faça
 7. $z := g_3(x)$; $z_c := \infty$;
 8. para cada x_j da sequência x faça
 9. se $x_j = 0$ então
 10. $z_Y := g_3(x_1, \dots, x_{j-1}, 1, x_{j+1}, \dots, x_m)$
 11. $z_N := g_3(x_1, \dots, x_{j-1}, 2, x_{j+1}, \dots, x_m)$
 12. $f := f_{z_Y} + f_{z_N}$;
 13. se $A_{z_Y} = A_{z_N}$ e $A_{z_Y} > 0$ então
 14. $A := A_{z_Y}$; $c := c_{z_Y} + c_{z_N}$;
 15. senão se $A_{z_Y} < 0$ ou $A_{z_N} < 0$ então
 16. $A := \max\{A_{z_Y}, A_{z_N}\}$; $c := c_{z_Y} + c_{z_N}$;
 17. senão
 18. $A := 0$; $c := c_{z_Y} + c_{z_N} + f \cdot c_j$;
 19. se $c < z_c$ então
 20. $z_c := c$; $z_f := f$; $z_A := A$; $z_P := j$;
- fim**

O comando com o número 5 calcula todas as combinações de i zeros numa sequência de m elementos. Bayes [Ba 73] utilizou o algoritmo de Chase [Ch 70] na sua implementação. O comando da linha 6 precisa calcular todas as sequências de números 1 e 2 para as $m-i$ posições da sequência que não contém zero. Vamos apresentar abaixo, rapidamente os algoritmos *Combina* e *Zero_Um* que executam estas tarefas, sem maiores explicações. Eles recebem na primeira iteração, no parâmetro *Primeiro*, a informação de que é a primeira sequência a ser produzida. Nas demais iterações, o algoritmo *Combina* recebe uma sequência com i números zero nas posições exatas em que foram produzidos na iteração anterior e o algoritmo *Zero_Um* recebe exatamente a sequência da iteração anterior. Eles recebem também o número m que indica o tamanho da sequência. O Algoritmo *Combina* recebe o parâmetro i indicando o número de elementos zero que deve estar na sequência produzida. No caso do algoritmo *Zero_Um*, o parâmetro i corresponde ao número de elementos diferentes de zero que a sequência deve conter, mas esta informação é extraída da própria sequência que ele recebe e que deve conter exatamente i números diferentes de zero, inclusive na primeira iteração, informando em que posições da sequência estes elementos devem ficar. Os algoritmos, após a produção de todas as sequências, devolvem o valor *verdadeiro* no parâmetro *Fim*, significando que a última sequência foi produzida na iteração anterior. A sequência é armazenada no vetor S da posição 1 a m , sendo que a posição zero deve ser reservada para área de trabalho dos algoritmos.

Algoritmo: Combina

Importação: Sequência S , inteiros i e m e a variável lógica *Primeiro*.

Exportação: Sequência S e a variável lógica *Fim*.

Temporários: Variáveis inteiras j, k, l .

início

$Fim := false;$

se *Primeiro* então

para cada $j := 1, 2, \dots, i$ faça $s[j] := 0;$

para cada $j := i + 1, i + 2, \dots, m$ faça $s[j] := 1;$

senão

$j := m; \quad s[0] := 1;$

enquanto $s[j] = 0$ faça $j := j - 1;$

se $m - j \geq i$ então $Fim := true;$

senão

$k := m - j + 1;$

enquanto $s[j] \neq 0$ faça $j := j - 1;$

$s[j] := 1;$

para cada $l := j + 1, j + 2, \dots, j + k$ faça $s[l] := 0;$

para cada $l := j + k + 1, j + k + 2, \dots, m$ faça $s[l] := 1;$

fim

Algoritmo: Zero_Um

Importação: Sequência S , inteiros i e m e a variável lógica *Primeiro*.

Exportação: Sequência S e a variável lógica *Fim*.

Temporário: Variável inteira j .

início

$Fim := false;$

se *Primeiro* então

para cada $j := 1, 2, \dots, m$ faça

se $s[j] \neq 0$ então $s[j] := 1;$

senão

$j := m; \quad s[0] := 1;$

enquanto $s[j] \neq 1$ faça

se $s[j] = 2$ então $s[j] := 1;$

$j := j - 1;$

se $j = 0$ então $Fim := true;$

senão

$s[j] := 2;$

fim

Em cada vértice $z = (x_1, x_2, \dots, x_m)$ do grafo de computação temos um conjunto de STP's enraizadas em z correspondentes à sub-tabela T_z que pode ser obtida dividindo-se sucessivamente a tabela original pelas condições C_i , tais que $x_i \neq "-"$ e tomando-se a sub-tabela pelo valor de x_i . A frequência f_z do vértice z é a soma das frequências das regras presentes na tabela T_z e a ação A_z corresponde às ações das regras em T_z . Logo, para pares de predecessores (z_Y, z_N) e (z'_Y, z'_N) de z

$$\begin{aligned} f_{z_Y} + f_{z_N} &= f_{z'_Y} + f_{z'_N} \\ A(z_Y, z_N) &= A(z'_Y, z'_N), \end{aligned}$$

onde $A(z_Y, z_N)$ representa o valor da ação do vértice z em relação ao par de predecessores (z_Y, z_N) .

Como consequência imediata desta observação, a linha 12 do algoritmo de programação dinâmica precisa ser executada somente para o primeiro par de predecessores de um vértice z . O cálculo da ação associada ao vértice também pode ser modificado para tirar proveito da mesma observação.

Capítulo 3

Mais poder de Notação às Tabelas de Decisão

Embora as tabelas de decisão sejam um importante mecanismo de especificação de decisões complexas, existem situações que não são facilmente descritas dentro do formato tradicional de tabelas de decisão. King & Johnson [Ki-Jo 75] propuseram que a parte de descrição de uma linha de condição deveria admitir testes de pertinência a intervalos.

3.1 Conjuntos de Valores

Muitos sistemas de tabelas de decisão impõem restrições sobre o formato das condições. É comum na literatura, por exemplo, a imposição de que uma condição numa tabela de decisão deve ser composta de uma variável e um literal com um operador relacional entre eles. Esta forma de atuar sobre as condições tem por objetivo facilitar uma análise mais aprofundada das condições, como a verificação da dependência ou não das condições, observando se elas operam sobre a mesma variável. De posse de tal informação, seria possível alertar o usuário sobre a possibilidade de algumas ambigüidades serem apenas aparentes. De qualquer forma, como esta informação não permite uma atuação completa sobre o problema da ambigüidade, visto que existem condições implicitamente dependentes que não se apresentam sob a forma de variáveis comuns nas diferentes condições, mas são fruto da semântica particular do problema, preferimos verificar as ambigüidades estruturais, ou seja, que constam das entradas de condição da tabela de decisão e supor que as condições são independentes. Caso não sejam, cabe ao usuário estar consciente de que o tratamento automático será o de condições independentes, no que se refere à ambigüidade.

Usaremos o termo *formas usuais* de escrever as condições para denotar:

- Linhas de condição limitadas onde a parte de descrição é composta somente de uma variável seguida de um operador relacional e de outra variável ou constante. Pode também ser composta de uma única variável do tipo lógico. Estas formas podem ser precedidas pelo operador (“not”). Em outras palavras, trata-se de expressões lógicas que não tem conectivos “and” e “or”.
- Linhas de condição estendidas onde a parte das entradas tem somente o nome de uma

variável ou constante ou estes precedidos por um dos operadores $<, \leq, >, \geq, \sim$. Admite também todas as entradas irrelevantes.

Nas tabelas de decisão que serão tratadas neste capítulo, as linhas de condição serão mais genéricas. As condições limitadas das tabelas de decisão, podem ser qualquer expressão lógica, inclusive usando os conectivos “and” e/ou “or”. As condições estendidas possuem na parte de descrição o nome de uma variável ou expressão e nas entradas de condição, uma outra variável, constante ou um *conjunto de valores*, definido por um nome ou por um dos operadores $<, \leq, >, \geq, \sim$, seguidos de uma variável ou constante. Ainda podem conter os símbolos de entrada irrelevantes “-”, “\$” e “*”.

X	0	1	2	-1	≤ -2	≥ 3
Y	1	1	~ 1	~ 1	-	-
Z	X	W	$>W$	$<X$	*	\$
T	C ₁	C ₂	C ₃	C ₄	C ₅	C ₆

Nas linhas de condição estendidas, as entradas de condição que não são irrelevantes podem ser de dois tipos: Elas especificam o nome de um conjunto de valores ou a própria entrada define um conjunto. Este conjunto definido na própria entrada, pode ser o nome de uma variável ou constante, portanto, um conjunto unitário, ou um conjunto definido pelos operadores acima relacionados seguidos de uma constante ou nome de uma variável e neste caso, o conjunto pode ser teoricamente infinito.

Como o símbolo “-” é usado para indicação especial de entrada indiferente (“-” = “Y” + “N”), os números negativos nas linhas estendidas devem vir seguidos de um número ou variável sem espaços em branco. Nestas linhas, as entradas “-” têm significado muito parecido ao seu uso nas linhas limitadas. Indica que a linha é irrelevante para a regra e qualquer uma das entradas presentes na linha pode ser satisfeita. O símbolo “*” indica que a variável de condição não satisfaz a nenhuma das especificações relevantes daquela linha e que isto é implicitamente determinado pelas outras condições. O símbolo “\$”, analogamente, diz que a variável da condição satisfaz à especificação relevante presente na linha e que isto é implicitamente determinado pelas outras condições. A utilização do símbolo “\$” nas entradas estendidas só tem sentido nas linhas que definem uma única entrada de condição relevante. Isto decorre imediatamente da restrição de exclusão. Todos os três símbolos indicam a irrelevância da variável de condição para aquela regra, portanto, torna o teste supérfluo, mas a interpretação deles é diferente, como o é também nas linhas limitadas.

Os nomes nas entradas de condição das linhas estendidas podem ser variáveis ou nomes de conjuntos, ou seja, em lugar de colocar a própria definição do conjunto na entrada, coloca-se o seu nome. Deste modo, conjuntos complexos podem ser especificados nas entradas de condição através de um nome, visto que torna-se impraticável colocar nas entradas de condição uma definição exaustivamente longa de um conjunto. Para que isto seja possível é necessário definir previamente os conjuntos. Assim, a tabela de decisão tradicional ganha uma nova parte adicional: a de *definição de conjuntos de valores* [Se-Ma 88].

C_1	$\{1, 2, \dots, 25\}$			
C_2	$\{'c'..'e'\} \cup \{'h'..'m'\}$			
C_3	$\{x \in \mathcal{R} : -1.0 < x < 1.5\}$			
C_4	$\{3, 6, 7, 10\} \cup \{15..30\}$			
$X = Y$	Y	N	–	N
Z	< 0	0	C_1	> 25
W	C_2	'Z'	'W'	C_2
R	C_3	2.5	> 4.0	< -2.0

Na figura anterior, a parte de definição de conjuntos está mostrada na parte superior da tabela antes da parte de condições e é interpretada por linhas. No exemplo, estão sendo definidos quatro conjuntos de valores, C_1 , C_2 , C_3 e C_4 , respectivamente, cujos elementos são os descritos no lado direito da linha correspondente.

Para uma utilização prática para os pré-processadores de tabelas de decisão, é preciso adotar uma sintaxe para a definição de conjuntos e também do restante da tabela. Isto será feito no próximo capítulo. Por enquanto, fica expandida a estrutura da tabela de decisão que passa a ter uma seção de definição de conjuntos que, quando necessário serão descritos em notação matemática. Na tabela acima, são definidos os conjuntos C_1 , C_2 , C_3 e C_4 em linguagem matemática. Na parte de condições, a terceira regra testa a pertinência da variável Z ao conjunto C_1 . Analogamente, as duas primeiras regras testam a pertinência da variável W ao conjunto C_2 e assim com outros conjuntos e variáveis. Note que a tabela de decisão acima é mista, pois apresenta linhas de condição limitada (a primeira) e estendidas.

Em alguns casos o uso de conjuntos facilita muito a descrição. Suponha o conjunto $C = \{1, 3, 5, 10, 11, 15, 20, 30\}$ e que desejamos executar uma ação A se, $x \in C$ e $y = 0$. Uma possível solução para a tabela tradicional seria a seguinte, considerando as formas usuais de entradas de condição estendidas.

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
x	1	3	5	10	11	15	20	30
$y = 0$	Y	Y	Y	Y	Y	Y	Y	Y
A	1	1	1	1	1	1	1	1

Outra forma mais trabalhosa seria escrever na forma de uma sequência de condições limitadas, como mostrada em seguida.

	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆	R ₇	R ₈
$x = 1$	Y	*	*	*	*	*	*	*
$x = 3$	*	Y	*	*	*	*	*	*
$x = 5$	*	*	Y	*	*	*	*	*
$x = 10$	*	*	*	Y	*	*	*	*
$x = 11$	*	*	*	*	Y	*	*	*
$x = 15$	*	*	*	*	*	Y	*	*
$x = 20$	*	*	*	*	*	*	Y	*
$x = 30$	*	*	*	*	*	*	*	Y
$y = 0$	Y	Y	Y	Y	Y	Y	Y	Y
A	1	1	1	1	1	1	1	1

Os asteriscos nas entradas de condição mostram bem o fato de que as condições são dependentes e que a satisfação de uma condição determina implicitamente a falsidade das demais. Este exaustivo trabalho pode ser substituído simplesmente pela tabela de decisão que está na figura 3.1(a).

C	{1,3,5,10,11,15,20,30}
x	C
$y = 0$	Y
A	1

(a)

C	$(\{1..31\} \times \{1,3,5,7,8,10,12\}) \cup$ $(\{1..30\} \times \{4,6,9,11\}) \cup (\{1..28\} \times \{2\})$	
	R ₁	ELSE
Dia:Mes	C	
Válida	1	
Inválida		1

(b)

Figura 3.1: Tabelas com definição de conjuntos

Este caso supra citado mostra a substituição de um trabalho exaustivo por um mecanismo de definição de conjuntos. Existem problemas que podem ser especificados na forma de conjuntos e que uma solução na forma de tabelas tradicionais não é conseguida com facilidade. Seja C o conjunto de duplas (D,M) correspondentes aos dias e meses válidos do ano. Vamos denotar, na linguagem matemática, o produto cartesiano entre dois conjuntos por " $\times$ ". A tabela de decisão que verifica a validade de uma data qualquer ignorando-se os anos bissextos seria a que está na figura 3.1(b).

Parece difícil construir uma tabela com linhas de condição estendidas, para todas as condições sem violar a restrição de exclusão sobre a variável *Dia* e dentro das *formas usuais* de escrever as linhas de condição. Segue na figura 3.2 uma solução com encadeamento de tabelas, inclusive para exemplificar uma tal utilização.

	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆	R ₇	R ₈	R ₉	R ₁₀	R ₁₁	R ₁₂	ELSE
Mês	1	2	3	4	5	6	7	8	9	10	11	12	
goto M30				1		1			1		1		
goto M31	1		1		1		1	1		1		1	
goto Fev		1											
Inválida													1

	R ₁	ELSE
Dia $\geq$ 1	Y	
Dia $\leq$ 30	Y	
Válida	1	
Inválida		1

Tabela M30

	R ₁	ELSE
Dia $\geq$ 1	Y	
Dia $\leq$ 31	Y	
Válida	1	
Inválida		1

Tabela M31

	R ₁	ELSE
Dia $\geq$ 1	Y	
Dia $\leq$ 28	Y	
Válida	1	
Inválida		1

Tabela Fev

Figura 3.2: Tabelas encadeadas para verificação de data

Na figura 3.3 segue uma solução para o problema com apenas uma única tabela, utilizando linhas limitadas para a variável de condição *Dia*.

	R ₁	R ₂	R ₃	R ₄	R ₅	R ₆	R ₇	R ₈	R ₉	R ₁₀	R ₁₁	R ₁₂	ELSE
Mês	1	2	3	4	5	6	7	8	9	10	11	12	
Dia $\geq$ 1	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
Dia $\leq$ 31	Y	\$	Y	\$	Y	\$	Y	Y	\$	Y	Y	Y	
Dia $\leq$ 30	-	\$	-	Y	-	Y	-	-	Y	-	Y	-	
Dia $\leq$ 28	-	Y	-	-	-	-	-	-	-	-	-	-	
Válida	1	1	1	1	1	1	1	1	1	1	1	1	
Inválida													1

Figura 3.3: Tabela limitada para verificação de data

Suponha uma tabela para verificar os atributos de um registro de nome "Reg", através de um campo denominado "Atr". Vamos supor que a implementação da solução para o problema indique que o referido campo contém o valor "-1" se o registro está vazio, "0" se parcialmente ocupado e "1" se estiver cheio. Uma tabela simples para isto seria a da figura 3.4(a), omitindo outras linhas de condição e de ação.

Um usuário contemplando esta tabela isoladamente estaria diante de um enigma com relação

	R ₁	R ₂	R ₃
Reg.Atr	-1	0	1

(a)

Vazio	{-1}		
SemiCheio	{0}		
Cheio	{1}		
	R ₁	R ₂	R ₃
Reg.Atr	Vazio	SemiCheio	Cheio

(b)

Figura 3.4: Tabelas que verificam conteúdo de registro

a seu significado. Considere agora a tabela equivalente da figura 3.4(b), com a utilização de conjuntos.

Considerando que o usuário esteja familiarizado com a estrutura de uma tabela de decisão, vemos que fica mais fácil entender o significado da regras, se usarmos a notação de conjuntos com nomes sugestivos. É claro que o mesmo efeito seria possível considerando-se os nomes dados aos conjuntos como nomes de variáveis ou constantes do programa com o valor que eles recebem na tabela. De qualquer forma, é um instrumento adicional no sentido da legibilidade. A utilização de conjuntos de valores em linhas de condição estendida está implementada num gerador de aplicações chamado “LDT” de Ervino Marussi [Se-Ma 88] e foi implementado também no nosso sistema (Capítulo 4).

3.2 Ações Comuns de Preparação

Vamos incorporar à tabela de decisão uma outra seção, onde são especificadas ações que devem ser executadas antes de iniciar a fase de decisão. Ela é introduzida pensando em situações em que os dados à disposição precisam ser modificados antes da decisão. Para um exemplo genérico, suponha que os dados de que dispomos são $x_1, x_2, \dots, x_n$ e que a decisão se baseia sobre uma função $f(x_1, \dots, x_n)$. Se f é simples, a própria cadeia que define a função pode ser colocada na linha de condição da tabela, mas ela pode ser algo mais complexa. A princípio, isto poderia ser contornado, do ponto de vista do programa do usuário, colocando estas ações comuns fora da tabela, em pontos que a antecedem. Mas, considerando que tais ações são de interesse e uso exclusivo da tabela, a sua especificação dentro de uma parte inicial da tabela pode tornar-se mais interessante. Além disso, esta providência vem de encontro às necessidades da parte final deste trabalho.

Capítulo 4

Um Pré-Processador de Tabelas de Decisão

Esta parte do trabalho trata de aspectos de implementação de um pré-processador de tabelas de decisão, gerando código “Pascal” ou “C”. Trata-se de um pré-processador e não somente um tradutor, no sentido de que ele faz mais do que simplesmente receber uma tabela de decisão e convertê-la em comandos das linguagens citadas. O pré-processador chamado “Decis”, recebe como entrada um arquivo, contendo um programa fonte em Pascal ou C, onde a parte de decisões está especificada através de tabelas de decisão, podendo conter um grande número delas, inclusive com chamadas encadeadas, com ou sem retorno. O programa traduz estas tabelas para código Pascal ou C e transforma as tabelas originais em comentários da linguagem. O código traduzido de cada tabela segue imediatamente após a sua ocorrência como comentário.

4.1 Funcionamento

Um *grupo de decisão* é o conjunto de tabelas relacionadas entre si com encadeamentos entre chamadas e as especificações de transformações invocadas por estas tabelas. Entende-se por *transformação* uma sequência de ações (comandos) que devem ser executadas. Esta sequência recebe um nome, que é o *nome da transformação*, e as tabelas podem especificar as ações através de referências a estes nomes. Os elementos de um grupo de decisão devem estar contíguos, começando pela tabela que inicia a sequência de execução dentro do grupo, seguida pelas demais tabelas, estas em qualquer ordem e, por último, as especificações das transformações do próprio grupo. Um grupo de decisão pode estar localizado em qualquer lugar do arquivo de entrada e ali ficarão os comandos gerados a partir dele. Como podemos observar, uma tabela isoladamente é um caso particular de um grupo de decisão que contém um único elemento que é a tabela.

Sendo o Decis um pré-processador que analisa tabelas no meio de código em Pascal ou C, precisamos identificar os comandos que são destinados a ele, para que atue somente sobre os objetos que desejamos. Ele os identifica porque são compostos pelo símbolo “%” seguido de uma palavra ou outro símbolo, dependendo do comando e podem estar em qualquer posição da linha. Estes comandos são transformados em comentários e assim inseridos no arquivo gerado. Como trata-se de uma linguagem de comandos, não deve ser colocado mais de um comando numa linha. A tabela de decisão possui uma estrutura própria para uma interpretação visual

por linhas e colunas, assim, as suas especificações são potencialmente encerradas linha a linha e, se continuadas numa outra seguinte, devem conter o sinal de existência de continuação “%+”.

O símbolo “%-” indica que é uma linha de comentário, podendo estar em qualquer coluna, mas toda a linha é mantida no código gerado como comentário.

A implementação do Decis usa o método de fatorações sucessivas para a conversão de tabelas em programas; para a seleção da próxima condição, utiliza a estratégia “Esquerda para Direita”. O Decis verifica as ambigüidades sintáticas e detectando-as não efetua a tradução, produzindo apenas uma cópia do arquivo de entrada com mensagens indicativas de erro. Admite entradas de condição implícitas “\$” e “*” e a “else rule”. Quanto a esta última, o critério que vale é o mais geral, consistindo na livre especificação para ações arbitrárias como de outras regras quaisquer e, no caso de ausência, é invocada uma regra vazia. Permite também que regras especificadas na tabela tenham ações vazias, ou seja, se selecionadas nenhuma ação é executada. As tabelas mistas são convertidas em limitadas, antes de iniciar a tradução. A tabela de decisão, originalmente fornecida pelo usuário, é representada por quatro matrizes, para as suas seguintes partes: descrição das condições e ações e entradas de condição e ação. Estas estruturas são globais e a manipulação de tabelas e sub-tabelas é feita utilizando a estrutura proposta na seção 2.4.2, a propósito do algoritmo de divisão de uma tabela por uma linha de condição.

Para obtermos as informações a serem colocadas na estrutura de dados globais acima, é preciso analisar o arquivo de entrada. A análise é feita por um autômato de estados finitos [Se-Me 83,Ah-Ho-UI 74,Wi 76], que percorre a cadeia de entrada, analisando-a e construindo a imagem da tabela de decisão na memória. A definição de conjuntos de valores que permite construções recursivas é analisada por um analisador de precedência de operadores [Ah-Se-UI 86]. Durante a análise das expressões que definem os conjuntos de valores, o analisador constrói um predicado que o representa, mas a variável cuja pertinência será verificada somente será conhecida mais adiante, na parte de condições. Assim, é construída uma tabela de conjuntos de nome *Tab_Set* que armazena o seu nome e a cadeia com o predicado correspondente à expressão que define o conjunto, deixando um indicador na cadeia onde o nome da variável deve ser inserido posteriormente. Como podem ocorrer conjuntos de n-uplas, é preciso que tal indicador assinale também qual elemento da n-upla será inserido naquele ponto. Como os componentes de uma n-upla constituem uma sequência ordenada, o indicador deve armazenar somente um número indicando qual a ordem do componente que deve ser inserido. Normalmente, na maior parte das situações, os conjuntos são de elementos com um só componente e assim o indicador acima, neste caso, contém “#1”. Para as linhas estendidas, o analisador constrói uma tabela de nome *Tab_Var_Cond* cujo conteúdo é substituído a cada nova linha e que guarda separadamente cada componente da n-upla que está presente na parte de descrição de uma linha de condição. Na análise de uma entrada de condição que é o nome de um conjunto que está no índice *i* da *Tab_Set*, o analisador obtém a expressão condicional a ser testada juntando o conteúdo da tabela *Tab_Var_Cond* à cadeia que está na *i*-ésima posição da *Tab_Set*.

4.2 Linguagem e Limites de um Grupo de Decisão

O arquivo de entrada deve conter um dos comandos

```
%generate C
```



```
%generate Pascal
```

que indicará a linguagem fonte a ser gerada a partir das tabelas de decisão. Deve estar presente em qualquer ponto do arquivo, antes da ocorrência da primeira tabela.

Um grupo de decisão começa quando o pré-processador, percorrendo o arquivo de entrada, fora de um grupo, encontra uma tabela de decisão. Ele se encerra com o final do arquivo de entrada, ou encontrando o comando

```
%end transf
```

indicando que encerrou todas as transformações do grupo. Mesmo que não haja transformações no grupo, se é desejo especificar diferentes grupos de decisão num arquivo, é necessário colocar este comando para separá-los. Um grupo é composto de um conjunto de tabelas de decisão, sendo que a primeira delas determina o ponto de entrada do fluxo de execução. Assim, a ordem das tabelas é irrelevante dentro de um grupo, a menos da primeira tabela. A especificação da primeira transformação determina o fim das tabelas do grupo e o início das transformações, sendo que estas podem vir em qualquer ordem.

4.3 Especificações de uma Tabela de Decisão

Já vimos teoricamente como é a estrutura de uma tabela de decisão, mas precisamos dizer como é feita a especificação desta estrutura para o pré-processador. Em seguida faremos esta descrição na ordem em que as partes devem ser especificadas. A tabela do exemplo 4.3 mostra como se escreve uma tabela de decisão para um problema imaginário de acordo com as convenções do Decis.

```
%
%generate C
%
%Decision table Exemplo
%
%preparation
    Calcula_Salário()
    Calcula_Faltas()
    Calcula_Avaliação()
%
%Sets
%
    Faixa1      | <10000
    Faixa2      | >=10000
    Nota1       | 0..5
    Nota2       | 6..10
%
%Conditions    |
```


%	Salário		Faixa1	Faixa1	Faixa2	Faixa2
	Avaliação		Nota1	Nota2	Nota1	Nota2
	Faltas <= 10		Y	-	Y	Y
%	%Actions					
%	Salário=P1*Salário				1	
	Salário=P2*Salário		1			1
	Salário=P3*Salário			1		
	Congela(Func)				2	
	Promove(Func)		2	2		2
	Else_Rule(Func)					1
%	%end table					
%						

Exemplo 4.3.

4.3.1 Nome da Tabela de Decisão

Cada tabela deve ter obrigatoriamente um nome, segundo as regras de formação de rótulos da linguagem fonte. Este nome é utilizado como “label” no código gerado para efeito de desvios se for necessário. Tal rótulo é inserido entre a tabela de decisão e a sua tradução. O nome da tabela aparece no comando inicial de uma tabela de decisão e é através dele que o pré-processador detecta o início de uma nova tabela e de um novo grupo, se for este o caso.

```
%decision table Nome_Tabela
```

4.3.2 Ações Comuns de Preparação (opcional)

Esta parte serve para listar um conjunto de ações que devem ser executadas antes de iniciar o processo de decisão. O comando

```
%preparation
```

identifica esta seção na tabela. Nas linhas seguintes devem ser relacionadas as ações de preparação propriamente ditas. Cada uma delas pode ser um comando qualquer da linguagem fonte ou uma chamada para uma transformação, que é feita através de um comando %exec seguido do seu nome. O seguinte esquema dá a forma geral desta parte.

```
%preparation
  %exec Nome.transf (a)
  Comandos da linguagem fonte (b)
```

A intenção dessa parte é produzir a execução de um trecho do programa gerado, destinado a preparar alguns dados para as condições ou os conjuntos de valores a serem especificados na tabela de decisão. Na tabela que está no exemplo 4.3, as ações especificadas são comandos da linguagem fonte que referenciam nomes de funções, enquadrando-se no tipo (b) do esquema acima. Imaginemos que a tabela de decisão é executada para cada funcionário da empresa e que é lido o arquivo contendo informações sobre o vencimento padrão, horas extras, adicionais de insalubridade e de horário noturno e gratificação de função, de modo que é necessário juntar vários dados para se obter o salário de cada funcionário. Analogamente, para a avaliação e número de faltas, são necessárias rotinas para obtê-los.

No esquema acima, a opção (a) trata de chamadas a transformações feitas através do comando `%exec` seguido do seu nome. Estas são sequências de comandos da linguagem fonte definidas após o término da especificação das tabelas de um grupo, como será visto com mais detalhes em 4.4. Este tipo de especificação na parte de ações de preparação produz um desvio para o trecho do programa que define a transformação e, depois de executar os seus comandos, um desvio de volta para o próximo comando da tabela de decisão.

Na opção (b), deve-se colocar comandos da linguagem fonte; o pré-processador copia-os para o programa gerado, reconhecendo o último desses comandos pelo início da próxima parte da tabela de decisão.

4.3.3 Definição de Conjuntos de Valores (opcional)

Seguindo a técnica do LDT[Se-Ma 88], porém, ao contrário deste, opcionalmente podem ser especificados conjuntos de valores; as condições da tabela de decisão podem testar a pertinência de dados a esses conjuntos. Essa parte da tabela é encabeçada pelo comando

`%Sets`

A especificação de cada conjunto é feita dando-se um nome ao conjunto (com até 32 caracteres, começando por letra), seguido de um separador “|” ou “=” e em seguida na mesma linha uma cadeia que define o conjunto. Seja CV_i o nome de um conjunto e S_i a cadeia de símbolos que o define. Assim, esta parte da tabela é especificada da seguinte maneira, onde o alinhamento vertical é mera questão estética.

`%Sets`

```

 $CV_1$  |  $S_1$ 
 $CV_2$  |  $S_2$ 
 $CV_3$  |  $S_3$ 
      :
      :
```

Vamos apresentar a seguir como as expressões S_i são especificadas. Sejam “constante” uma constante numérica inteira ou real sem sinal ou precedida de “-” (“menos” unário) ou cadeias de caracteres (“string”) e “variável” o nome de uma variável válida na linguagem fonte sendo gerada.

Def.: Uma expressão S_i pode ser especificada da seguinte forma:

- a) Uma constante ou variável. Neste caso S_i descreve um conjunto unitário.
- b) Um símbolo de comparação ($<$, $<=$, $=$, $>=$, $>$) seguido de uma constante ou variável.
- c) Um intervalo de constantes ou variáveis. Emprega-se a notação de “subrange” em Pascal.

As partes seguintes dessa definição são dadas em ordem decrescente de precedência dos operadores. Sejam S_i e S_j duas cadeias de símbolos que descrevem conjuntos arbitrários. São expressões válidas:

- d) (S_i) .
- e) A complementação de S_i , $\sim S_i$.
- f) A intersecção de S_i e S_j , $S_i \& S_j$.
- g) A diferença de S_i e S_j , $S_i \setminus S_j$.
- h) O produto cartesiano de S_i e S_j , $S_i : S_j$ (Conjunto de n-uplas).
- i) A união de S_i e S_j , $S_i ; S_j$.

Como visto, os operadores utilizados para complementação, intersecção, diferença, produto cartesiano e união são “ $\sim$ ” (til), “ $\&$ ”, “ $\setminus$ ”, “ $:$ ” e “ $;$ ”, respectivamente.

Para exemplificar, segue a figura com uma parte da tabela de decisão contendo uma série de definição de conjuntos.

%Sets		
CV1		Constante
CV2		Variável
CV3		1..35
CV4		≥ -25
CV5		$< Z$
CV6		'a'..'h'
CV7		$>1.5 \& < 3.14$
CV8		1;3;7;-45;35;89
CV9		$\geq 5 \& \leq 15$; 45..79; 99;101; ≥ 200
CV10		1..200\101..120
CV11		$\sim(1..200\setminus 101..120)$
CV12		'a'..'h':(1..34;38..99)
CV13		1..31:(1;3;5;7;8;10;12);1..30:(4;6;9;11); 1..29:2
Pi		3.14

4.3.4 Condições

Esta parte especifica as linhas de condição. O comando

```
%conditions      |
```


assinala este fato ao pré-processador. Nas linhas seguintes a este comando são especificadas as condições tanto limitadas como estendidas. A parte de descrição da condição é separada das entradas pelo símbolo “|”, numa posição de livre escolha dentro da linha, por parte do usuário, mas esta posição deve ser a mesma para todas as linhas, e é determinada pela localização do símbolo correspondente no comando %conditions. Naquele comando, como visto acima, o símbolo “|” (chamado de *separador*) é colocado na posição desejada, fixando a coluna em que ele ficará em todas as linhas de condição, separando a parte de descrição da parte de entradas de condição.

As entradas de condição ficam à direita do separador “|” alinhadas à esquerda por coluna, regra por regra, ou seja, todas as entradas de condição de uma mesma regra devem começar na mesma coluna. A coluna mais à esquerda de cada entrada de condição da primeira linha determina o alinhamento vertical das regras da tabela de decisão.

As entradas de linhas limitadas comportam os símbolos “Y”, “N”, “-”, “\$” e “*” e a parte de descrição, neste caso, admite quaisquer expressões lógicas, podendo incluir funções que retornam este tipo de valores. Para linhas deste tipo, o pré-processador gera comandos “if *C*”, onde *C* é a cadeia que está na parte de descrição da linha.

Se a linha é da forma estendida, o pré-processador admite do lado esquerdo uma variável ou uma n-upla de variáveis cuja pertinência deve ser testada a conjuntos determinados pelas entradas de condição. Nelas, são admitidas as formas usuais como descritas na seção 3.1 e nomes de conjuntos definidos na parte de especificação de conjuntos de uma tabela (seção 4.3.3). Para linhas deste tipo, o pré-processador fará a conversão para forma limitada antes de iniciar o processamento da tabela e os comandos “if” gerados para avaliação das condições testará a pertinência das variáveis de condição aos respectivos conjuntos, através de predicados que representam estes conjuntos. Por exemplo, se uma determinada regra requer que uma variável *x* deva pertencer ao conjunto CV3 da lista acima, o pré-processador avaliará esta pertinência com o comando:

if ($x \geq 1$) and ($x \leq 35$) ...

As entradas de condição estendidas não podem ser variáveis com nomes Y e N, porque seriam confundidas com as entradas limitadas. Como o autômato consegue verificar as expressões, normalmente seria possível saber se a parte de descrição de uma linha de condição contém uma expressão lógica ou não, mas como ele não conhece os tipos das variáveis, fica impossível saber se uma única variável que está na parte de descrição é “booleana” ou não.

4.3.5 Ações

A execução da tabela produz a execução das ações assinaladas numa única coluna, correspondendo à regra que for selecionável pela transação corrente. Como o pré-processador verifica ambigüidades estruturais e não as admite, existe, no máximo, uma regra selecionável para cada transação. Por outro lado, em razão de utilizar o método “esquerda para direita”, valem as observações feitas em 2.4.9.

As entradas de ação devem estar alinhadas verticalmente à esquerda com as entradas de condição da regra correspondente, ou seja, todas as entradas tanto de condição como de ação de uma mesma regra começam numa única coluna. São números que indicam a ordem de execução

da respectiva ação dentro da regra. As ações que não são executadas numa regra, podem ter as entradas correspondentes em branco, ou com o valor zero, ou ainda com o símbolo “-”.

A especificação da “else rule” é feita através de uma coluna de entradas de ação com uma coluna vazia de entradas de condição à direita da última regra, ou seja, trata-se de uma coluna que estipula uma sequência de ações localizada à direita do corpo da tabela. Neste caso, excepcionalmente, o alinhamento vertical da coluna não é determinado pela primeira linha de condição e sim pela posição da primeira linha de ação que tem uma entrada além da última coluna de condições. Na tabela que está no exemplo 4.3, a “else rule” está especificada na coluna à direita além da última coluna de condições e determina que uma função, no caso, “Else_Rule(Func)” seja executada se nenhuma regra for satisfeita. É o caso, no exemplo, de um funcionário com salário menor que 10.000, avaliação entre zero e cinco e com mais que 10 faltas no período. Esta parte é especificada pelo cabeçalho

%Actions

Como ocorre com as linhas de condição, cada linha de ação tem as partes de descrição e de entradas separadas pelos símbolo “|”, cuja posição é determinada pela coluna fixada através do separador no comando %conditions.

A parte de descrição da linha de ação pode comportar os seguinte comandos:

%exec table Nome.Tabela.Dec	(a)
%goto table Nome.Tabela.Dec	(b)
%exec Nome.Transf	(c)
Comandos_da_linguagem fonte	(d)

Se a parte de descrição contém uma ação do tipo (a), é produzido um desvio para a Tabela de decisão cujo nome foi especificado. Depois de executadas as ações dessa segunda tabela, é feito um retorno para a tabela chamadora, executando-se a ação seguinte na regra selecionada.

No caso do tipo (b), a execução é análoga ao caso anterior, exceto que não é feito o retorno para a tabela chamadora. Na implementação deste pré-processador, não é permitida que uma mesma tabela seja chamada por comandos do tipo (a) e (b), ou seja, se ela foi chamada pelo comando %exec então não poderá ser chamada pelo comando %goto e vice-versa. A primeira tabela de um grupo de decisão não pode ser chamada pelo comando %exec pois o seu final determina o encerramento da execução do grupo e não poderá haver retorno. Esta restrição pode ser contornada colocando-se inicialmente na base da pilha um indicador para o rótulo do fim do grupo, o que não é feito pelo pré-processador, porque esta permissão impede que possa ser chamada pelo comando goto.

No caso do tipo (c) é produzido um comando de desvio durante a execução do programa gerado para o rótulo Nome.Transf, que é o nome de uma transformação. Atingido o fim da transformação, o controle é retornado ao próximo comando da regra.

No caso do tipo (d), o comando da linguagem fonte é copiado no código gerado nos lugares apropriados para sua execução. Esse comando pode ser inclusive uma chamada de procedimento ou, no caso de C, o nome de uma “macro”.

4.3.6 Fim da Tabela

O fim de cada tabela é reconhecido pelo comando

```
%end table
```

Qualquer símbolo entre este comando e o final da linha é entendido como simples comentário. Esta observação é válida para todos os comandos do pré-processador que iniciam ou finalizam uma tabela. Vale também para os comandos iniciais e finais da especificação de uma transformação que será vista na seção 4.4.

Para exemplificar, suponha o sub-problema, parte de um programa, que consiste em ler os caracteres de um teclado para o preenchimento de campos numéricos inteiros ou reais, de acordo com o valor da variável *Tipo* que pode assumir um dos valores (*Inteiro*, *Real*). Na questão, a cadeia *S* armazena o número que está sendo lido e a posição do elemento corrente no vetor *S* é apontado pela variável *Atual*. A variável *Entrada* contém o símbolo corrente que deve ser examinado e *Houve_Ponto* é uma variável “booleana” que indica se já ocorreu algum ponto ou vírgula decimais, para evitar duplicidade deste símbolo dentro de um número real. Podemos observar, de imediato, que o símbolo “-” unário só é aceito no início do campo, o que é detectado pelo fato da variável *Atual* conter o valor 1. No caso de um tipo inesperado, a tabela executa a “else rule” que especifica uma ação que contém uma chamada para a transformação de nome *Erro*. Estão definidos três conjuntos de valores *P*, *U* e *CP*, o primeiro com os símbolos admitidos como ponto decimal, o segundo contendo os algarismos do sistema decimal e o terceiro como o complemento do primeiro. Assim, $P = \{',', '.'\}$ e $U = \{ '0'..'9' \}$. É claro que se esperaria que os nomes destes conjuntos fossem algo como *Ponto* e *Número* para melhor qualificar estes conjuntos de valores, mas como estes nomes serão repetidos várias vezes numa linha de condição, teríamos problemas de espaço nas linhas deste texto. Ao transportar esta tabela do programa real para este exemplo, introduzimos algumas modificações de caráter simplificador para facilitar as explicações. Na tabela, a linha imediatamente seguinte ao comando `%conditions` é um comentário que enumera as regras para facilitar a identificação de cada uma delas, enquanto a tabela está sendo discutida.

```
%decision table Leitura
```

```
%sets
```

```
P = ',';','.'
```

```
U = '0'..'9'
```

```
CP= ~(' ',';','.')'
```

```
%conditions
```

```
%— Número da Regra
```

```
Tipo = Inteiro
```

```
Tipo = Real
```

```
Entrada
```

```
Atual = 1
```

```
Houve_Ponto
```

```
S[Atual]
```

```
%Actions
```

	1	2	3	4	5	6	7	8
Tipo = Inteiro	Y	Y	*	*	*	*	*	*
Tipo = Real	*	*	Y	Y	Y	Y	Y	Y
Entrada	'-'	U	'-'	P	P	U	U	U
Atual = 1	Y	-	Y	-	-	-	-	-
Houve_Ponto	*	*	*	N	Y	N	Y	Y
S[Atual]	-	-	-	-	-	-	P	CP

%exec Vírgula_Ponto					1			
%exec Le_Char		1	1	1	2	1	1	1
%exec Avança		2	2	2	3	2	2	2
%exec Inverte_Pto					4		3	
%exec Erro_Ponto					1			
%exec Erro								1
%end table								

Além das verificações já citadas pela tabela, vamos enumerar algumas situações importantes nela especificadas:

1. O símbolos “*” que aparecem na tabela são empregados como entradas de condição “N” implícito, pois podemos observar que se o *Tipo* for *Inteiro*, então obrigatoriamente a condição *Tipo = Real* é falsa e vice-versa. Analogamente, a condição expressa pela variável booleana *Houve_Ponto* é obrigatoriamente falsa se o *Tipo* for inteiro, como na regra 2, ou se ainda nenhum caractere foi lido, o que é detectado pelo valor 1 na variável *Atual*, como ocorre nas regras 1 e 3.
2. Se a *Entrada* contém um símbolo “-” (menos unário) e não é a primeira posição do vetor *S*, nenhuma regra é satisfeita e a ação especificada na “else rule” é executada. Caso contrário, as regras 1 e 3 tratam adequadamente o símbolo para os tipos inteiro e real respectivamente.
3. Se o tipo do campo for inteiro e *Entrada* contém um algarismo, a regra 2 executa as ações próprias para o caso.
4. Sendo o tipo do campo um real e a *Entrada* contém um símbolo admitido como ponto decimal, existem duas possibilidades: Se ele não ocorreu até o momento, é adequadamente aceito pela regra 4. Caso contrário, existe um erro de duplicidade de ponto decimal, cujo tratamento é feito pela regra 5, invocando a transformação de nome *Erro_Ponto*.
5. A regra 7 é um caso especial. Suponha que o tipo do campo seja um número real, *Entrada* contém um dígito e que já ocorreu o ponto decimal. Considere também que outras partes do programa tratam de teclas de movimentação do cursor dentro do campo. Assim, é possível que a variável *Atual* esteja apontando exatamente para o elemento de *S* com o ponto decimal. Deste modo, o usuário arrependendo-se do ponto decimal que foi digitado, deseja sobrepor-lo com um número e isto requer que o conteúdo da variável *Houve_Ponto* seja invertido. Isto é feito pela regra 7. Dentro de condições análogas, exceto que a posição corrente de *S* não contém um ponto decimal, o símbolo de entrada deve ser normalmente tratado, o que é feito pela regra 8.
6. Se o tipo do campo é real e a entrada contém um dígito e ainda não ocorreu o ponto decimal, a regra 6 executa as ações adequadas para o caso.

As linhas de ação da tabela contêm na parte de descrição chamadas para transformações. Cada uma delas deve especificar comandos da linguagem fonte para o cumprimento de suas tarefas o que não é mostrado aqui. Descreveremos genericamente a função de cada transformação.

1. *Vírgula_Ponto* deve transformar os símbolos externos admitidos como ponto decimal em “.” que é a forma usual de representação interna.
2. *Le_Char* simplesmente aceita o símbolo de entrada na cadeia *S*.
3. *Avança* produz o avanço do apontador *Atual* para a próxima posição do vetor *S*.
4. *Inverte_Pto* inverte o conteúdo da variável *Houve_Ponto*.
5. *Erro_Ponto* trata da duplicidade de ponto decimal num campo de número real.
6. *Erro* trata de todas as situações não enumeradas pela tabela de decisão.

4.4 Especificações das Transformações

Um grupo de decisão para o pré-processador é composto de uma sequência de tabelas de decisão, que são detectadas pelo seu comando inicial, sendo que a primeira delas determina o início do próprio grupo. Encerrada a parte de tabelas, devem ser especificadas as transformações que foram referenciadas pelas linhas de ação e nas seções de preparação das tabelas do grupo. Se não existe referência a nenhuma transformação, esta parte é simplesmente omitida. Para finalizar o grupo deve-se colocar o comando `%end transf`, independentemente de haver ou não especificação de transformações. Se é o único grupo, ou o último do arquivo, o encerramento da entrada determina o fechamento do grupo automaticamente, se estiver ausente. Assim, a tarefa de tradução de uma única tabela dispensa qualquer comando relativo a tratamento de grupo de decisão.

Uma transformação é um sequência de comandos da linguagem fonte que pode ser referenciada pelo seu nome. A sua especificação tem os comandos inicial e final e o corpo central que é a simples listagem dos seus comandos. O comando inicial é

```
%transf Nome_Transf
```

e o comando final é

```
%end Nome_Transf
```

onde em ambos os comandos “Nome_Transf” é o nome da transformação que deve ser um rótulo válido na linguagem fonte. Assim, o esquema da especificação de uma transformação é o seguinte:

```
%Transformation Nome_Transf
      :
      Comandos_da_Linguagem_Fonte
      :
%end Nome_Transf
```

Feita a descrição das partes de uma tabela e de uma transformação que são os componentes de um grupo de decisão, vamos mostrar o esquema geral de um grupo de decisão que é o seguinte:

```

%decision table T1
    "corpo da tabela T1"
%end table T1
%decision table T2
    "corpo da tabela T2"
%end table T2
:
%decision table Tn
    "corpo da tabela Tn"
%end table Tn
%transformation A1
    "corpo da transformação A1"
%end A1
%transformation A2
    "corpo da transformação A2"
%end A2
:
%transformation Am
    "corpo da transformação Am"
%end Am
%end transf

```

4.5 Código Gerado e Comentários

O código gerado para uma tabela de decisão é inserido logo após o fim desta. A execução de uma ação do tipo %exec... que é a chamada de uma tabela ou transformação exige a execução desta seguida do retorno para a ação seguinte. Para isso é gerado código para o gerenciamento correto dos retornos de elementos que podem ser chamados de diversos pontos de uma mesma tabela ou mesmo de diferentes tabelas de um mesmo grupo. A estrutura utilizada é uma pilha que é colocada no programa gerado. Neste caso, deve ser inserido o comando

```
%var
```

numa linha do programa fonte, indicando ao pré-processador o local onde deve ser declarada esta estrutura de dados. É claro que deve ser colocado numa linha da região onde são feitas as declarações das variáveis do programa. Uma alternativa para a pilha seria a utilização de funções e procedimentos para o código produzido para tabelas e transformações, mas apresenta a dificuldade de que todas as variáveis devem ser globais o que nem sempre acontece. No caso da linguagem Pascal poderíamos contornar o problema com procedimentos encaixados, mas esta solução não existe em C. No caso da Linguagem fonte ser Pascal, devem ser inseridos os comandos

```

%label
%init

```


indicando os locais de declaração de rótulos e de inicialização de variáveis relativas à pilha. O código gerado para a determinação do ponto de retorno é inserido logo após o comando de finalização do grupo

```
%end transf
```

Após um comando `%end transformations` encerrando um grupo de decisão, pode-se colocar outros grupos, mas elementos destes novos grupos não podem referenciar elementos de grupos já finalizados ou a serem futuramente abertos. Como pode ser observado o mesmo comando foi escrito com as palavras `transf` e `transformations`. Isto é aceito pelo pré-processador, além de outras abreviaturas como segue:

decision	dec
transformations	transf
conditions	cond
actions	act
execute	exec
preparation	prep

O pré-processador produz o arquivo com o mesmo nome do arquivo com o programa fonte, seguido da extensão `'.fte'`, onde ficam as tabelas traduzidas. A listagem do programa fonte com as indicações dos possíveis erros pode ser impressa ou armazenada num arquivo em disco com o mesmo nome do arquivo com o programa fonte, seguido da extensão `'.lst'`. A escolha é feita conversacionalmente no início do programa.

O programa foi escrito em Pascal no sistema "Turbo Pascal 5.0". Para o seu funcionamento os seguintes arquivos são necessários: `'Decis.exe'`, `'Tabela.prc'`, `'Pascal.rel'` e `'C.rel'`.

Nas tabelas de decisão, o alinhamento vertical das colunas é essencial e, assim, a tabulação utilizada pelos editores torna-se relevante, pois um fator de tabulação incorreto produz o desalinhamento da entrada. Após a ativação do pré-processador, ele pergunta o valor da tabulação (2 a 16). No caso de não haver tabulação, o valor é indiferente.

4.6 Um exemplo simples

Para exemplificar, vamos retomar a tabela apresentada na seção 2.4.9, para escolher a próxima condição pelo método "esquerda para direita". A descrição da primeira linha de condição naquela tabela é:

Linha C_i é irrelevante.

Para o momento, vamos substituí-la por

Linha_Irrelevante(i)

e vamos supor que seja o nome de uma função que retorna verdadeiro ou falso, conforme as entradas de condição sejam todas irrelevantes ou não. Assim, dentro das especificações do pré-processador, a tabela poderia ser escrita da seguinte forma:

```

%generate Pascal
%decision table Esquerda-para-Direita
%sets
    YN          | 'Y','N'
    IR          | '-','$','*'
%conditions
    Linha_Irrelevante(i) | Y    N    N    N    N    N
    C[k,1]          | -    IR   IR   IR   YN   YN
    C[i,1]          | -    YN   IR   IR   YN   YN
    L < Min         | -    -    Y    N    Y    N
    L = Min         | -    -    *    Y    *    Y
    J > Max         | -    -    -    Y    -    Y
%actions
    Remove_Linha(i) | 1
    k := i          |      1    1    1    1
    Max := J        |      2    2    2    2    1
    Min := L        |      3    3          3    2
%end table

```

Esta tabela editada num arquivo e submetida ao Decis, produz a seguinte saída:

```

(* PRE-PROCESSADOR DE TABELAS DE DECISAO - VERSAO 1.1
DEPARTAMENTO DE CIENCIA DA COMPUTACAO - IME - USP *)

```

```

(* %generate Pascal *)
(*2
%decision table Esquerda-para-Direita
%sets
    YN | 'Y','N'
    IR | '-','$','*'
%conditions
    Linha_Irrelevante(i) | Y    N    N    N    N    N
    C[k,1]          | -    IR   IR   IR   YN   YN
    C[i,1]          | -    YN   IR   IR   YN   YN
    L < Min         | -    -    Y    N    Y    N
    L = Min         | -    -    *    Y    *    Y
    J > Max         | -    -    -    Y    -    Y
%actions
    Remove_Linha(i) | 1
    k := i          |      1    1    1    1
    Max := J        |      2    2    2    2    1
    Min := L        |      3    3          3    2
%end table *)

ESQUERDA_PARA_DIREITA:
if (Linha_Irrelevante(i)) then
begin
    (* rule 1 *)
    Remove_Linha(i);
end
else
begin
    if (((C[k,1]='-')or(C[k,1]='$'))or(C[k,1]='*')) then
begin
    if ((C[i,1]='Y')or(C[i,1]='N')) then
begin
    (* rule 2 *)

```

```

    k:=i;
    Max:=J;
    Min:=L;
  end
else
  begin
    if (((C[i,1]='-')or(C[i,1]='$'))or(C[i,1]='*')) then
      begin
        if (L<Min) then
          begin
            (* rule 3 *)
            k:=i;
            Max:=J;
            Min:=L;
          end
        else
          begin
            if (L=Min) then
              begin
                if (J>Max) then
                  begin
                    (* rule 4 *)
                    k:=i;
                    Max:=J;
                  end;
                end;
              end;
            end;
          end;
        end;
      end
    end
  end
else
  begin
    if ((C[k,1]='Y')or(C[k,1]='N')) then
      begin
        if ((C[i,1]='Y')or(C[i,1]='N')) then
          begin
            if (L<Min) then
              begin
                (* rule 5 *)
                k:=i;
                Max:=J;
                Min:=L;
              end
            else
              begin
                if (L=Min) then
                  begin
                    if (J>Max) then
                      begin
                        (* rule 6 *)
                        Max:=J;
                        Min:=L;
                      end;
                    end;
                  end;
                end;
              end;
            end;
          end;
        end;
      end
    end;
  end
end;
end;

```


Capítulo 5

O Gerador I-M-E

Este capítulo trata de aspectos relacionados à implementação de um gerador de aplicações. Ele recebe como entrada um arquivo com especificações segundo o método I-M-E e produz como saída um programa em linguagem Pascal. Está escrito em Pascal, utilizando o sistema *Turbo Pascal*, 5.0.

5.1 Algumas Palavras sobre o Método I-M-E

Embora o presente trabalho pretendesse apenas apresentar o gerador de aplicações I-M-E, esbarramos na necessidade de escrever algumas pouquíssimas palavras sobre o método em que ele se baseia. O método I-M-E teve o seu início com o trabalho de E. Marussi que implementou um gerador de aplicações chamado L.D.T., cujas noções básicas se encontram em [Se-Ma 88]. Nele foram introduzidas algumas modificações por V. W. Setzer, mantendo a sua estrutura geral e chamando-o de método I-M-E.

Dado um problema genérico, o método supõe que a sua solução é efetivada através de um processo e sub-processos. O processo e os sub-processos tratam de exatamente um conjunto de dados, cada um, atuando sobre os seus elementos. Estes conjuntos podem ser compostos de elementos complexos e não necessariamente de elementos de um conjunto particular de dados físico. Por exemplo, o conjunto de dados tratado pode ser o nome dos alunos e sua respectiva nota que, por alguma razão qualquer, estão em arquivos distintos. Assim, o processo ou sub-processo, que queira tratar das notas dos alunos, manipula um conjunto abstrato de elementos que são compostos de dados obtidos a partir de dois arquivos físicos. Se tais dados estivessem num único arquivo físico, ele corresponderia exatamente ao conjunto manipulado pelo processo ou sub-processo em questão. No exemplo citado, onde um aluno tem a sua respectiva nota, cada elemento do conjunto manipulado pelo processo ou sub-processo é composto de um elemento do arquivo de nomes e do correspondente elemento do arquivo de notas. Isto é, considerando $A = \{N_1, N_2, \dots, N_p\}$ e $B = \{n_1, n_2, \dots, n_p\}$, os conjuntos de nomes dos alunos e de suas respectivas notas, o conjunto de dados manipulado pelo processo ou sub-processo seria $C = \{(N_i, n_i)\}$. Se pretendêssemos processar somente os dados relativos aos alunos aprovados, o conjunto tratado pelo processo ou sub-processo seria $C = \{(N_i, n_i) : n_i \geq 5\}$.

Suponha o problema anterior do arquivo de nomes de alunos e sua correspondente nota no arquivo de notas ligeiramente modificado, como segue. Sejam $A = \{N_1, N_2, \dots, N_p\}$ e

$B = \{n_{11}, n_{12}, \dots, n_{1m_1}, n_{21}, n_{22}, \dots, n_{2m_2}, \dots, n_{p1}, n_{p2}, \dots, n_{pm_p}\}$, os conjuntos de nome dos alunos e de notas respectivamente, onde cada aluno N_i tem m_i notas, a saber, $n_{i1}, n_{i2}, \dots, n_{im_i}$. Neste caso, o processo que desejasse tratar cada nota com o seu respectivo nome de aluno, manipularia o conjunto C , onde cada um de seus elementos é um par composto do nome do aluno e uma de suas notas. Assim, para obter um elemento a ser tratado seria necessário ler um nome de aluno em A e diversas notas do arquivo B . Formalmente, o processo ou sub-processo trataria do conjunto

$$C = \{(N_1, n_{11}), (N_1, n_{12}), \dots, (N_1, n_{1m_1}), \\ (N_2, n_{21}), (N_2, n_{22}), \dots, (N_2, n_{2m_2}), \\ \vdots \\ (N_p, n_{p1}), (N_p, n_{p2}), \dots, (N_p, n_{pm_p})\}$$

Cada processo ou sub-processo manipula um conjunto de dados, onde cada elemento tem um significado próprio de unidade para o problema específico e da forma como se pretende resolvê-lo. Para exemplificar o fato de que o conjunto tratado por um sub-processo é dependente do problema particular e do enfoque atribuído pelo usuário, suponha no problema anterior que a solução encontrada foi tratar o arquivo A de nomes como o conjunto a ser manipulado por um processo P e que para cada um de seus elementos um sub-processo SP trataria as suas notas no arquivo de notas B . Isto resolveria o problema que desejasse tratar cada aluno e sua respectiva média de notas. Assim, o conjunto abstrato tratado pelo processo seria $C = \{N_i, M_i\}$, onde M_i seria obtido pela execução de um sub-processo que trataria do conjunto abstrato $C'_i = \{n_{i1}, n_{i2}, \dots, n_{im_i}\}$, para cada aluno N_i , $1 \leq i \leq p$.

Existe uma diferença conceitual entre conjuntos de dados físicos e conjuntos de dados tratados por um processo ou sub-processo. Como estes dois objetos estão presentes nas discussões, diremos que o conjunto tratado por um processo ou sub-processo P é o conjunto *abstrato* de P .

5.1.1 Partes ou Etapas

O método I-M-E propõe que o tratamento de um conjunto abstrato de dados por um processo ou sub-processo seja dividido em três *partes*, também chamadas de *etapas*.

- **Parte de Inicialização:** É a parte executada uma única vez para todo o conjunto abstrato, antes de iniciar a execução repetitiva das operações sobre cada um de seus elementos. Por exemplo, no processo que calcula uma função $f(c_1, c_2, \dots, c_n)$, onde $C = \{c_1, c_2, \dots, c_n\}$ é o conjunto abstrato, poderíamos ter que inicializar f com algum valor, o que é feito adequadamente nesta parte. Vamos denotá-la simplesmente por *etapa* ou *parte I* ("initialization part") ou ainda de *etapa* ou *parte inicial*.
- **Parte Principal:** É a parte do processo ou sub-processo que opera sobre cada um dos elementos do conjunto abstrato, inclusive a sua leitura e demais operações que são feitas. Denotaremos por *etapa* ou *parte M* ("main part") ou ainda de *etapa* ou *parte principal*.
- **Etapa de Finalização:** É a parte do processo que opera sobre o conjunto abstrato como um todo, de forma análoga à etapa inicial, exceto que a sua execução é efetivada após o tratamento de todos os elementos do conjunto abstrato. Denotaremos por *etapa* ou *parte E* ("ending part") ou ainda de *etapa* ou *parte final*.

Para exemplificar, suponha um conjunto $C = \{c_1, c_2, \dots, c_n\}$ de inteiros e desejamos calcular $f = (\sum c_i) \bmod n$. Como podemos ver, para o processo que resolve o problema, o conjunto abstrato é exatamente o conjunto C . A parte de inicialização faria a atribuição do valor zero para f . A parte principal incrementaria f com o valor de cada elemento do conjunto e a parte final calcularia o módulo de f pelo número de elementos do conjunto.

Como dissemos a etapa principal trata de cada elemento de um conjunto abstrato. Assim, vemos que toda etapa M de um processo ou sub-processo determina implicitamente um “loop”. Uma das propriedades do método é que os “loops” não são especificados; eles são providenciados automaticamente pelo gerador I-M-E. Como deduzir onde encerrar o “loop”? Isto é determinado pela estrutura do conjunto abstrato que está sendo tratado [Ja 75]. Como será visto mais adiante, serão especificados critérios que determinam quais elementos do conjunto abstrato devem ser tratados e a sua exaustão encerra o “loop”.

5.1.2 Fases

Cada uma das partes de um processo ou sub-processo é executada numa sequência de três fases:

- Seleção: Fase de leitura, que é a obtenção dos dados a serem processados, tanto de arquivos físicos como de tabelas e sequências. Esta operação trata de escolher dentro do conjunto qual o elemento a ser tratado num determinado momento, independentemente de qual o suporte físico em que está o conjunto. É claro que do ponto de vista da implementação desta operação, a seleção de um arquivo implica na leitura de um registro num buffer da memória, enquanto a seleção de uma tabela implica na movimentação do apontador para um determinado elemento, mas isto fica a cargo do gerador. Por simplicidade, denotaremos por *fase S* (“selection”). Dentro da etapa M , esta fase atua sobre os conjuntos de dados de forma que as fases seguintes tratem apenas dos elementos do conjunto abstrato do processo ou sub-processo.
- Escolha: Nesta fase, o processo ou sub-processo trata de escolher quais operações de transformação serão executadas sobre os dados selecionados na fase de seleção. Denotaremos por *fase C* (“choice”). Em termos de linguagens de programação, corresponde aos comandos de seleção “if”. Dentro do método esta fase é implementada por tabelas de decisão.
- Transformação: Fase de execução das operações escolhidas na fase C sobre os dados selecionados na fase S . Denotaremos por *fase T* (“transformations”).

Estas divisões são fisicamente separadas e executadas nesta ordem. Assim, a especificação de um processo ou sub-processo neste método é uma sequência de instruções que obedecem estritamente à ordem das etapas e das fases. A execução de um processo ou sub-processo requer a execução das etapas inicial, principal e final, nesta ordem, e a execução de cada uma dessas etapas produz a execução das fases de seleção, escolha e transformação, também nesta ordem. Algumas destas partes e fases podem estar ausentes, exceto a parte principal, pois cada processo ou sub-processo trata de um conjunto abstrato de dados e nesta parte é necessário especificar o que fazer com cada um de seus elementos. Na etapa M sempre deve estar presente a fase de seleção que obtém cada um dos elementos do conjunto abstrato do processo ou sub-processo.

Dissemos que a etapa principal determina implicitamente um “loop” e que o gerador o produz automaticamente, junto com um mecanismo para o seu encerramento, que é apurado pelos critérios especificados para apontar quais elementos compõem o conjunto abstrato a ser tratado. Estes critérios e informações sobre qual conjunto abstrato o processo está operando são especificados na fase de seleção. A cada iteração do “loop” automático, um novo elemento do conjunto abstrato é obtido, satisfazendo as condições especificadas. Dizemos que tal elemento foi *selecionado*.

Um problema é solucionado dentro do método por um processo e sub-processos. Eventualmente os sub-processos são desnecessários. O problema envolve um conjunto abstrato que é tratado pelo processo. Na especificação deste processo pode ser necessário tratar outros conjuntos abstratos o que deve ser feito por sub-processos, cada um tratando de exatamente um conjunto abstrato e, neste caso, a execução do processo produz a execução destes sub-processos.

Como o processo e sub-processo têm exatamente a mesma estrutura, vamos utilizar o termo sub-processo para referir-nos genericamente a qualquer um deles, a menos que a distinção seja necessária.

Pelo exposto, a estrutura de um processo ou sub-processo é:

- Initialization part
 - Selection phase
 - Choice phase
 - Transformation phase
- Main part
 - Selection phase
 - Choice phase
 - Transformation phase
- Ending part
 - Selection phase
 - Choice phase
 - Transformation phase

Como as tabelas de decisão na sua parte de ações podem invocar um sub-processo, a seleção pode obter resultados da execução de sub-processos, e transformações e ações de preparação podem efetuar chamadas de sub-processos que têm todo o poder de especificação do sub-processo chamador, podemos ter problemas complexos especificados no método I-M-E.

A invocação de um sub-processo, em qualquer das formas citadas, significa que o sub-processo chamador, que por definição trata de um conjunto abstrato de dados, tem a necessidade de usar o resultado do processamento de um outro conjunto e isto é feito através de sub-processos.

A fase de seleção é composta de comandos de seleção; a de escolha por tabelas de decisão e a fase de transformação pela definição das transformações referenciadas nas tabelas de decisão. Tal definição é a enumeração dos comandos de transformação necessários para efetivar a transformação desejada. Se a etapa não possui a fase *C*, então a fase *T* é uma simples enumeração dos comandos de transformação, sem a definição de transformações.

A especificação para o gerador I-M-E, que chamaremos simplesmente de *gerador*, é exatamente como encontrado no esquema, mas todos os comandos destinados a ele são precedidos

pelo símbolo "%". Isto é feito para distinguir os comandos do gerador com outros que o usuário pode inserir na própria linguagem Pascal. Tal marcação dos comandos deveu-se ao fato do projeto ser incremental. Aos poucos fomos introduzindo novos comandos do gerador, até atingirmos especificações completas sem uma linha em Pascal. Agora seria o momento de inverter a especificação: os comandos precedidos de "%" seriam simplesmente inseridos no ponto correspondente do programa gerado.

5.1.3 Sub-processos de Agrupamento

Existem sub-processos anômalos na sua estrutura, no sentido de que eles não possuem a etapa M , contrariando a regra geral acima mencionada. São chamados de sub-processos de agrupamento e têm a finalidade de agrupar o conjunto abstrato de um sub-processo, de acordo com o valor de um campo especificado pelo usuário.

Consideremos um sub-processo SP que trata de um conjunto abstrato C na sua etapa principal e f o campo dos registros que compõem o conjunto abstrato pelo qual o usuário deseja efetuar o agrupamento. O sub-processo SP trata cada um dos elementos de C e particiona o conjunto C em sub-conjuntos

$$\begin{aligned} C_1 &= \{c_{11}, c_{12}, \dots, c_{1n_1}\} \\ C_2 &= \{c_{21}, c_{22}, \dots, c_{2n_2}\} \\ C_3 &= \{c_{31}, c_{32}, \dots, c_{3n_3}\} \\ &\vdots \\ C_m &= \{c_{m1}, c_{m2}, \dots, c_{mn_m}\}, \end{aligned}$$

de modo que:

$$\begin{aligned} (1) \quad & \cup_{i=1}^m C_i = C, \\ (2) \quad & C_i \cap C_j = \emptyset, i \neq j, \\ (3) \quad & c_{ij} \cdot f = c_{ik} \cdot f, \quad 1 \leq j, k \leq n_i \quad \text{e} \\ (4) \quad & c_{in_i} \cdot f \neq c_{i+1,1} \cdot f, \quad 1 \leq i < m. \end{aligned}$$

Se $c_{in_i} \cdot f < c_{i+1,1} \cdot f$, $1 \leq i < m$, temos uma ordenação dos elementos de C e a condição (4) especifica o ponto comumente denominado de "quebra".

A cada sub-conjunto C_i da partição de C é executado um sub-processo de agrupamento SP' somente com as partes inicial e final. Por exemplo, estas partes poderiam corresponder à emissão de um título e ao rodapé de cada agrupamento, respectivamente. Cada elemento de C_i é tratado individualmente pelo sub-processo SP como todos os demais elementos de C , mas o sub-conjunto C_i , como um todo, é inicializado e finalizado pelas etapas inicial e final de SP' . Na etapa final deste sub-processo de agrupamento SP' podem ser usadas funções de totalização e contagem automáticas para o sub-conjunto C_i da partição de C , que foi recentemente agrupado:

- **Count:** Conta o número de elementos agrupados no sub-conjunto da partição. Na etapa final do sub-processo de agrupamento, aparece na forma **Count**.
- **Max:** Obtém o valor máximo de um campo f dos elementos do sub-conjunto da partição. Na etapa final do sub-processo de agrupamento, aparece na forma **Max(f)**.

- **Min:** Obtém o valor mínimo de um campo f dos elementos do sub-conjunto da partição. Na etapa final do sub-processo de agrupamento, aparece na forma **Min(f)**.
- **Total:** Totaliza o valor de um campo f dos elementos do sub-conjunto da partição. Na etapa final do sub-processo de agrupamento, aparece na forma **Total(f)**.
- **Average:** Calcula o valor médio de um campo f dos elementos do sub-conjunto da partição. Na etapa final do sub-processo de agrupamento, aparece na forma **Average(f)**.

Estas funções são chamadas normalmente de *funções de agregação* e quando utilizadas nas especificações para o gerador, devem vir precedidas do símbolo "%".

Os agrupamentos podem ser encaixados, de modo a proporcionar partições de sub-conjuntos que já são uma partição. Para este segundo nível de agrupamento seria especificado um outro sub-processo de agrupamento. Esta forma de tratar o conjunto abstrato de um sub-processo é especificada ao gerador através dos comandos de seleção da etapa principal, onde se estipula por qual campo se fará o agrupamento do conjunto abstrato e qual o sub-processo de agrupamento que deverá ser executado para cada sub-conjunto da partição.

5.2 Declarações

O gerador precisa produzir a declaração das variáveis utilizadas no processo, para o perfeito funcionamento do programa resultante em Pascal. No projeto inicial, pretendíamos obter as informações necessárias para isto de um Dicionário de Dados da instalação, que seria carregado pelo usuário através de programas especiais para isso. Como essa não era uma parte essencial da pesquisa, deixamos esses programas para uma outra etapa do desenvolvimento. Em seu lugar introduzimos o dispositivo essencial do gerador para ler as especificações de declarações e produzir as declarações correspondentes em Pascal. Será bastante simples fazer o gerador buscar essas especificações no Dicionário de Dados, onde seriam introduzidas por programas que poderiam utilizar telas e menus, orientando aos usuários as possibilidades de declarações.

As informações sobre a estrutura de dados utilizada devem ser colocadas antes do processo sendo especificado. Vamos apresentar o código de declarações em Pascal produzido pelo gerador após a apropriação das informações sobre a estrutura de dados.

Antes de especificar os processos e sub-processos, apresenta-se o comando

%declarations

seguido da lista de variáveis que compõem a estrutura de dados utilizada.

Os identificadores produzidos pelo gerador no programa objeto para o tratamento das estruturas declaradas pelo usuário terminam com "-" (grifo). Assim, existe a restrição sobre as variáveis do usuário para que não terminem com este caractere. O gerador precisa cuidar para que não ocorram conflitos entre os nomes produzidos. Assim, existem variáveis no código em Pascal que terminam com um, dois, ou três destes caracteres.

As especificações que apresentam as variáveis ao gerador produzem as correspondentes declarações na linguagem Pascal. Poderia imaginar-se que isto se efetivaria, permitindo-se que fossem feitas diretamente nessa linguagem no início das especificações. O primeiro entrave a esta alternativa se deve à necessidade de incorporar ao gerador a capacidade de identificar estas estruturas e suas características nessa linguagem, para trabalhar com elas nos comandos

especificados no restante do programa. Além disso, para cada identificador especificado pelo usuário, pode ser necessário produzir declaração de mais de uma variável. Normalmente para identificadores do tipo arquivo ou tabela, vários identificadores são gerados, para permitir o controle das operações que são efetuadas sobre eles.

De um modo geral, para que um gerador torne-se efetivo num grande número de aplicações, seria necessário escolher um ambiente específico para que fosse possível produzir comandos adequados para ele, utilizando ferramentas próprias do sistema. Por exemplo, um ambiente que disponha de mecanismos para manipulação de arquivos indexados permitiria a produção de comandos para execução de funções fornecidas para a sua manipulação, sem a necessidade de implementar esta estrutura de dados, que é um trabalho complexo.

5.2.1 Arquivos

A idéia intuitiva de um arquivo corresponde à de uma sequência de dados e a ela costumamos atribuir um nome. Deste modo, nem sempre é natural ao usuário a representação de arquivos por descritores que não são propriamente arquivos e sim variáveis que armazenam os seus atributos e que permitem a um mesmo identificador designar, num certo momento, o arquivo de funcionários de uma empresa e, num outro instante, designar o arquivo de medicamentos consumidos num mês pelo ambulatório da empresa. A contradição ainda é maior, se o nome do arquivo é "FUNC". Assim, o nome lógico de um arquivo está intimamente relacionado com o próprio arquivo e não é adequado que ele passe a ser o identificador de um outro conjunto de dados. Todavia, existem problemas em que é necessária a alteração do arquivo externo associado a um arquivo lógico durante a execução, o que ainda não é feito pelo gerador, mas que pode ser implementado, desde que se definam as regras em que esta operação possa ser efetuada.

Considere o seguinte trecho de declarações na forma como se especifica um arquivo para o gerador.

```
%file Notas(Numero:integer;  
             Nome:string[30];  
             Nota1,Nota2,Nota3,Nota4:real)  
on 'Alunos.Ent'
```

Esta especificação produz as seguintes declarações em Pascal, extraído de um programa produzido pelo gerador.

```
type  
T_NOTAS_=record  
  NUMERO:integer;  
  NOME:string[30];  
  NOTA1:real;  
  NOTA2:real;  
  NOTA3:real;  
  NOTA4:real;  
end; (* record *)
```

```
var
  NOTAS_Buf_:T_NOTAS_;
  NOTAS:file of T_NOTAS_;
```

O gerador produz a declaração de um tipo registro para os elementos do arquivo. Para um arquivo declarado com o nome A , o tipo é identificado com $T_A_$, onde T é a própria letra “T”. A variável arquivo recebe o mesmo nome de A , declarado pelo usuário, como sendo uma sequência de elementos do tipo $T_A_$. O “buffer” utilizado para leitura é declarado com o nome $A_Buf_$, como um registro do tipo $T_A_$.

Como podemos ver, a variável assim especificada fica vinculada a um arquivo externo fixo. As especificações do usuário sempre farão referência ao arquivo pelo nome que ele atribuiu e diz respeito ao elemento corrente do arquivo, ou porque foi selecionado, ou porque será inserido um novo registro. Internamente, o gerador substituirá adequadamente a variável arquivo pelo buffer correspondente. Numa expressão, a sequência $A.C$, onde trata-se do campo C do arquivo A , será substituída por $A_Buf_.C$.

Além do que aparece acima, sempre que houver a especificação de um arquivo, o gerador produz variáveis do tipo enumeração. Sejam $A_1, A_2, \dots, A_p$ os arquivos e $P_1, P_2, \dots, P_q$ o processo e sub-processos presentes no arquivo de entrada. As seguintes variáveis são declaradas:

$$A_{i--}, A_{2--}, \dots, A_{p--} : (\text{Closed--}, P_{1--}, P_{2--}, \dots, P_{q--}).$$

A variável A_{i--} indica que o arquivo A_i está fechado se o seu valor é Closed-- . Caso contrário, se A_{i--} vale P_{j--} , então o arquivo A_i está aberto e sua abertura ocorreu no sub-processo P_j . O fato de a constante Closed-- terminar com um caractere “-” enquanto outras constantes da enumeração com dois destes símbolos se deve à possibilidade de o usuário especificar um sub-processo de nome Closed . No início do código, após as declarações, são produzidos os comandos:

```
A_{i--} := Closed_
assign(A_i, 'Nome_Arquivo')
```

para todo o arquivo A_i , $1 \leq i \leq p$. *Nome_Arquivo* no comando acima é o nome externo do arquivo. Estes comandos que são executados no início do programa objeto produzem a amarração definitiva de um nome com o correspondente arquivo físico, evitando que haja alterações nesta correspondência. Quando o arquivo é aberto o seguinte comando é produzido:

$$A_{i--} := P_{j--}$$

onde P_j é o sub-processo em que ocorreu a abertura.

A variável A_{i--} associada ao arquivo A permite o controle do seu estado. Sempre que um arquivo é utilizado num sub-processo, o gerador deve providenciar que ele esteja aberto quando tal evento ocorrer durante a execução. Assim, no início da etapa de um sub-processo em que se dá a sua utilização, ele deve ser aberto, caso ainda não o foi. A convenção empregada é a de que ele deve ser fechado no final da etapa do sub-processo em que foi aberto.

Se um arquivo A é utilizado numa etapa qualquer de um sub-processo, devem ser produzidos os comandos de verificação do seu estado e de sua abertura. Este último deve ser executado no caso do comando de verificação apurar que tal arquivo está fechado. No final desta etapa, deve ser gerado o comando que verifica se o arquivo foi aberto no sub-processo corrente. Analogamente, deve ser produzido o comando de fechamento a ser executado em caso afirmativo. Para

um arquivo identificado pelo usuário com o nome *Notas* e utilizado numa certa etapa de um sub-processo *Grava_Notas*, para inserção de novos registros, o gerador produz, no início desta etapa, o seguinte comando, extraído de um exemplo de programa obtido pelo gerador:

```
if(NOTAS__=CLOSED_)then
begin
  {$I-}reset(NOTAS);{$I+}
  if ioresult <> 0 then rewrite(NOTAS)
  else seek(NOTAS,filesize(NOTAS));
  NOTAS__:=GRAVA_NOTAS__;
end;
```

No final da mesma etapa, o seguinte trecho foi produzido pelo gerador, com relação ao fechamento do arquivo.

```
if(NOTAS__=GRAVA_NOTAS__)then
begin
  close(NOTAS);
  NOTAS__:=CLOSED_;
end;
```

No caso de arquivos, que são objetos de comandos de seleção, pressupõe-se que eles estão fechados, pois cada conjunto abstrato é tratado por um sub-processo e não teria sentido um mesmo conjunto ser tratado por dois sub-processos simultaneamente. Estes arquivos são abertos no início da etapa por comandos de inicialização produzidos pelas especificações de seleção como será visto mais adiante em 5.3.

A princípio deveria ser anexada à declaração uma informação sobre o modo de utilização dos arquivos, indicando se ele é utilizado como entrada, saída ou para ambas as operações. Ela serviria para subsidiar o gerador na escolha dos parâmetros corretos na produção dos comandos de abertura. Na maior parte dos casos esta informação poderia ser obtida a partir das especificações fornecidas pelo usuário, identificando se existem para um determinado arquivo somente comandos de gravação, ou somente de leitura, ou de ambos os tipos, numa determinada etapa. O problema ocorre quando existem somente comandos de gravação e se deseja concatenar novos dados a um arquivo já existente. O gerador não teria como distinguir esta forma daquela em que se deseja regravar o arquivo desde o início. Uma possível solução até razoavelmente interessante seria entender que sempre que ocorrem somente comandos de inserção e o arquivo físico já existe, então devemos considerar que o usuário deseja aumentar o arquivo existente. Caberia ao usuário eliminar o arquivo, antes de ativar o programa gerado, se sua intenção for regravar todo o arquivo de saída. Esta hipótese assumida com relação aos arquivos não implica em grandes prejuízos, visto que ele não apagará arquivos existentes. O maior mal causado é acrescentar novos registros num arquivo, quando se desejava iniciá-lo com estes dados. Mas este mal é reparável, bastando reexecutar o programa com a mesma entrada e com o arquivo previamente apagado por ação manual do operador. Implementamos esta estratégia no gerador. Para implementá-la é necessário produzir comandos que verifiquem se um certo arquivo físico já existe, como pode ser visto no último exemplo.

Como podemos observar no exemplo das variáveis produzidas a partir da especificação de um arquivo, é gerado um tipo registro. No caso mostrado, trata-se de um registro simples em que todos os campos são elementares. A declaração do tipo é feita por um algoritmo recursivo que possibilita declarar tipos de registros com campos e sub-campos de profundidade¹ teoricamente arbitrária. O seguinte esquema do algoritmo utilizado efetua esta tarefa.

Algoritmo: Declara_Tipos.

Importação: Registro X a ser declarado.

Saída: Declaração de X.

início

 sejam $x_1, x_2, \dots, x_n$ os campos de X.

 para cada campo x_i de X

 se x_i é elementar

 então produza ' x_i : Tipo_de_ x_i ;'

 senão produza ' x_i = record'

 Declara_Tipo(x_i)

 produza 'end;'

fim

A primeira chamada ao algoritmo faz a abertura do tipo principal, com os seguintes comandos:

 produza 'T_X_ = record'

 Declara_Tipos(X)

Desejando uma saída “indentada” da declaração de um tipo registro, basta utilizar a profundidade da recursão como um mecanismo para esta finalidade, ou seja, utilizar um parâmetro que é incrementado na própria chamada.

5.2.2 Tabelas

Tabelas são relações como no Modelo Relacional de Dados [Co 70], isto é, são entendidas como um conjunto de linhas agrupadas em colunas. Cada coluna, aqui denominada de *campo*, tem associado a ela um *nome*, e pode ser uma outra tabela. A cada linha é associado um índice interno, que pode ser referenciado por meio de um campo implícito de nome *Current*. Assim, a referência ao índice é feita como se faz a um campo qualquer. Considere o seguinte trecho de especificações:

```
%table Tabela[0..100] (Simbolo:string[20];
                       Colisao:integer)
```

O Gerador produz as seguintes declarações para o tratamento da tabela:

¹Um registro pode ser representado como uma árvore em que a raiz contém o nome do registro e, recursivamente, cada vértice tem nos seus filhos os seus respectivos campos.

```

type
  T_TABELA_=record
    SIMBOLO:string[20];
    COLISAO:integer;
  end; (* record *)
var
  TABELA:array[0..100] of T_TABELA_;
  TABELA_7_:integer;

```

Para cada tabela Tb , com índices de n_1 a n_2 que são os limites especificados logo após o nome da tabela (0 e 100, no exemplo), é produzido um tipo $T_Tb_$, que é um registro. A tabela é tratada como um vetor com elementos deste tipo, como podemos observar no exemplo. Também para cada tabela é criado um apontador para o elemento corrente. Qualquer referência à tabela Tb é feita pelo usuário pelo nome dado à tabela. O gerador substitui estas ocorrências pelo elemento corrente. O usuário sempre refere-se ao elemento corrente da tabela e, conseqüentemente, o gerador precisa gerar a indexação do elemento correto. Uma ocorrência da tabela Tb numa expressão é substituída por $Tb[Tb_n_]$ no código objeto, onde $Tb_n_$ é declarado como o apontador do elemento corrente da tabela Tb , ou seja, é o índice da tabela. O valor de n (7, no exemplo), será explicado mais adiante. Deste modo, para cada tabela, são produzidas três declarações, a saber: o tipo dos elementos da tabela, o vetor matriz que implementa a tabela e a variável que indexa o vetor.

A princípio, poderíamos adotar, para índice de uma tabela, uma variável com o mesmo nome da tabela seguido de algum prefixo ou sufixo que as diferenciasses das demais, mas isto não é suficiente. Como existe a possibilidade de campos serem tabelas, um mesmo nome pode ser utilizado para uma tabela e para o seu campo que também é uma tabela. Isto produziria dois índices com o mesmo nome. Várias soluções são possíveis. Uma delas é declarar índices de tabelas, que não são campos, como variáveis e índices de tabelas, que são campos, também como um campo do mesmo registro. Isto provocaria um código mais extenso no acesso a um campo de um elemento da tabela. Para exemplificar, suponha uma tabela Tb que possui um campo de mesmo nome e que também é uma tabela e I uma cadeia de caracteres que é utilizada para índice de tabelas com o nome Tb . Assim, para indexar um elemento do campo Tb de um certo elemento da tabela Tb , teríamos que utilizar a sequência de símbolos seguinte

$$Tb[I].Tb[Tb[I].I]$$

que é custosa. Outra alternativa seria utilizar o apontador $Tb_I_$ para a vetor e $Tb_Tb_I_$ para o campo, aumentando o tamanho do identificador à medida que a árvore que define o registro da tabela se aprofunda. A solução aqui adotada é mais simples. O apontador de uma tabela é uma “palavra” formada pelo seu nome, independentemente se é ou não um campo, seguida do sufixo que é o índice de entrada do símbolo na tabela de identificadores, produzida pelo gerador, durante a tradução. O nome e o sufixo são separados pelo caractere ‘_’ e a palavra é também encerrada pelo mesmo caractere. Isto garante a unicidade do apontador de uma tabela. Dentro desta solução, se tivéssemos adotado a estratégia de utilizar um contador sequencial para o sufixo, teríamos que guardar esta informação para saber qual o número se relaciona com o apontador de cada tabela. A opção mencionada tem a vantagem de que tal número é imediatamente obtido da localização do nome na tabela de identificadores do gerador, mas

apresenta a desvantagem de produzir uma numeração aparentemente aleatória para o usuário. Por razões de comodidade vamos nos referir à tabela de identificadores do gerador por *TIG*.

Qualquer referência à tabela *Tb* nas especificações se relaciona com o elemento corrente, apontado pelo indexador da tabela *Tb*. Deste modo, toda ocorrência do símbolo *Tb* numa expressão é substituída, pelo gerador, por

$Tb[Tb_n_]$,

onde *n* é o índice de entrada na *TIG* onde a tabela *Tb* está armazenada (7, no exemplo).

Como dissemos, o apontador de uma tabela genérica *Tb* é visto como um campo automático de nome "Current". Assim, ele pode ser obtido e utilizado nas expressões através da sequência de símbolos *Tb.Current*. O elemento corrente de uma tabela é automaticamente alterado por uma comando de seleção sobre uma tabela (vide 5.3) ou atuando diretamente sobre o apontador através de um comando de transformação (vide 5.5), como veremos adiante.

5.2.3 Telas

O gerador permite a utilização de telas, de modo que em lugar do usuário escrever os comandos para a sua construção nos pontos adequados do programa, ele descreve o formato da tela na parte de declarações e o gerador produz a partir dessa descrição os comandos para a sua exibição e processamento. Seria desejável que existisse um mecanismo sofisticado de desenho de telas integrado ao gerador, mas isto não faz parte do projeto. Em razão disto, as telas são declaradas, como se declara arquivos e tabelas, não possuindo facilidades para produção de telas complexas.

A declaração de uma tela é feita especificando os seus campos constantes e variáveis, com as suas características. No caso de campos constantes, a cadeia declarada aparecerá no vídeo sempre que a tela for apresentada. O seu atributo é a posição dentro do vídeo que é passada pela respectiva linha e coluna. O seu tamanho é deduzido da própria cadeia.

Os campos variáveis têm um nome e podem conter valores do tipo cadeia de caracteres ("strings"), inteiro ou real. Outros atributos são a linha e coluna que especificam a sua posição dentro do vídeo, o tamanho do campo e o número de decimais, se o campo comporta valores do tipo real. Um campo variável que, ao ser utilizado para entrada de dados, requer obrigatoriamente a digitação de pelo menos um caractere, ou seja, o campo não pode ser "saltado" apenas pressionando a tecla "Enter", deve receber o qualificativo *non_empty*. Suponha, agora, que uma certa tela utilizada para entrada de dados contém um campo que informa automaticamente o número de ordem do registro que está sendo digitado. Desta forma, todos os campos são de entrada, exceto este que é usado apenas como uma informação ao operador e que é mantido pelo programa. Na apresentação deste tipo de tela, o gerador precisa produzir um código que não solicite entrada de dados para o campo variável, mas que é informativo. Isto é feito através do qualificativo *output* que segue a declaração de campos desta espécie. Detectando esta palavra, o gerador fica sabendo que o referido campo é exclusivamente de saída, inclusive quando a tela está sendo usada para a entrada de dados, isto é, ele permanece protegido.

Quando uma tela está sendo usada para entrada de dados, estamos admitindo, nesta fase do trabalho, a tecla "Enter" como de finalização de um campo. Ou seja, pressionando esta tecla, o operador deseja informar que a digitação de um campo foi encerrada, embora, enquanto a tela toda não for completada, ele possa retornar a este campo para redigitá-lo. Esta mesma ação no último campo da tela, entendido como o último na lista de campos na declaração, produz a

transmissão da tela. Vamos chamar isto de *mecanismo trivial de transmissão de tela*. Digitando teclas ↑ e ↓, pode-se passear pelos campos para correções antes de efetuar a transmissão da tela. Assim, para o usuário é como se, encerrado um campo, o seu encerramento apenas produzisse a passagem para o próximo campo, sem que haja o fenômeno de transmissão do campo, para seu processamento. O usuário tem a visão de que ocorre a transmissão de toda a tela, após o encerramento de todos os seus campos. A despeito desta impressão, a implementação da leitura de uma tela é feita com transmissões de cada campo e o programa gerencia um possível retorno a um campo já transmitido.

Outras teclas de transmissão

Se for do interesse do usuário, podem ser definidas como teclas de transmissão de telas, quando utilizadas para entrada de dados, algumas (ou todas) das teclas de função entre *F1* a *F10*. Efetuada a transmissão de uma tela, é possível obter a informação de qual tecla foi utilizada para transmiti-la e utilizá-la em tabelas de decisão futuras. Se o usuário definiu teclas de transmissão para uma tela *Tl*, o gerador produz automaticamente um campo de nome **Transmit_Key**, podendo ser referenciado por *Tl.Transmit_Key*. Este campo assume um dos valores *F1* a *F10*, que são declarados automaticamente pelo gerador, dependendo de qual das teclas declaradas foi pressionada pelo operador. Tal campo só é declarado pelo gerador se na especificação da tela forem citadas as teclas de transmissão. Caso contrário, este campo não pode ser referenciado (nem existe no programa objeto) e a única forma de transmissão é o mecanismo trivial. Se existe a especificação de teclas de transmissão numa tela e durante a sua utilização para entrada de dados, for acionado o mecanismo trivial de transmissão, a ação corresponde a pressionar a primeira tecla de transmissão da lista fornecida para este fim. Assim, consultando o campo *Transmit_Key* desta tela, observa-se o valor da primeira tecla de transmissão.

Outra forma de transmissão de uma tela é especificar na *Transmit_Key* a palavra **all**, informando ao gerador que qualquer tecla transmite a tela. A sua utilidade é para situações em que se emprega o termo “Pressione uma tecla para prosseguir”. Neste caso, também é definido o campo *Transmit_Key* e, portanto, ele pode ser referenciado nesta tela. Como qualquer tecla transmite a tela, todos os campos variáveis, se houver, serão considerados como exclusivamente de saída, mesmo que não tenham o atributo **output** ou sejam qualificados de **non_empty**. Além disso, não é válida a especificação da tecla de encerramento (**end_key**), pois qualquer tecla já foi definida como sendo de transmissão, inclusive aquelas entre *F1* a *F10*.

Estas teclas de transmissão de tela têm o objetivo de assinalar o final da entrada de dados para os campos de uma tela, retornando ao controle do programa.

Para o funcionamento correto da tecla de transmissão, são declaradas as constantes *F1* a *F10* num módulo compilado em separado que são usadas no programa objeto para o tratamento de teclas de transmissão. Se estes símbolos forem desnecessários para o problema, é possível utilizá-los como variáveis, a critério do usuário, redefinindo-as na parte de declarações. Assim, o usuário pode redefinir os identificadores *F1* a *F10* como sendo, por exemplo, arquivos. Feito isto, ele não poderá utilizar as teclas redefinidas como teclas de transmissão, embora as demais, não redefinidas, possam continuar sendo usadas, normalmente. Como, a priori, as declarações podem ser feitas em qualquer ordem, é possível que o usuário declare uma tecla de transmissão, que, em declarações seguintes, serão redefinidas, ou vice-versa. Uma verificação da correta utilização destas redefinições pode parecer trabalhosa na medida em que é necessário armazenar as teclas

utilizadas e as redefinições ocorridas para que a cada nova declaração seja verificada a ocorrência de conflito. O procedimento adotado que supera o problema é inserir automaticamente o símbolo Fi , $1 \leq i \leq 10$, na *TIG*, à medida que ele ocorre como uma tecla de transmissão. Isto feito, qualquer tentativa de declarar uma variável com este nome será vetada por duplicidade. Analogamente, se tiver ocorrido a redefinição, o símbolo constará da *TIG* por ação do usuário e a inserção automática da constante, correspondente à tecla de transmissão, será impugnada, também por duplicidade.

Tecla de encerramento de uma tela

Podemos associar uma tela, quando utilizada para entrada de dados, a uma sequência de registros, um para cada tela transmitida, de forma análoga à sequência de elementos de um arquivo. No caso de arquivos propriamente ditos, o fim de dados é assinalado por uma marca de “fim de arquivo”, fisicamente detectado, enquanto no caso de uma sequência de telas, o seu final deve ser necessariamente informado pelo operador. Esta operação pode ser realizada utilizando teclas de transmissão que devem ser comparadas, no retorno, com o valor estipulado para esta finalidade, numa tabela de decisão de responsabilidade do usuário. Por exemplo, suponha que a tecla *F1* foi declarada para transmissão de tela, com o objetivo estabelecido pelo usuário, de funcionar como encerramento do arquivo de telas. Assim, a cada tela transmitida, o usuário deve especificar uma comparação entre o valor do campo **Transmit_Key** com *F1*. Sendo iguais, o fluxo de execução seria desviado para a finalização do sub-processo.

É possível, opcionalmente, especificar uma das teclas entre *F1* a *F10* como o fim da sequência de telas. A diferença no seu funcionamento em relação à opção anterior, é que automaticamente a entrada de dados através da respectiva tela é interrompida, sem que haja a transmissão do registro corrente. Ao detectar a tecla de finalização da sequência de registros associados à tela, as ações tomadas correspondem ao de encerramento de um arquivo por atingir o seu final. A tecla definida para esta finalidade é chamada de tecla de encerramento e declarada como **End_Key**.

Do modo como o gerador implementa as telas, em caso de haver a especificação de teclas entre *F1* a *F10* como sendo de transmissão e uma delas ser novamente citada como tecla de encerramento, o conflito é resolvido em favor da última.

Identificadores produzidos para uma Tela

O gerador produz para cada tela *Tl* dois procedimentos, um para inicialização de seus campos variáveis de nome **Initialize_Tl** e outro para apresentação da tela no vídeo de nome **Display_Tl**. A inicialização é invocada no início da execução do programa e sempre que o usuário desejar “limpar” a tela. A apresentação da tela é efetuada quando é executada a entrada de dados através da tela e todas as vezes que algum comando modifica o conteúdo de algum campo variável da tela.

Para possibilitar a execução das tarefas relativas a uma tela *Tl*, o gerador produz a declaração de um tipo de nome **T.Tl**, onde *T* é a própria letra “T”, que é um registro contendo, como elementos, os seus campos variáveis. Os campos constantes são utilizados somente no procedimento de apresentação da tela no vídeo. Além dos campos variáveis, se houver a especificação de teclas de transmissão, é produzido um campo de nome “**Transmit_Key**”, onde fica armazenada a tecla utilizada na transmissão do registro, para o programa, através da tela. Para

cada campo variável C do tipo real ou inteiro, é declarado um campo auxiliar de nome $C_$ do tipo "string", que é utilizado para armazenar a cadeia da forma como digitada pelo operador para um eventual retorno ao campo antes da transmissão da tela. O tamanho do campo auxiliar é o tamanho declarado para o campo correspondente.

Para cada tela Tl , é declarada uma variável $Tl.n_$, onde n é o índice de entrada de Tl na TIG . Ela não é acessível ao usuário e tem a finalidade de indicar, na execução do código objeto, se ocorreu ou não o final da sequência de registros associados a uma tela utilizada para entrada de dados. Também é produzida uma variável $Tl.Buf_$, que é um registro do tipo $T.Tl_$, acima mencionado, onde ficam armazenados os campos variáveis. Toda referência a Tl numa expressão é substituída por $Tl.Buf_$.

Sempre que uma tela é modificada (1 ou todos os seus campos), o gerador produz um comando de apresentação da tela com os novos valores. Do mesmo modo, se uma tela é empregada para entrada de dados, ela é apresentada no vídeo quando a ação de seleção é iniciada. Neste caso, a tela mostra o conteúdo corrente dos seus campos variáveis, a serem alterados durante a tarefa de digitação de seus novos valores. Desejando que a tela esteja "limpa" para a entrada de novos dados, é necessário que seja especificado o comando de "limpeza" (vide 5.5.4). Estando na etapa M , tal comando é repetido para cada iteração.

Um fato relativamente sem importância é que, se uma tela está presente no vídeo e, na execução do código, é iniciada a seleção de dados através da mesma tela, ocorrerá uma nova apresentação da mesma tela, sem que haja qualquer alteração nos seus campos, provocando uma sensação desagradável ao operador de lentidão e de código redundante. Para contornar este efeito, sempre que as especificações ao gerador contêm uma tela, é produzida uma variável que armazena qual das telas está ocupando correntemente o vídeo e serve para controlar se uma determinada tela deve ser apresentada ou não quando utilizada para entrada de dados. Assim, para um arquivo com especificações ao gerador contendo telas $T_1, T_2, \dots, T_n$, é produzida a seguinte variável:

Current_Screen_: (NIHIL_, $T_1_$ _, $T_2_$ _, ..., $T_n_$ _)

que é inicializada com o valor NIHIL_, no início da execução do código gerado, e atualizado com o valor $T_i_$ sempre que a tela T_i for exibida no vídeo. Também deve ser alterado para NIHIL_, quando outros comandos não relacionados com telas produzirem saída no vídeo, o que ainda não acontece com os programas produzidos pelo gerador.

A **end_key** não produz nenhuma variável adicional; a sua presença somente determina a geração de código diferente na entrada de dados através da tela. Por definição, ocorrendo a tecla de encerramento, a tela corrente não é transmitida e o fim da sequência de registros é assinalado, alterando o valor da variável controladora do fim do "arquivo" de telas. A entrada de dados é feita através da especificação dos comandos de seleção (vide 5.3).

Para a entrada de dados através de uma tela é necessário um procedimento que assume o controle do teclado de acordo com os atributos do campo que está sendo lido. Isto é feito através de um módulo compilado em separado e que contém uma rotina que é invocada pelos comandos produzidos pelo gerador. É exatamente neste módulo que são declaradas as constantes $F1$ a $F10$. Também é declarado um vetor de nome **Vector_Screen_** que armazena informações sobre quais são as teclas de transmissão e de encerramento de telas, permitindo a sua imediata detecção, sem a necessidade de comparar a entrada com cada uma das teclas definidas na tela. O vetor acima é indexado pelos caracteres #0 a #255 e comporta os valores:

- 1, se o caracteres correspondentes são teclas de transmissão;
- 2, se o caractere correspondente é a tecla de encerramento;
- 0, se as teclas correspondentes não são utilizadas para estas duas finalidades.

Deste modo, para cada caractere “c” de entrada, basta verificar a *c*-ésima posição do vetor, para detectar as suas propriedades relativas à transmissão e encerramento. Implementado desta forma, não seria difícil estender a capacidade de se especificar como teclas de transmissão outras teclas além de *F1* a *10*.

O exemplo a seguir mostra como é especificada uma tela para o gerador.

```
%screen Tela (Const 'ENTRADA DE DADOS':1,5;
    Const 'Nome:' :4,5;
    Const 'Numero:':4,55;
    Const 'Nota1:':6,5;
    Const 'Nota2:':8,5;
    Const 'Nota3:':10,5;
    Const 'Nota4:':12,5;
    Const 'Pressione F1 para encerrar':14,1;
    Const 'Pressione F2 para transmitir':15,1;
    Var Nome:string,non_empty,4,12,30;
    Var Numero:integer,non_empty,4,63,3;
    Var Nota1:real,non_empty,6,12,3,1;
    Var Nota2:real,non_empty,8,12,3,1;
    Var Nota3:real,non_empty,10,12,3,1;
    Var Nota4:real,non_empty,12,12,3,1;
    Var Msg:string,output,18,5,16;
    Transmit_Key:F2,F5,F8,F9,F10;
    End_Key:F1)
```

Esta especificação produz as seguintes declarações no programa objeto, extraído de um trecho de um exemplo de programa obtido pelo gerador.

```
type
  T_TELA_=record
    NOME:string[30];
    NUMERO:integer;
    NUMERO_:string[3];
    NOTA1:real;
    NOTA1_:string[3];
    NOTA2:real;
    NOTA2_:string[3];
    NOTA3:real;
    NOTA3_:string[3];
    NOTA4:real;
    NOTA4_:string[3];
    MSG:string[16];
    TRANSMIT_KEY:char;
```

```

    end; (* record *)
var
    TELA_3_: integer;
    TELA_Buf_: T_TELA_;
    CURRENT_SCREEN_: (NIHIL_, TELA__);

```

No início da execução do programa objeto, é invocado automaticamente o procedimento que faz a inicialização dos campos variáveis da tela e, como vimos, a variável `CURRENT_SCREEN_` é inicializada com o valor `NIHIL_`. No exemplo anterior não são mostrados os procedimentos relacionados à tela que fazem a sua inicialização e apresentação no vídeo.

5.2.4 Sequências

Como vimos rapidamente em 5.1.2, a fase de seleção dentro da etapa principal produzem os “loops” através dos comandos de seleção que apresentaremos em 5.3. Este mecanismo repetitivo faz com que as especificações da etapa M sejam executadas sobre todos os elementos de um conjunto abstrato. Uma possibilidade para estes comandos é varrer o conjunto dos naturais, na verdade, a sequência dos naturais, pois normalmente a ordem é relevante. Para implementar esta possibilidade, o gerador produz declarações distintas dependendo da forma como a sequência é descrita nas especificações. São permitidas sequências de inteiros (negativos ou não) e de caracteres. As duas podem ser expressas por uma enumeração de todos os elementos da sequência ou por um intervalo. As sequências e seus elementos são constantes e somente podem ser utilizados quando selecionados. Assim, uma referência à sequência significa uma referência ao elemento correntemente selecionado, não necessariamente na etapa corrente. A sua utilização, sem prévia seleção, acessa o primeiro elemento da sequência, visto que, no início da execução do código objeto, são inicializadas com o primeiro valor.

A especificação de uma sequência por intervalo é feita usando exatamente a notação de intervalo em Pascal, consistindo do limite inferior, seguido do limite superior e separados por ‘..’. Para cada sequência desta espécie é produzida uma variável do mesmo tipo dos seus elementos. O valor desta variável é incrementado a cada iteração do “loop”, produzido pelo comando de seleção que atua sobre a sequência.

A especificação de uma sequência S pela enumeração de todos os seus n elementos $s_1, s_2, \dots, s_n$, produz um vetor constante de n células, começando pela de número zero, de tal modo que o conteúdo da $(i - 1)$ -ésima célula de S é s_i . É declarado ainda um apontador para o elemento corrente do vetor. Para isto, correspondente à sequência S , temos no programa objeto a declaração de:

- Um tipo `T.S_` que é um vetor com n células que admitem valores dos elementos da sequência;
- Um vetor constante `S` do tipo `T.S_`;
- Um apontador `S_m_` para o vetor `S`, onde m é o índice da *TIG* onde está armazenado o símbolo S .

Segue um exemplo de especificação de duas sequências com elementos inteiros, a primeira delas definida por intervalo e a segunda por enumeração completa dos seus elementos.

```
%set Naturais = {0..100}
%set Conjunto = {2,4,6,84,20}
```

A seguir, podemos observar declarações relativas às duas sequências acima, obtidas de um trecho de um exemplo de programa produzido pelo gerador.

```
type
  T_CONJUNTO_=array[0..4] of integer;
const
  CONJUNTO:T_CONJUNTO_=(2,4,6,84,20);
var
  NATURAIS:integer;
  CONJUNTO_10_:integer;
```

5.2.5 Variáveis de Sub-processos de agrupamento

Nos sub-processos de agrupamento (vide 5.1.3), as funções de agregação disponíveis são **Count**, **Max**, **Min**, **Total** e **Average**, com a interpretação apresentada. Com exceção da primeira, todas as funções devem ser acompanhadas do campo sobre o qual se quer computá-la. A função **Count** contabiliza o número de iterações do “loop”, que executa o sub-processo de agrupamento, ou seja, o número de elementos de um sub-conjunto da partição do conjunto abstrato. Estas funções ocorrem somente na etapa *E* de um sub-processo de agrupamento e assim que são detectadas, devem ser armazenadas para produzirem a declaração de variáveis necessárias ao atendimento destas funções e à sua atualização.

Os sub-processos de agrupamento podem ocorrer depois de serem referenciados e, deste modo, não é possível produzir os comandos de atualização necessários, nos locais em que as referências ocorrem, porque ainda desconhecemos quais funções são requeridas. Conseqüentemente, a inclusão destes comandos e declarações fica para o segundo passo da conversão.

Num mesmo sub-processo podemos encontrar todas as funções de agregação e a mesma função atuando sobre campos diversos. Por exemplo um problema pode requisitar os valores máximos de salários e de consignações da folha de pagamento de uma empresa, sendo que o agrupamento é feito pelo setor. Assim, num mesmo sub-processo de agrupamento ocorrem referências a

%max(Salário) e %max(Consig).

Para o perfeito funcionamento deste mecanismo, precisamos de variáveis distintas para a mesma função de agregação. No exemplo acima, uma para armazenar o valor máximo de salário e outra para o valor máximo de consignações. Além disso, precisamos de diferentes variáveis para as mesmas funções em diferentes sub-processos de agrupamento. Pois, no caso do exemplo, é possível que estes dois valores máximos fossem requeridos também agrupados em departamentos, que englobam os setores. Assim, só no exemplo particular, precisamos de quatro variáveis distintas para armazenar os valores máximos de Salário e Consignações por Setor e por Departamento. Também poderíamos desejar obter os valores mínimos destes campos para os mesmos agrupamentos. Portanto, para um sub-processo de agrupamento *SP*, declaramos as seguintes variáveis:

- **Count_SP_**, se ocorre as funções **Count** ou **Average**.
- **Max_SP_Cn_**, **Min_SP_Cn_**, **Total_SP_Cn_**, se ocorrem as funções **Max**, **Min**, e **Total**, respectivamente, onde **C** é o campo sobre o qual a função está sendo computada e **n** o índice da *TIG* onde o símbolo **C** está armazenado. Este índice é necessário, pois podem ocorrer dois campos de duas estruturas distintas, com o mesmo nome.

A variável **Total_SP_Cn_** também é declarada pelo gerador se ocorre a função **Average(C)** na etapa final do sub-processo de agrupamento *SP*.

Pelo exposto, a ocorrência da função **%Count** numa expressão da etapa de finalização de um sub-processo de agrupamento é substituída por **Count_SP_**. Analogamente, a ocorrência de **%Max(E.C)**, como acima, computando o valor máximo do campo *C* da estrutura de dados *E* é substituída por **Max_SP_Cn_**, onde **n** é o índice da *TIG* onde está o campo *C*. O mesmo ocorre quando as funções **Min** e **Total** são encontradas. De forma semelhante, se a função **%Average(E.C)** ocorre numa expressão, ela é substituída por **(Total_SP_Cn_ / Count_SP_)**.

A inicialização destas variáveis de agrupamento e os comandos que as atualizam são produzidos pelo gerador num segundo passo da conversão. Os primeiros são inseridos na etapa inicial do sub-processo de agrupamento e os segundos no ponto em que ocorre a seleção por agrupamento, logo após os comandos produzidos para a seleção do próximo elemento do conjunto abstrato e para verificação do agrupamento (vide 5.3).

Para que, no segundo passo da conversão, possa ser feita a geração dos comandos acima, a tabela com os elementos do método, mantida pelo gerador, durante a tradução, deve guardar informações sobre quais funções de agregação são invocadas e sobre quais campos elas atuam, num sub-processo de agrupamento. Além disso, no arquivo intermediário é assinalado o local em que devem ser colocados os comandos de atualização das variáveis de agrupamento, junto com a informação de qual o sub-processo de agrupamento que as referencia.

Os sub-processos de agrupamento têm a sua parte de inicialização executada sempre que houver uma alteração no valor do campo e também no primeiro elemento do conjunto abstrato. Para que o funcionamento seja correto, introduzimos duas variáveis de controle do agrupamento para cada nível. Seja *C* o campo de agrupamento. As variáveis

C_Pn_ e **C_An_** ,

onde *n* é o índice da *TIG* onde está o campo *C* e *P* e *A* são os próprios caracteres '*P*' e '*A*', respectivamente. **C_Pn_** é a sinalizadora do primeiro evento, indicando que deve ser acionada a etapa de inicialização do sub-processo de agrupamento em caso afirmativo. **C_An_** armazena o valor anterior do campo de agrupamento *C*. Seu valor comparado ao valor corrente do campo indica se devem ou não ser executadas as etapas de finalização e inicialização do sub-processo de agrupamento.

5.2.6 Variáveis simples e auxiliares

A forma como os elementos do método I-M-E são especificados acaba por reduzir drasticamente o uso de variáveis temporárias por parte do usuário. Normalmente, ele fará referência somente às estruturas mais complexas, como arquivos e tabelas. As variáveis e "buffers" necessários para a manipulação dessas estruturas serão geradas automaticamente. Nesta tarefa, além dos identificadores relacionados com essas estruturas, são necessários alguns temporários para o

controle do programa. Deste modo o gerador produz a declaração de variáveis auxiliares da forma I_n - para as do tipo “inteiro”, Ch_n - para as do tipo “caractere”, R_n - se for “real” e S_n - para as cadeias de caracteres (“strings”). Em todos estes casos, o índice n é um número seqüencial que começa com 1 e é incrementado à medida que novos temporários se tornem necessários.

A despeito da pouca utilização de variáveis simples por parte do usuário, ele pode especificá-las ao gerador que produzirá a sua declaração normal em Pascal.

5.2.7 Sinônimos

Suponha que se deseja comparar ou processar simultaneamente dois elementos de um arquivo, por exemplo, em um *auto-relacionamento* [Se 87]. O efeito seria o mesmo se tivéssemos duas cópias idênticas do arquivo e quiséssemos fazer a comparação entre os elementos dos dois arquivos. Dizemos que A e B são sinônimos se ambos são nomes de um mesmo conjunto de dados. Isto possibilita referenciar dentro de uma etapa de um sub-processo dois elementos distintos de um mesmo conjunto (arquivo ou tabela). Da forma como utilizamos os conjuntos estruturados², o nome A atribuído a um arquivo referencia um de seus elementos. Se quisermos referenciar simultaneamente um outro registro deste mesmo arquivo, basta utilizar um sinônimo de A que fará referência a um outro elemento do mesmo arquivo. O efeito obtido é o mesmo que imaginar a existência de várias cópias de um mesmo conjunto estruturado com nomes distintos. Para arquivos, os sinônimos são especificados na própria declaração, colocando-se em lugar de apenas um nome, uma lista deles. Na especificação

`%file  $A_1, A_2, \dots, A_n$  (...) ...`

o gerador assume que os nomes que estão na lista são sinônimos. Para isto, é produzida uma declaração de variáveis como visto na parte que tratava de um único arquivo, feito em 5.2.1, somente para o identificador A_1 . Os demais são incluídos na lista de declaração da variável descritora do arquivo e do “buffer” de armazenamento e da enumeração com os nomes dos sub-processos.

Para variáveis do tipo tabela, a declaração de sinônimos pode ser feita de modo análogo ao descrito para arquivos, utilizando uma lista de nomes. Neste caso, o gerador produz uma única tabela como no caso de um único nome (vide 5.2.2), atribuindo-lhe a identidade do primeiro da lista. Além disso, é produzida a declaração de uma lista de apontadores para a tabela.

Para as seqüências definidas por intervalo, o sinônimos produzem a declaração de uma lista de variáveis do tipo dos elementos da seqüência. Se a seqüência é definida pela enumeração de seus elementos, então é produzida uma lista de apontadores para o vetor que armazena os seus elementos. O vetor constante é produzido somente para o primeiro nome da lista. Também neste caso, os sinônimos são introduzidos pela lista de nomes na declaração.

Outra forma alternativa para especificação de sinônimos é mostrada a seguir:

`%synonym  $A_1, A_2, \dots, A_n = A$`

que apresenta os nomes da lista como os sinônimos de A , previamente declarado.

²Usaremos o termo conjuntos estruturados para referir-nos a seqüência de dados como arquivos e tabelas.

5.3 Comandos de Seleção

Como vimos, as especificações no método I-M-E dividem cada sub-processo em três partes fisicamente separadas e cada uma delas cumpre o seu papel em três fases também fisicamente separadas: seleção, escolha e transformação. A fase de escolha é implementada pelas tabelas de decisão, as transformações por comandos de transformação e a seleção por comandos de seleção que passaremos a tratar.

O comando de seleção obtém os dados de uma estrutura de dados para o seu processamento. Estando na etapa M , determina o início de um “loop” que trata de cada um dos elementos do conjunto abstrato associado ao sub-processo. Os comandos de seleção desta etapa servem para entregar ao sub-processo o próximo elemento do conjunto abstrato. Na verdade, são os comandos de seleção que definem, através de suas propriedades, o conjunto abstrato a ser tratado pelo sub-processo. Descreveremos o comando de seleção para a etapa M . Nas demais etapas, a descrição é a mesma, exceto que o gerador não produz códigos que executam “loops”. A forma geral de um comando de seleção é a seguinte.

$$\begin{array}{l} \%select (m) E_1 \text{ where } S_1 \text{ when } D_1, \\ \quad E_2 \text{ where } S_2 \text{ when } D_2, \\ \quad \vdots \\ \quad E_n \text{ where } S_n \text{ when } D_n \\ \text{grouping by } A_1 \text{ exec subprocess } SP_1 \\ \quad A_2 \text{ exec subprocess } SP_2 \\ \quad \vdots \\ \quad A_p \text{ exec subprocess } SP_p \end{array}$$

$E_1, E_2, \dots, E_n$ são identificadores de variáveis, telas, tabelas ou arquivos cujos elementos devem ser selecionados³ chamaremos de *ítems do comando de seleção*. Esses ítems são a única parte obrigatória do comando. Chamaremos

$S_1, S_2, \dots, S_n$ de *condições de seleção*
 $D_1, D_2, \dots, D_n$ de *condições de acesso*
 $A_1, A_2, \dots, A_p$ de *ítems de agrupamento* e
 $SP_1, SP_2, \dots, SP_p$ de *sub-processos de agrupamento*.

Na fórmula geral acima, (m) é uma expressão opcional do tipo inteiro que especifica um limite superior para o número de iterações do comando de seleção. Chamaremos esta expressão de *limite de iterações*. Se considerarmos uma etapa M , com um só comando de seleção, atuando sobre um arquivo de nomes de alunos $A = \{N_1, N_2, \dots, N_p\}$, num processo que deseja tratá-los, o conjunto abstrato associado será $C = A$ se as condições de seleção e acesso e o limite de iterações não foram especificados. Se o limite m foi especificado, mantidas as demais condições, teríamos

³A seleção corresponde à tradicional “leitura” nas linguagens algorítmicas o que, segundo V. W. Setzer, está errado pois o computador não “lê”; só o ser humano “lê”.

$C = \{N_i : i \leq m\}$ como o conjunto abstrato. Desta forma, a expressão que fornece o limite de iterações é um mecanismo de encerramento do “loop” determinado pelo comando de seleção da etapa principal. Em outras situações, o seu encerramento pode ser provocado pela exaustão dos elementos selecionáveis do conjunto abstrato, sem que o número máximo estipulado pelo limite de iterações seja atingido e esta será a única forma, se o limite não for especificado. Na verdade, o conjunto abstrato é um ente imaginário e os comandos devem atuar sobre os objetos declarados pelo usuário, utilizando as especificações dos comandos de seleção, na tentativa de construir estes conjuntos abstratos. Assim, os mecanismos de encerramento de “loops”, que teoricamente verificam se todos os elementos do conjunto abstrato já foram tratados, devem ser traduzidos em termos de operações sobre os conjuntos definidos pelo usuário. Eventualmente, o conjunto abstrato coincide com algum conjunto físico.

5.3.1 Ítens do Comando de Seleção

Cada item do comando de seleção contribui com um de seus elementos na formação do conjunto abstrato C . Se $n = 1$, C é um subconjunto de E_1 . O fato de ser ou não propriamente contido depende da presença e atuação da condição de seleção associada ao item E_1 . No caso de $n > 1$, o conjunto abstrato determinado pelo comando de seleção é uma t -upla, ($t \leq n$), onde cada componente é obtido de itens distintos. Normalmente, $C = \{e_1, e_2, \dots, e_n : e_i \in E_i\}$, mas, dependendo da condição de acesso e de sua interpretação no problema específico, podemos ter $t < n$, como no exemplo da seção 5.3.3.

5.3.2 Condição de Seleção

A condição de seleção especifica uma condição que os elementos selecionados devem satisfazer, ou seja, uma das propriedades dos elementos do conjunto abstrato é satisfazer ao predicado contido na condição de seleção. Deste modo, se C é o conjunto abstrato e p o predicado especificado na condição de seleção, então para todo $c \in C$, $p(c) = \text{verdadeiro}$. O gerador implementa esta especificação através de comandos de leitura do arquivo e de um comando que verifica se o registro lido satisfaz a condição, de tal sorte que o comando de seleção entrega ao processamento da etapa M , somente os registros válidos. Procedimento análogo produz o resultado desejado sobre uma tabela. Assim, se o comando de seleção opera sobre um arquivo A e a condição de seleção é:

$$A.F = 'X',$$

então são selecionados somente os registros de A , cujo campo F contém o valor ‘X’. A implementação da condição de seleção funciona de forma a descartar todos os registros que não satisfazem às propriedades por ela estipuladas, passando a analisar o próximo registro. Como podemos ver, não é possível saber, a priori, quais registros compõem o conjunto abstrato; assim, o gerador produz comandos que analisam exaustivamente cada um dos possíveis elementos, de modo a transferir para o processamento da etapa principal somente aqueles que são selecionáveis. Este trabalho é produzido pelo gerador; ao usuário fica a impressão de uma “caixa preta” que calcula o conjunto abstrato a ser tratado pelo sub-processo.

No código objeto, o comando de seleção produz um rótulo que determina o ponto inicial do “loop” que chamaremos de *base do loop*. Cada item de seleção determina um rótulo no

programa que vamos designar de *rótulo do item*. Os rótulos tem a forma $SP.S.n$ -, onde SP é o nome do sub-processo que contém o comando, S é a própria letra "S" e n é um número inteiro que começa com 1 e é incrementado à medida que novos itens de seleção e novos comandos aparecem no sub-processo. A base do loop e o rótulo do primeiro item são os mesmos, desde que não haja limite de iterações.

Detalhando a implementação da condição de seleção, a expressão lógica especificada é testada após a leitura de um elemento. Caso ela seja verdadeira, o fluxo de execução continua na sequência dos comandos. Caso contrário, é produzido um desvio para o rótulo do respectivo item.

5.3.3 Condição de Acesso

A condição de acesso é uma condição que determina se um certo item deve ou não ser selecionado. Na fórmula geral, se a condição D_i não é satisfeita, o item E_i correspondente não é selecionado na iteração corrente. Caso contrário deve ser selecionado.

A condição de seleção é um predicado que atua sobre os campos do item selecionado, para informar quais elementos do conjunto devem ser selecionados. Diferente desta, a condição de acesso é um predicado lógico que normalmente não atua sobre os campos do item a ser selecionado, pois ela estabelece em que condições uma estrutura deve ser selecionada. Se tal predicado atua sobre a variável que é o próprio item de seleção, deve-se ter um cuidado especial na primeira iteração, pois ela não está definida antes desta e o resultado da avaliação da condição de acesso é que orienta a seleção. De qualquer forma, quando utilizada atuando sobre campos do próprio item, permite que a sua seleção esteja condicionada ao valor do registro anterior.

Para vermos como esta especificação altera a formação do conjunto abstrato de um sub-processo, vamos supor um exemplo. Considere um arquivo de alunos $A = \{a_1, a_2, \dots, a_n\}$, contendo entre outras informações o nome, um campo b que admite valores um e zero, dependendo se ele é bolsista ou não e um índice numérico n que é o índice de entrada para uma tabela de mensalidades. Existem duas tabelas de mensalidades X e Y , cujo campo *Valor* é o que deve ser cobrado do aluno da seguinte forma: Se ele é bolsista, o valor é obtido da n -ésima linha da tabela X , caso contrário, da n -ésima linha da tabela Y . Um processo que trata de imprimir os recibos de pagamento de todos os alunos trata do conjunto abstrato

$$C = \{(a_i, s_i) : s_i = \begin{cases} X_{n_i} & \text{se } b_i = 1 \\ Y_{n_i} & \text{senão} \end{cases}\},$$

onde $b_i = a_i \cdot b$ e $n_i = a_i \cdot n$.

Neste exemplo particular, o comando de seleção que produziria o conjunto abstrato correto seria

```
%select A,
  X where Current = A · n when A · b = 1,
  Y where Current = A · n when A · b = 0
```

5.3.4 Encerramento da Etapa Principal

Um comando de seleção produz um "loop" na parte M , cuja extensão começa fisicamente na base do loop. O encerramento da execução do loop que caracteriza a exaustão do conjunto abstrato,

ou eventualmente, parte dele, no caso de ser expresso por encaixamento de comandos de seleção, precisa ser traduzido em termos de comandos sobre as variáveis. Assim, estabelecemos a seguir os critérios de encerramento da execução dos loops, após um certo número de iterações:

1. O limite de iterações foi especificado no comando de seleção e já foi selecionado aquele número de elementos.
2. nenhum item foi selecionado no comando de seleção na última iteração.

O primeiro critério de encerramento é óbvio, pois o número limite de iterações já foi atingido. O segundo critério é adotado para encerramento, pois se nenhum item é selecionado no comando de seleção, um hipotético não encerramento provocaria, teoricamente, a execução dos comandos especificados sobre o mesmo elemento do conjunto abstrato selecionado na iteração anterior. O encerramento por este critério indica que cada item especificado já teve todos os seus elementos selecionados ou a condição de acesso correspondente tem valor *falso*.

Como normalmente um comando de seleção contém um único item, a condição de acesso pode ser utilizada, não somente como condição de leitura do item, mas também como de encerramento do loop.

5.3.5 Código para Comando de Seleção

Para implementar este comando com toda a sua generalidade, é preciso declarar um temporário *In* que inicia cada iteração com o valor zero, que é alterado para 1, se qualquer item for selecionado. No final do comando de seleção é possível verificar se houve a seleção de algum novo elemento ou se deve ser encerrado o loop correspondente. Vamos chamar esta variável de *controladora do loop*.

A variável controladora do loop é sempre alterada, quando ocorre a obtenção, com sucesso, de um item. Como vimos, ela é necessária para verificar o encerramento do loop determinado pelo comando de seleção, visto que um dos critérios de encerramento é a não seleção de nenhum de seus itens. Assim, ao final do sub-processo de seleção, precisamos ter armazenado a informação de que algum item foi selecionado. A variável controladora conterá o valor 1 em caso afirmativo, e zero em caso contrário. Sendo esta variável utilizada em função do comando de seleção poder conter vários itens, ela torna-se desnecessária se um único item está presente, mas como isto alteraria a estrutura do código gerado, implicando em ter que adiar a geração dos comandos e manter um armazenamento de informações em maior quantidade, o mesmo tratamento é adotado, independentemente do número de itens presentes num comando de seleção. Assim, ele produz, inicialmente, o rótulo indicando a base do loop e a inicialização da variável controladora com zero. Na parte final do comando de seleção, após serem gerados os comandos para todos os itens, é produzido o comando que chamaremos de *comando de controle do loop*.

Estamos tratando essencialmente do comando de seleção na etapa *M*. Se esse comando estiver nas etapas de inicialização ou finalização, os aspectos relacionados com o loop não devem ser considerados. Portanto, não são produzidos a variável controladora e os comandos de sua inicialização e nem o comando de avaliação do seu valor, para desvio de encerramento do loop. O rótulo da base do loop é produzido, pois corresponde ao rótulo do primeiro item do comando de seleção, em razão de, neste caso, não existir a especificação do limite de iterações.

Para cada item do comando de seleção, são produzidos, além do rótulo do item:

- Comandos de Inicialização: São os relativos à estrutura que está sendo selecionada e não somente ao registro que deve ser obtido. Um exemplo de comando de inicialização é a abertura de arquivos e a preparação do apontador de tabelas.
- Comandos de Leitura: São os relativos à obtenção do próximo elemento, incluindo a verificação das condições de seleção e acesso e da atualização da variável controladora do loop.

Como os comandos de leitura referem-se a cada novo dado da sequência de dados que compõem a estrutura que está sendo selecionada, eles devem estar após a base do loop. Ao contrário, como os comandos de inicialização devem ser executados antes da seleção do primeiro elemento e dizem respeito à estrutura como um todo, eles devem estar antes da base do loop, considerando que a ordem de execução dos comandos segue a sequência em que eles são especificados. Assim, somente podemos iniciar a gravação dos comandos de leitura do i -ésimo item após encerrada a gravação dos comandos de inicialização de todos os itens. Como o número de itens num comando de seleção pode ser arbitrariamente grande, é prudente produzir os comandos de leitura de um item assim que eles são detectados pelo gerador. Para isto, precisamos de um arquivo temporário para os comandos de leitura ou senão armazená-los na memória até o final do comando de seleção, para que antes possamos gravar todas as inicializações. O arquivo com o código gerado é organizado de forma que as especificações ao gerador são mantidas como comentário seguidas do código correspondente em Pascal. Assim, até mesmo os comandos de inicialização precisam esperar o encerramento do comando de seleção antes da sua gravação e passamos a ter a necessidade de dois arquivos temporários ou duas áreas de memória que armazenam os comandos de inicialização e de leitura, respectivamente, durante o período em que o gerador analisa um comando de seleção. Em razão da lentidão que caracteriza o manuseio de arquivos temporários, preferimos armazenar em duas áreas de memória no "heap", através de funções que simulam a gravação de texto em arquivos em disco.

Suponha um comando de seleção com itens $E_1, E_2, \dots, E_n$, que produzem comandos de inicialização $i_1, i_2, \dots, i_n$, comandos de leitura $l_1, l_2, \dots, l_n$, com rótulos de itens $r_1, r_2, \dots, r_n$ e a base do loop b . Teríamos o seguinte esquema do arquivo com o código objeto, lembrando que r_1 é o mesmo que b , se não existe a expressão que estabelece o limite de iterações:

(* especificação da seleção dos itens $E_1, E_2, \dots, E_n$
como comentário *)

Comando de inicialização da variável
contadora de iterações

i_1

i_2

:

b: Comando de inicialização da variável
controladora do loop

Comando de verificação da variável

```

                                contadora de iterações
Comando de atualização da variável
                                contadora de iterações
r1:    l1
r2:    l2
        ⋮
rn:    ln
Comando de controle do loop

```

Se existe a expressão m que especifica o limite de iterações, então uma variável auxiliar do tipo inteiro, que chamamos, no esquema acima, de *variável contadora de iterações*, é inicializada também com zero, antes de todos os comandos de inicialização que antecedem a base do loop. Imediatamente após a base do loop b , a variável controladora do loop é inicializada com zero. Segue a isto o comando de comparação da variável contadora de iterações com a expressão m e um desvio para o rótulo de encerramento do loop (vide 5.3.12) a ser executado se o número desejado de iterações já foi cumprido que, no esquema anterior, são chamados de comando de verificação da variável contadora de iterações. Ele implementa o mecanismo de encerramento do loop pelo número de iterações. Depois deste comando de verificação, é produzido o comando que atualiza a variável contadora de iterações, que simplesmente a incrementa.

O mecanismo de encerramento do loop pelo fato de nenhum item ser selecionado na iteração corrente é implementado pelo que chamamos, no esquema, de comando de controle do loop que consiste numa avaliação sobre a variável controladora do loop e de um desvio para um rótulo adequado, como veremos adiante na seção 5.3.12.

De um modo geral, a seleção de um item, seja um arquivo ou tabela, tem quase sempre as mesmas especificações. Não importa ao usuário se a estrutura de dados que está sendo tratada reside no disco ou na memória. O gerador, através de informações passadas nas declarações, deve produzir comandos adequados para a sua leitura. Agora, vamos descrever os códigos gerados para a seleção de cada item.

5.3.6 Código para Seleção de Arquivos

Vamos considerar um arquivo A como um item a ser selecionado no comando de seleção. Sejam C a condição de seleção, D a condição de acesso, SP o sub-processo e n o número de ordem do rótulo do item e Ip a variável controladora do loop. O seguinte código é produzido em Pascal, correspondendo aos comandos de leitura.

```

SP_S_n:  if (not eof (A)) and (D) then          (1)
        begin
            Le_um_registro_de A em A.Buf.;
            if not C then goto SP_S_n;           (2)
            Ip:=1;                               (3)
        end;

```

O comando "Le_um_registro_de" depende de se o arquivo A é do tipo texto ou estruturado. Se do segundo tipo, ele corresponde a


```
read(A,A_Buf_);
```

Para arquivos do tipo texto, o comando deve especificar os campos na ordem em que foram declarados. Assim, se A tem campos $C_1, C_2, \dots, C_n$, o comando corresponde a

```
readln(A,A_Buf_-C_1,A_Buf_-C_2,..., A_Buf_-C_n);
```

Estes são os comandos de leitura para o ítem A , que é um arquivo. Eles são colocados após o rótulo do ítem. Existem ainda os comandos de abertura do arquivo (“reset(A)”) que são colocados antes da base do loop e que chamamos, genericamente, de comandos de inicialização.

Se as condições de acesso e seleção (ou uma delas) estão ausentes, os comandos de leitura ficam ligeiramente modificados: A linha assinalada com (2) do esquema acima desaparece e a linha (1) perde o trecho “and (D)”. Se a etapa não é a principal, a linha (3) não é produzida.

5.3.7 Código para Seleção de Tabelas

Agora, vamos supor que o ítem do comando de seleção é uma tabela T . Sejam i e f os limites inferior e superior para o índice de T , C a condição de seleção, D a condição de acesso, SP_S_n o rótulo do ítem, Ip_- a variável controladora do loop e q o índice da tabela de identificadores do gerador, onde está a tabela T . O seguinte código é produzido para os comandos de leitura.

```
SP_S_n:  if (T.q_< s_2) and (D) then
          begin
            T.q_ := T.q_ + 1;
            if not C then goto SP_S_n;
            Ip_:=1;
          end;
```

No comando acima, $s_2 = f$. O comando de seleção para uma tabela permite especificar o intervalo de índices sobre o qual se quer fazer a seleção, da seguinte forma:

```
%select
:
T starting_at s_1 to s_2 where C when D,
:
```

Nesta especificação, s_2 é o limite superior da parte da tabela a ser selecionada. Esta forma de seleção é conveniente, pois, normalmente, as tabelas são declaradas com tamanho superior ao necessário, de modo que, a partir de um certo índice, os seus elementos não são significativos e não devem ser selecionados. O comando de inicialização é:

```
T.q_ := s_1 - 1;
```

Se cláusula **starting_at** está ausente, s_1 vale i .

Quanto à presença de cláusulas e à etapa em que o comando aparece, aplicam-se as mesmas observações feitas para arquivos.

A tabela admite também a seleção indexada, ou seja, em lugar de percorrer sequencialmente os seus elementos, é passado no comando de seleção o índice do elemento que se deseja selecionar. Na verdade, devido à visão relacional das tabelas, e lembrando que existe uma coluna virtual de nome *Current* ela corresponde a uma condição de seleção, pois deseja-se selecionar somente o elemento cujo campo “*Current*” da tabela tem um valor determinado. Como esta coluna é acessível em expressões utilizando a sequência de símbolos *T.Current*, para a tabela *T* isto pode ser feito especificando

```
%select
      :
      T where T.Current = s
      :
```

onde *s* é a expressão que determina o índice do elemento da tabela que se deseja selecionar. Embora o efeito do comando seja exatamente o pretendido, a forma como a condição de seleção é implementada, produz um código extremamente ineficiente, já que, por esta cláusula, o gerador produz comandos de leitura que varrem sequencialmente a tabela até encontrar um elemento que satisfaça a condição de seleção. Para contornar esta dificuldade, a cláusula pode ser especificada da forma seguinte:

```
where Current = s
```

que encontrada pelo gerador, produz código de acesso direto. Quando a palavra *Current* é encontrada, logo após o *where*, o gerador entende que deve produzir uma busca direta ao elemento cujo índice tem o valor calculado pela expressão *s*. É claro que se isto melhora um dos lados da questão, afeta outra, embora sem consequências. Normalmente, a condição de seleção é um predicado lógico arbitrário, podendo conter conectivos “and” e “or”. Utilizando esta sintaxe especial, diminuimos a expressividade da condição de seleção, mas isto não é um prejuízo, pois não tem sentido prático uma condição mais complexa, uma vez que só temos um único registro que satisfaz à restrição imposta por ela. Para este caso, o gerador produz os comandos de leitura, sem o comando de inicialização.

```
SP_S_n:  if (D) then
          begin
            T_q := s;
            Ip_ := 1;
          end;
```

5.3.8 Código para Seleção de Sequências

Como vimos, uma sequência é um conjunto ordenado de inteiros ou caracteres. Podem ser declarados pelo usuário, através de enumeração de todos os seus elementos ou de um único intervalo. As operações sobre a sequência, por parte do usuário, são independentes da forma como elas foram apresentadas, mas os comandos produzidos para os dois casos são distintos. A seleção de um item *S*, deste tipo, declarado por um intervalo *i..f*, produz o seguinte comando de inicialização

```
S := i - 1;
```

e os seguintes comandos de leitura

```
SP_S_n_:  if (S < f) and (D) then
            begin
              S := S + 1;
              if not C then goto SP_S_n_;
              Ip_:=1;
            end;
```

onde SP é o nome do sub-processo corrente, C a condição de seleção, D a condição de acesso e $Ip_$ a variável controladora do loop. Se a sequência foi definida através de enumeração de seus elementos, $s = \{s_1, s_2, \dots, s_l\}$, então o seguinte comando é produzido para a inicialização

```
S_q_ := 0;
```

onde q é o índice do elemento da TIG que armazena o símbolo S . Os comandos de leitura produzidos são exatamente os mesmos que aqueles que o gerador produz para tabelas. As diferenças ficam por conta do vetor que, neste caso, é uma constante e do seu limite superior que é obtido pelo gerador, a partir do número de seus elementos.

```
SP_S_n_:  if (S_q_ < l) and (D) then
            begin
              S_q_ := S_q_ + 1;
              if not C then goto SP_S_n_;
              Ip_:=1;
            end;
```

5.3.9 Código para Seleção de Telas

Quando estávamos tratando de declaração de variáveis, discutimos o uso de teclas de transmissão de telas (vide 5.2.3), que são teclas através das quais é sinalizado o fim da digitação dos campos associados a uma tela. Precisamos estabelecer qual o mecanismo de finalização de entrada de um campo, além das rápidas linhas já passadas sobre o assunto.

Teclas de Transmissão de Campo

Vamos adotar um sistema de transmissão de campo implícito, cujo funcionamento não pode ser alterado pelo usuário. Teoricamente, seriam possíveis diversos mecanismos, como por exemplo, o campo seria automaticamente transmitido quando ele tivesse todos os seus caracteres preenchidos. Vamos enumerar as três teclas de transmissão de campo, admitidas para a seleção através de telas, a primeira delas, incondicional e as outras duas condicionadas ao número de ordem do campo dentro da definição da tela. Seja uma tela T com campos variáveis editáveis $C_1, C_2, \dots, C_n$. As seguintes teclas são de transmissão de um campo C_i , que está sendo correntemente digitado:

- “Enter”. Transmite o campo C_i e assinala que o próximo a ser editado é C_{i+1} , se $i < n$. Caso contrário, transmite o campo e também a tela.

- “↑”. Se $i > 1$, transmite o campo C_i e assinala que o próximo a ser editado é C_{i-1} .
- “↓”. Se $i < n$, transmite o campo C_i e assinala que o próximo a ser editado é C_{i+1} .

Se o campo sendo editado não admite o valor vazio e é pressionada uma das teclas de transmissão de campo “Enter” ou “↓”, deve ser exibida uma mensagem de erro. A terceira opção de transmissão não precisa se manifestar, pois obrigatoriamente terá que repassar pelo campo corrente.

Rotina em Separado

Uma rotina para leitura de dados através de uma tela costuma ser extensa e nem sempre uma tarefa simples. Assim, em lugar de produzir no código objeto os comandos da linguagem Pascal que efetivam esta tarefa, a seleção deste tipo de item é implementada por uma série de chamadas a um procedimento, que lê do teclado uma cadeia de caracteres referentes a um campo. Tal procedimento é definido num módulo à parte e compilado separadamente. O seu nome é `Le_Strn_` e para o programa gerado constitui numa “caixa preta”. Sabemos apenas o que ele faz, o que devolve e o que deve receber. `Le_Strn_` lê uma cadeia de caracteres do teclado, respeitando certas restrições que são passadas como parâmetros e algumas outras que estão numa variável global. Recebe como parâmetros:

- (X, Y) . A posição inicial da cadeia, através da respectiva linha e coluna.
- n . O tamanho da cadeia.
- *Tipo*. O Tipo da cadeia, que admite valores zero, um ou dois, informando se é um “string”, inteiro ou real. Estes tipos impõem algumas restrições sobre o que pode ser aceito, mas todos eles são tratados pelo procedimento como uma cadeia de caracteres.
- *Atr*. Um atributo informando se a cadeia admite ou não uma cadeia vazia. Na prática, especifica se é possível, durante a entrada de dados de uma tela, saltar a digitação do campo, simplesmente pressionando a tecla “Enter”.
- *Resposta*. O índice i , informando que o campo sendo editado é C_i , dentro da sequência $C_1, C_2, \dots, C_n$, em que os campos variáveis editáveis foram declarados.
- *Num_Campos*. O número total de campos.
- *TrKey*. Recebe através deste parâmetro, na entrada de um campo C_j , um valor nulo, indicando que deve ser feita a edição normal da cadeia. Pode receber um valor não nulo, correspondendo a uma tecla de transmissão de tela, indicando que na edição de um campo $C_i, i < j$, foi pressionada a tecla de transmissão de tela presente neste parâmetro. Nesta situação, a tarefa do procedimento se restringe à verificação da tentativa de transmissão do campo C_j com uma cadeia vazia quando, na sua declaração, isto foi proibido. Se tal fato ocorrer, a transmissão é cancelada e exibida uma mensagem de erro.

A variável global de nome `Vector_Screen_` declarada no mesmo módulo que contém o procedimento `Le_Strn_` é um vetor que foi rapidamente mencionado em 5.2.3 e é inicializado com zeros.

Vamos chamar de *arquivos de telas* à sequência $T_1, T_2, \dots, T_n$, onde cada elemento é um registro associado à tela de entrada T . Seria desejável que este vetor tivesse os seus valores

atualizados somente antes da edição do primeiro elemento do arquivo de T e depois do último elemento. Mas para atender a generalidade do comando de seleção, isto não é possível, pois podemos ter casos de entrada de dados através de telas T e T' , alternadamente, cada uma delas com conjunto distinto de teclas de transmissão. Assim, a cada iteração do comando de seleção do referido item, é feita a dupla atualização do vetor `Vector_Screen_`.

A execução de `Le_Strn_` é encerrada quando ocorre uma tecla de transmissão de campo, de tela ou de encerramento do arquivo da correspondente tela e devolve:

- *Resposta.* O índice do próximo campo a ser editado. Ele vale zero se efetivamente deve haver a transmissão da tela e -1 se ocorreu a `End_Key`.
- *TrKey.* A tecla de transmissão de tela utilizada. Este valor será atribuído ao campo `Transmit_Key` da tela, se ela foi especificada. Neste caso, se a tela foi transmitida trivialmente, será armazenada a primeira tecla da lista de teclas de transmissão, fornecida na declaração.

As restrições gerais que o procedimento impõe, com relação ao tipo são:

- Se o tipo é inteiro, só admite dígitos de zero a nove e o sinal “-” na primeira posição do campo.
- Se o tipo é real, admite caracteres aceitos para o tipo inteiro e também uma ocorrência do ponto decimal.

Comandos Gerados

O gerador produz os comandos de seleção de uma tela, através de chamadas ao procedimento `Le_Strn_`, que, a esta altura, é uma “caixa preta”. Como comando de inicialização relativa ao item T que é uma tela, é produzido o seguinte:

```
T_t_ := 0;
```

que é simplesmente a inicialização do indicador de fim do arquivo de telas associado a T , com o valor falso. Para os comandos de leitura, que seguem, vamos supor uma tela T armazenada no índice t da tabela de identificadores do gerador, C a condição de seleção, D a condição de acesso e $I_p_$ a variável controladora do loop. Para os comandos de leitura é necessário declarar três temporários I_{q-} , Ch_{r-} e A_{s-} , onde q, r, s são número inteiros incrementados à medida que novos temporários são necessários. I_{q-} é do tipo inteiro e armazena o índice do campo que está sendo atualmente tratado, Ch_{r-} é um caractere que armazena a tecla de transmissão e A_{s-} é um “string” que armazena cadeia que está sendo atualmente editada. A primeira destas variáveis auxiliares tem o seu valor inicial igual a 1, a segunda recebe o caractere nulo ($\#0$). Em lugar do temporário Ch_{r-} poderíamos utilizar o próprio campo `Transmit_Key` da tela, mas isto provocaria a necessidade de um tratamento diferenciado quando uma tela não possui tal campo. Os comandos de leitura também são gerados, conforme o esquema seguinte. Para simplicidade, não vamos listar todos os comandos.

```
1  SP_S_n_ :   (* rótulo do item *)
2    se ( $T.t_ = 0$ ) e ( $D$ ) então
3      Inicializa  $I_{q-}$  e  $Ch_{r-}$ 
```

```

4      Inicializa Vector_Screen_
5      enquanto  $I_{q-} \leq n$  faça
6          caso  $I_{q-}$ 
7              -1: Comandos de encerramento do arquivo de Telas
8              0: Comandos de transmissão de tela
9               $i: (1 \leq i \leq n)$  Comandos de leitura de um campo
10         fim-caso
11     fim-enquanto
12     se  $\mathcal{C}$  then goto SP_S_n_ :
13     se  $T_{t-} = 0$  então  $I_{p-} := 1$ 
14     Acerta os valores de Vector_Screen_
15 fim-se

```

Na maioria das vezes, a variável T_{t-} seria desnecessária, mas a sua presença se presta ao atendimento do comando de seleção genérico, que permite diversos itens. Neste caso, seria possível haver encerrado o arquivo de telas, mas outros registros ainda precisam ser selecionados de outros itens do mesmo comando de seleção.

No esquema, o comando da linha 9 trata da leitura de cada um dos campos editáveis da tela, $C_1, C_2, \dots, C_n$, dependendo do valor de i , através de chamada ao procedimento *Le_Strn_*. Antes da chamada, é preciso passar o conteúdo do campo para a cadeia auxiliar A_s- . Inversamente, no retorno do procedimento, é preciso atualizar o valor do campo com o conteúdo desta variável. Se o tipo do campo é inteiro ou real, estas operações são feitas com o campo auxiliar declarado no registro da tela e, por fim, é chamada uma rotina do sistema *Turbo Pascal* que transforma a cadeia num número a ser armazenado no campo correspondente.

O comando da linha 7 é uma pequena lista de ações que devem ser executadas no fim do arquivo de telas, como atribuir o valor 1 à variável T_{t-} e forçar a saída do loop, fazendo

$$I_{q-} := n + 1.$$

O comando da linha 8 trata de efetivar a transmissão de uma tela. Se houve a declaração da tecla de transmissão, o valor do parâmetro correspondente passado pelo procedimento *Le_Strn_* deve ser transportado para o campo *Transmit_Key* da tela. Se a tela foi transmitida trivialmente, e houve a declaração de teclas de transmissão, então o campo *Transmit_Key* da tela deve armazenar o equivalente à primeira tecla de transmissão, dentro da lista fornecida na declaração da tela T . Feito isto, o programa força a saída do loop fazendo a mesma atribuição do final dos comandos da linha 7. Normalmente poderíamos atualizar a variável controladora do loop neste ponto, visto que um registro da tela foi selecionado com a transmissão da tela, mas teríamos problemas se ele não satisfizer à restrição especificada na condição de seleção. A transmissão trivial ocorre somente no último campo da tela. Se for pressionada uma tecla de transmissão na entrada do campo $C_i, i < n$, o procedimento incrementa o índice i e devolve a tecla de transmissão num dos parâmetros. Na entrada do campo C_{i+1} , em razão deste valor, o procedimento *Le_Strn_* não fará a entrada dos dados, mas verificará a compatibilidade do campo com os seus atributos.

O comando da linha 12 testa a validade da condição de seleção sobre a tela recentemente digitada. O comando da linha 13 verifica se a tela foi transmitida ou se o arquivo correspondente

de telas foi encerrado. No primeiro caso, atualiza a variável controladora do loop. O comando da linha 14 acerta o conteúdo do vetor `Vector_Screen..`

Na maioria dos casos, os comandos de seleção especificados no arquivo de entrada contêm somente um item e alguns comandos que são produzidos seriam dispensáveis, mas eles são necessários para o caso geral.

O comando da linha 6 (**case**) faz com que se escolha corretamente o campo a ser editado e o da linha 5 (**while**) permite ao operador “passear” pelos diversos campos da tela, antes de sua transmissão.

Nos casos em que são especificadas todas as teclas como sendo de transmissão *all* ou quando são especificadas algumas teclas de transmissão e não existem campos variáveis passíveis de edição, o gerador produz código diferente e mais simplificado sem a necessidade de chamadas à rotina `Le_Strn..` No primeiro caso, basta ler o carácter pressionado no teclado e atribuí-lo ao campo `Transmit_Key` da tela. No segundo caso, é preciso gerar um “loop” que permaneça lendo os caracteres do teclado até encontrar um dos especificados como tecla de transmissão ou de encerramento. No momento da geração do código, os atributos das telas estão disponíveis na *TIG* para serem consultados.

5.3.10 Código para seleção de “strings”

Qualquer referência ao nome de um arquivo, tabela ou sequência significa automaticamente uma referência ao elemento corrente. No caso de cadeias de caracteres (“strings”) existe a possibilidade de acessar a cadeia como um todo ou cada um de seus elementos. Um comando de seleção sobre uma variável ou campo do tipo “string” seleciona um de seus caracteres de forma análoga à seleção de uma tabela. Para isto o gerador precisa produzir a declaração de uma variável apontadora para o elemento corrente da cadeia. Como nem todas as cadeias precisam ter os seus elementos acessados individualmente, o gerador deve cuidar para que sejam declarados apontadores somente para cadeias que deles necessitem. Para isto a *TIG* deve manter uma informação extraída da análise das especificações para indicar a necessidade de tal declaração. O término da seleção de uma cadeia de caracteres ocorre de forma análoga ao de uma tabela, mas existe a possibilidade adicional de encerramento antes do valor máximo estipulado na declaração ou na cláusula `starting_at ...to ...`, através do esgotamento dos seus elementos úteis. Uma variável ou campo do tipo “string” admite a seleção indexada como descrita na seleção de uma tabela.

5.3.11 Agrupamentos

Vamos considerar o seguinte comando de seleção:

```
%select ...
    grouping by  $A_1$  exec subprocess  $SP_1$ 
                $A_2$  exec subprocess  $SP_2$ 
               :
                $A_p$  exec subprocess  $SP_p$ 
```


Como vimos em 5.3, $A_1, A_2, \dots, A_p$ são os campos de agrupamentos e $SP_1, SP_2, \dots, SP_p$, os sub-processos de agrupamento. Cada campo de agrupamento é o nome de um campo elementar de um dos itens do comando de seleção e deve ser totalmente qualificado, isto é, $A_i = E_1 \cdot E_2 \cdots E_l$, onde E_l é um campo elementar. Um prefixo da qualificação pode ser desprezado se não houver ambigüidade do sufixo correspondente em outra declaração.

Vamos chamar de uma *cláusula de agrupamento* G_i à cadeia seguinte de um comando de seleção.

A_i exec subprocess SP_i

Vamos estudar a cláusula G_1 . Ela tem a finalidade de agrupar os elementos do conjunto abstrato, previamente ordenado em A_1 , em seqüências de elementos que têm valores iguais no campo A_1 . As cláusulas de agrupamento só podem aparecer nos comandos da etapa principal e o sub-processo de agrupamento possui somente as etapas de inicialização ou de finalização (ou ambas), mas não a principal. Deste modo, é um sub-processo anômalo que é executado uma vez para toda a seqüência de registros com o mesmo valor no campo A_1 .

Sejam $R = \{R_1, R_2, \dots, R_l\}$ o conjunto dos registros selecionáveis por um comando de seleção e $S = \{R_i, R_{i+1}, \dots, R_k\}$ um sub-conjunto de R , tal que

- Os registros $R_i, R_{i+1}, \dots, R_k$ contém o mesmo valor no campo A_1 .
- Os registros R_{i-1} e R_i contém valores diferentes no campo A_1 , se $i > 1$.
- Os registros R_k e R_{k+1} contém valores diferentes no campo A_1 , se $k < l$.

Para todo sub-conjunto $S = \{R_i, R_{i+1}, \dots, R_k\}$ de R com as propriedades acima, antes do processamento de R_i deve-se executar uma única vez a parte de inicialização do sub-processo SP_1 ; Após processar-se R_k , deve-se executar uma única vez a parte de finalização de SP_1 . Pelo exposto, entre o processamento dos registros R_k e R_{k+1} são executadas as duas etapas do sub-processo SP_1 , mas a de finalização deve anteceder à de inicialização. Precisamos saber durante a execução se se trata do primeiro registro ou se o valor do novo campo é diferente do anterior. Em caso afirmativo, deve ser executada a etapa de inicialização. Já a etapa de finalização deve ser executada quando houver alteração do valor no campo de agrupamento e quando encerrar os dados de entrada.

Se temos várias cláusulas de agrupamento sucessivas num mesmo comando de seleção significa que dentro de um agrupamento, queremos efetuar um sub-agrupamento. Assim, em lugar de agruparmos em sub-conjuntos do conjunto de registros selecionáveis pelo comando de seleção, queremos agrupar em sub-conjuntos dos sub-conjuntos S , definidos pela cláusula de agrupamento anterior. Num exemplo prático, desejamos agrupar os funcionários de uma empresa por Departamento e dentro de cada departamento, agrupar por Setor. Seguindo o exemplo genérico, a cláusula de agrupamento G_2 agruparia através do campo A_2 , os registros $R_i, R_{i+1}, \dots, R_k$ e não mais $R_1, R_2, \dots, R_l$, a menos que $i = 1$ e $k = l$. Neste caso, o sub-processo de agrupamento a ser executado seria SP_2 .

Vamos supor um comando de seleção com uma seqüência de cláusulas de agrupamento $G_1, G_2, \dots, G_p$. Quando um evento produz a execução da etapa E (finalização) de um sub-processo de agrupamento SP_i , associado à cláusula G_i , $i < p$, antes dela devem ser executadas as

etapas de finalização de $SP_p, SP_{p-1}, \dots, SP_{i+1}$, nesta ordem. Analogamente, se um evento produz a execução da etapa de inicialização de um sub-processo SP_i , associado à cláusula de agrupamento G_i , depois dela devem ser executadas as etapas de inicialização de $SP_{i+1}, SP_{i+2}, \dots, SP_p$, nesta ordem.

Os eventos que produzem a execução das partes de inicialização e finalização de um sub-processo SP_i associado à cláusula de agrupamento G_i , $i > 1$, são os mesmos para G_1 , exceto que em lugar de considerar o conjunto de registros selecionáveis pelo comando de seleção, considera-se o conjunto dos agrupados pela cláusula G_{i-1} .

Um evento singular que ocorre no tratamento de agrupamentos é que, encerrando o loop de seleção, devem ser executadas as etapas de finalização de todos os sub-processos de agrupamento do comando de seleção, associado ao loop que está sendo encerrado, exceto se nenhum registro foi selecionado, o que pode ser verificado pelo valor da variável C_Pn , onde C é o campo de agrupamento da primeira cláusula, n o índice da tabela de identificadores do gerador onde C está armazenado e P a própria letra "P". O código para isto é produzido no comando de controle do loop. O mesmo ocorre se o loop for encerrado pelo esgotamento do número de iterações. Neste caso, o encerramento do loop é produzido pelo comando de verificação da variável contadora de iterações, na base do loop. Normalmente, neste ponto, o gerador insere um desvio para a etapa E . Excepcionalmente, neste caso, antes de executar este desvio, devem ser executadas as etapas de finalização de todos os sub-processos de agrupamento relativos ao comando de seleção corrente. Apesar desta necessidade, quando o gerador está analisando a expressão que fornece o limite de iterações, ele não tem conhecimento se existirão cláusulas de agrupamento e, muito menos, quais ocorrerão. Para superar este problema e considerando que um comando de seleção é traduzido em duas áreas de memória temporárias e separadas, para juntá-las no seu final, o gerador mantém um apontador para este comando de desvio, que na maioria dos casos é conservado. No final do comando, havendo a cláusula de agrupamento e tendo ocorrido a especificação do número máximo de iterações, é produzido um rótulo no comando de controle de loop e o comando apontado é substituído por um desvio para este rótulo e não imediatamente para a etapa E .

Os sub-processos de agrupamento podem referenciar as funções de agregação na sua etapa de finalização, como visto em 5.1.3. Na etapa de inicialização deste sub-processo são inseridos os comandos que inicializam as variáveis declaradas automaticamente pelo gerador em decorrência das funções de agregação utilizadas. Como mencionamos, as ocorrências destas funções são substituídas por cadeias de símbolos adequados. As variáveis são declaradas conforme explicado em 5.2.5. Assim, as variáveis associadas às funções de agregação **Count** e **Total** são inicializadas com zero e as associadas às funções **Max** e **Min** com os valores mínimos e máximos comportados pelos respectivos tipos. A função **Average** utiliza as variáveis associadas com as funções **Total** e **Count**. A inicialização das variáveis são feitas na etapa I , mas as funções somente aparecem na etapa E , ou seja, o gerador deduz quais variáveis deve inicializar na etapa I , quando estiver analisando a etapa E . Para contornar este pequeno problema, na análise da etapa I o gerador coloca uma "marca", para que no segundo passo ele inclua os comandos necessários. Além disso, estas mesmas variáveis precisam ser atualizadas a cada iteração. Isto é implementado através de comandos inseridos no programa objeto imediatamente após os comandos de seleção. A variável relativa à função **Count** é incrementada unitariamente; a relativa à função **Total(C)** tem o seu valor incrementado de C . Já as funções **Max(C)** e **Min(C)** tem o seu valor comparado com

C e substituído por este, se for o caso.

Como a ordem dos sub-processos numa especificação não obedece a nenhum critério, exceto que o primeiro deles é o processo pelo qual se inicia a execução, é possível e normal que os sub-processos de agrupamento ocorram depois do sub-processo em que são utilizados. Como os comandos de atualização das variáveis de agrupamento são colocados após o comando de seleção no sub-processo chamador, o gerador não pode saber quais variáveis devem ser atualizadas. Portanto, novamente ele coloca uma marca, juntamente como um indicador do sub-processo invocado para que, no segundo passo, ele possa produzir no ponto assinalado os comandos de atualização das variáveis associadas às funções de agregação que ocorrem no sub-processo invocado.

Um aspecto importante com relação às funções de agregação, que tratam de valores de campos, é o instante em que o valor do campo deve ser contabilizado, uma vez que eles podem ter o seu conteúdo alterado no curso da execução da etapa principal do sub-processo. A convenção utilizada é que o valor tomado para ser contabilizado nas funções de agregação é o do momento logo após a seleção, antes da execução das fases C e T . Parece sem utilidade, mas não existe impedimento para que as funções de agregação atuem sobre campos de variáveis que não foram selecionadas. Qualquer utilização deste tipo leva em conta valores dos campos, na saída da iteração anterior; contabiliza o seu valor na entrada da primeira iteração e não contabiliza o seu valor na saída da última iteração.

Considere o seguinte exemplo, com um trecho de comando de seleção que contempla somente a parte das cláusulas de agrupamento.

```
grouping by  $A_1$  exec subprocess  $SP_1$ 
            $A_2$  exec subprocess  $SP_2$ 
           ⋮
            $A_p$  exec subprocess  $SP_p$ 
```

onde A_i identifica um campo e tem a forma $E_{i_1} \cdot E_{i_2} \cdots C_i$, podendo conter apenas o sufixo mínimo que identifica o campo sem ambigüidade. Seja C_i o campo elementar de A_i , $1 \leq i \leq p$. O código produzido para uma cláusula arbitrária G_i é o que está apresentado a seguir.

```
if ( $C_i$ - $Pn_i$ -=0) and ( $C_i$ - $An_i$ - <>  $E_i \cdots C_i$ ) then
  begin
    Executa  $SP_p$ - $E$ -;
    Executa  $SP_{p-1}$ - $E$ -;
    ⋮
    Executa  $SP_i$ - $E$ -;
  end;

if ( $C_i$ - $Pn_i$ -=1) or ( $C_i$ - $An_i$ - <>  $E_i \cdots C_i$ ) then
  begin
    Executa  $SP_i$ - $I$ -;
     $C_i$ - $Pn_i$ - := 0;
     $C_i$ - $An_i$ - :=  $E_i \cdots C_i$ ;
     $C_{i+1}$ - $Pn_{i+1}$ - := 1;
```



```

Ci+2-Pni+2 := 1;
      ⋮
Cp-Pnp := 1;
end;

```

O comando “Executa S ”, no esquema significa um desvio para o rótulo S , pois SP-I- e SP-E- são rótulos que identificam as etapas de inicialização e finalização do sub-processo SP . Além disso, o comando inclui o empilhamento de informação para o correto retorno para o próximo comando. Para isto, o comando acima inclui também um rótulo logo após o citado comando de desvio.

Após a produção destes comandos, para cada uma das cláusulas de agrupamento são enumerados os comandos de atualização das variáveis associadas às funções de agregação que ocorrem nas etapas E dos sub-processos de agrupamento, na forma já mencionada.

Na parte dos comandos de inicialização do comando de seleção, é inserido

```

C1-Pn1 := 1;

```

somente para a primeira cláusula de agrupamento, indicando que é o primeiro registro do agrupamento e que deve ser executada a etapa inicial do sub-processo.

5.3.12 Encaixamento de Comandos de Seleção

Consideremos dois comandos de seleção numa etapa principal, conforme o exemplo seguinte:

```

%select A
%select B

```

Como cada comando de seleção na etapa principal determina um loop, temos um encaixamento de loops. Para cada registro selecionado pelo primeiro comando, vários registros são selecionados no segundo comando. Sejam $A = \{a_1, a_2, \dots, a_{n_A}\}$ e $B = \{b_1, b_2, \dots, b_{n_B}\}$ os conjuntos dos registros selecionados pelos comandos de seleção do exemplo. Assim, o sub-processo tem como conjunto abstrato o produto cartesiano

$$C = \{(a_i, b_j), 1 \leq i \leq n_A, 1 \leq j \leq n_B\}.$$

Como os comandos de seleção comportam cláusulas, o encaixamento deles nem sempre se traduz num produto cartesiano propriamente dito para o conjunto abstrato. Este seria o caso do exemplo acima. Havendo condições de seleção e acesso, obviamente o conjunto abstrato será um sub-conjunto do produto cartesiano. Por exemplo, se no segundo comando de seleção existe uma condição de seleção que depende do particular elemento de A correntemente selecionado, em geral uma equação de junção no sentido do modelo relacional de dados, temos para cada elemento a_i de A um sub-conjunto $B_i \subseteq B$ selecionado pelo segundo comando.

Estes encaixamentos com um número arbitrário de comandos de seleção são implementados tratando cada comando isoladamente, da forma descrita até o momento, a menos dos comandos de controle do loop e de verificação da variável contadora de iterações que são produzidos para cada comando de seleção. Tratando-se do primeiro comando de seleção é feito um desvio para a etapa final E ; nos comandos seguintes o desvio é feito para a base do loop do comando anterior, o que é ilustrado pelo seguinte esquema, para o comando de controle do loop:

```

if In_ = 0 then goto SP_E_      (a)
if In_ = 0 then goto SP_S.t_   (b)

```

O comando (a) é produzido se a especificação de seleção é a primeira da etapa principal do sub-processo *SP*, caso contrário o comando (b) é gerado. O rótulo *SP_E_* assinala o início da etapa *E* do sub-processo *SP* e *SP_S.t_* a base do loop do comando de seleção anterior. De forma análoga, os comandos de verificação da variável contadora de iterações produzem desvios para o rótulo de encerramento do loop que é *SP_E_* ou *SP_S.t_*, conforme o mesmo critério.

5.3.13 Seleção através de Sub-processos

É possível selecionar o resultado da execução de um sub-processo, para incorporá-lo ao elemento do conjunto abstrato selecionado pelos outros comandos de seleção. A implementação deste mecanismo de seleção é um simples desvio para o sub-processo e conseqüente retorno. Este comando de seleção não determina um loop, uma vez que o loop é produzido na execução do sub-processo chamado. A sintaxe desta forma de seleção é a seguinte:

%select $A_1, A_2, \dots, A_n$ thru subprocess *SP* when *D*

Neste caso, a semântica do comando é alterada. A lista de itens é um simples comentário para o entendimento de quais variáveis serão selecionadas através do sub-processo, pois isto é feito, na prática, por meio de variáveis globais. Não é permitida a especificação da expressão contadora de iterações e nem da condição de seleção. A condição de acesso só é permitida para todo o comando de seleção e não para cada um dos itens do comando. O código produzido pelo gerador para um comando de seleção por sub-processo é o seguinte:

```

if (D) then
  begin
    Empilha o índice do rótulo de retorno n
    goto SP_I_;
  label_n_:
  end;

```

onde o comando “Empilha o rótulo de retorno” é simplesmente construir um mecanismo que possibilite retornar para o comando seguinte, após o encerramento do sub-processo chamado.

A seleção de um sub-processo introduz dificuldades na geração do código, uma vez que nos outros comandos de seleção o código é produzido à medida que os itens são detectados pelo gerador, no arquivo de entrada das especificações. Como, neste caso, os itens são simples comentários, estamos num impasse, pois enquanto não termina a lista de itens não sabemos se a seleção será de um sub-processo ou de variáveis. Isto é decidido pela presença ou não da palavra “thru” na cadeia de entrada. Pela sistemática até aqui adotada, teríamos que descartar todo o código gerado e guardado nas duas memórias auxiliares. Outra alternativa seria adiar a produção do código de seleção para o fim da lista de itens, mas isto traria um relativo consumo de memória, pois cada item comporta diversas cláusulas. O gerador segue a primeira solução, mas a segunda tem a vantagem de não produzir código para ser descartado e o consumo de memória não é maior que o necessário para o armazenamento dos comandos de inicialização e leitura, nas

duas áreas auxiliares, até o fim da lista de itens. Mas tem a desvantagem de ter que manter uma lista de itens a serem selecionados e o trabalho adicional de interromper a manutenção desta lista se ocorrer alguma cláusula antes da palavra *thru*, pois ela é incompatível numa seleção por sub-processo ou senão manter organizadamente uma lista de condições de seleção e outra de condições de acesso, se escolher por produzir os comandos somente no final do comando de seleção. No caso de itens, que são tabelas, é necessário armazenar, potencialmente, os limites inferior e superior para seleção, de forma organizada, para não confundir com cláusulas de outros itens.

Uma observação quanto ao uso dessa forma de seleção: O encaixamento de comandos como

```
%select A
%select B where  $A \cdot X = B \cdot X$ 
```

produz a junção de um elemento de *A* com um de *B*, formando um elemento do conjunto abstrato; em seguida produz eventualmente a junção do mesmo elemento de *A* com o próximo selecionado de *B*, formando o próximo elemento do conjunto abstrato, e assim por diante. Já os comandos

```
%select Aluno
%select Media thru subprocess Calcula_Media
```

produz um elemento do conjunto abstrato formado pela junção de um elemento de *Aluno* com o resultado da execução de *Calcula_Media* que, se bem construído, deve colocar a média das notas desse aluno na variável *Media*; para isso esse sub-processo relacionaria as várias notas desse aluno.

Em ambos os exemplos temos um processamento de uma ligação 1 : *n* [Se 87]; no primeiro temos a junção tradicional do modelo relacional; no segundo o que foi denominado por Setzer de “super-junção” em [Se-Ma 88], isto é, a junção de um elemento com o resultado do processamento de *n* outros ligados a ele.

5.4 Tabelas de Decisão na Fase de Escolha

A fase de escolha (*C*) de uma etapa de um sub-processo é especificada através de tabelas de decisão, que foram tratadas nos primeiros capítulos deste trabalho. A tabela é submetida ao gerador da mesma forma como foi descrita para o pré-processador de tabelas de decisão, apresentado no capítulo 4, e o código gerado, em alguns casos, pode ser ligeiramente diferente em razão do gerador usar o método I de Pollack (2.4.7), no sub-algoritmo de escolha da próxima condição no algoritmo de fatorações sucessivas, enquanto o pré-processador usa o método esquerda para direita (2.4.9). As poucas mudanças nas especificações são enumeradas a seguir.

5.4.1 Entradas de Condição

Nas linhas de condição estendidas é permitido especificar uma entrada da forma $\sim C$, onde *C* é o nome de um conjunto definido na parte de definição de conjuntos. Esta entrada tem por objetivo informar que, para a regra em questão, a variável de condição deve pertencer ao complemento do conjunto *C*. É uma notação útil, mas de uso restrito, pois a restrição de exclusão proíbe

outras entradas de condição relevantes, além de C e $\sim C$, nas linhas de condição estendidas em que esta última está presente.

5.4.2 Descrição das Ações

Na parte de descrição das ações, podem estar comandos da linguagem Pascal, mas dentro do método I-M-E, que pretende separar as decisões das ações, isto deve ser evitado. Da mesma forma como no pré-processador, podem estar presentes chamadas para outras tabelas, com ou sem retorno, e chamadas de transformações pelos seus nomes. Além disso, o gerador admite os dois comandos abaixo, na parte de ações de uma tabela.

<code>%exec subprocess Nome_Subprocesso</code>	(a)
<code>%exit</code>	(b)

O primeiro deles (a), é uma chamada ao sub-processo de nome *Nome_Subprocesso*. O segundo (b) é o comando que encerra a execução do sub-processo, significando que deve ocorrer na sua etapa M e o fluxo de execução é desviado para a sua parte final. Este tipo de comando produz inconveniências na implementação. Por enquanto, todas as chamadas de sub-processos, transformações e tabelas exigiam o empilhamento do ponto de retorno, para que ao final da execução do código invocado, o fluxo de execução tivesse o curso desviado para o comando seguinte à chamada. Agora, com o comando `exit`, o desvio para a etapa E , independentemente de terem sido encerradas todas as chamadas empilhadas na etapa M , produz um problema de manutenção da pilha. Este comando traz a necessidade de empilhar mais um elemento além do indicador do ponto de retorno. A solução adotada é empilhar, além do ponto de retorno, um número que identifica o sub-processo chamado. Se o elemento invocado não for um sub-processo, produz o empilhamento de um valor nulo. Assim, ocorrendo o comando `exit`, a pilha deve ser deixada no estado em que estava no ponto em que o sub-processo foi chamado.

Como o comando `exit` produz o abandono do sub-processo corrente, através da execução imediata da sua parte de finalização, é claro que no momento que tal comando ocorre, não existe nenhuma chamada a sub-processo pendente na pilha, desde o seu topo até o índice i da pilha, que armazena o sub-processo corrente. Esta propriedade garante que o comando de abandono deve produzir o desempilhamento, a partir do topo, de todos os elementos até o último sub-processo empilhado. A partir desta constatação, é possível implementar a pilha, sem o empilhamento de um número associado ao sub-processo. Basta empilhar um número relacionado com o ponto de retorno, como feito normalmente, para os elementos que não são sub-processos e, para estes, deve ser empilhado o mesmo número, mas negativo. Assim, o acerto da pilha, provocado pelo comando `exit`, pode ser feito, buscando o primeiro elemento negativo, a partir do topo e o desempilhamento do ponto de retorno deve considerar os valores absolutos do que está empilhado.

5.4.3 Ações Comuns de Preparação

Na parte de ações de preparação, podem ser especificadas chamadas a sub-processos, transformações e tabelas através do comando `%exec table`. Todas as chamadas deste tipo são implementadas através de uma chamada a rotina de empilhamento do ponto de retorno, seguido

do comando de desvio e do rótulo para a ação seguinte. Além disso, são admitidos comandos na própria linguagem Pascal, mas aplica-se a eles a observação feita na parte de ações de uma tabela de decisão. Para cada comando de chamada é produzido o seguinte código:

```
Empilha (n,m)
goto Nome_Chamado;
label_n_:
```

onde n é um número indicador de que o ponto de retorno é um rótulo de nome `label_n_` e m o número associado ao sub-processo chamado ou um valor nulo, se o elemento chamado não for um sub-processo.

5.5 Comandos de Transformação

A fase de transformações (T) de uma etapa dentro do método I-M-E é especificada através de comandos de transformação.

Se existe a fase C numa etapa, então as transformações são referenciadas pelos seus nomes. Assim, a presente fase é uma sequência de definição das transformações referenciadas na fase de escolha. Isto é feito de forma análoga à utilizada pelo pré-processador de tabelas de decisão. Cada uma das transformações $T_1, T_2, \dots, T_n$ referenciadas é definida nesta fase, através de um cabeçalho, seguido dos comandos de transformação e de seu encerramento. Assim, uma transformação T_i é definida pelas seguintes especificações ao gerador, dentro da fase T .

```
%Transformations
:
%transf  $T_i$ 
:
Comandos de transformação
:
%end  $T_i$ 
:
```

Se a etapa não contém a fase de escolha, então as especificações da fase T são iniciadas logo após o comando `%Transformations` que especifica esta fase, com a enumeração dos comandos de transformação, como mostra o seguinte esquema.

```
%Transformations
:
Comandos de transformação
:
```


A fase de transformações termina com a especificação da última transformação referenciada nas tabelas de decisão, se existe a fase *C*. Caso contrário, ela termina quando esgotar os comandos de transformação. Nos dois casos, o gerador detecta este fato, porque estando na etapa *I* ou *M*, começa a etapa seguinte, com os símbolos %M e %E, respectivamente. Se já estamos na etapa final ou estamos na etapa *M* e não existe a etapa final, o fim das especificações da fase de transformações é detectada com o encerramento do sub-processo, que é passado ao gerador numa das seguintes formas:

%end process	(a)
%end subprocess	(b)

A primeira delas (a) é colocada no término de um processo e a segunda (b) no fim de um sub-processo. Os processos e sub-processos não têm diferenças estruturais; a distinção se deve ao problema. O processo trata do conjunto abstrato principal para a solução do problema, enquanto os sub-processos tratam de outros conjuntos por invocação do processo ou sub-processos. Em razão disto, o processo é o primeiro a ser especificado no arquivo de entrada e é único dentro dele. Os sub-processos aparecem em seguida, em ordem arbitrária.

Agora, vamos descrever cada um dos comandos de transformação que podem ser especificados ao gerador.

5.5.1 Comando de Inicialização

Este comando serve para inicializar tabelas inteiras ou variáveis simples com valores constantes. Por exemplo, considere uma tabela *T* com campos elementares *Valor* e *Nome*, que devem ser inicializados com 0 e *vazio*, respectivamente. O comando seria especificado da seguinte forma:

```
%initialize T (Valor:0;Nome:' ')
```

O código correspondente em Pascal, produzido pelo gerador é:

```
for Ij- := i1 to i2 do
  begin
    T.Valor := 0;
    T.Nome  := ' ';
  end;
```

onde *I_j-* é um temporário declarado automaticamente pelo gerador e *i₁* e *i₂* são os índices mínimo e máximo que constam da declaração de *T* e *Valor* e *Nome* devem ser campos elementares. No caso do exemplo, se *T* é uma tabela com uma coluna de nome *T₁*, que por sua vez tem um campo *T₂* e, assim sucessivamente, até que o campo *T_n* tem *Valor* como um de seus campos elementares, então seria produzido o comando

```
T.T1. T2... Tn. Valor := 0;
```

Para efetivar esta possibilidade, o gerador faz uma busca em “pré-ordem” na árvore que define o tipo dos elementos de *T*.

5.5.2 Comando de Troca

Este comando especifica dois operandos que devem ter os seus valores passados de um para o outro.

`%exchange  $V_1$  with  $V_2$`

onde V_1 e V_2 podem ser variáveis simples ou campos elementares de conjuntos estruturados e, neste caso, devem estar devidamente qualificadas, a menos que não exista ambigüidade. O gerador produz um temporário do mesmo tipo dos operandos para efetuar a operação, com os seguintes comandos.

```
Temp :=  $V_1$ ;
 $V_1$  :=  $V_2$ ;
 $V_2$  := Temp;
```

Por exemplo, para trocar os valores do campo *Valor* de um arquivo *A* com o campo *Valor* de uma tabela *T*, escreveríamos:

`%exchange A.Valor with T.Valor`

e o gerador produziria

```
Temp := A.Valor;
A.Valor := T.Valor;
T.Valor := Temp;
```

A variável *Temp*, se do tipo inteiro, é formado pela letra *I* seguida de um número inteiro positivo que é um contador de temporários do respectivo tipo e terminado pelo caractere “_”. Se for do tipo real, “string” ou caractere, muda somente as letras iniciais para *R*, *A* e *Ch*, respectivamente e é mantido o mesmo método de geração de temporários.

5.5.3 Comando de Atribuição

O comando que efetiva a atribuição é `%change`. Ele atua sobre variáveis simples, campos elementares de conjuntos estruturados e também sobre estas estruturas como um todo. Se ele atua sobre estruturas, como um todo, o gerador produz os comandos de atribuição, campo por campo. Se a variável alterada é um arquivo ou um de seus campos, ele produz a alteração do respectivo registro dentro do arquivo. Se é uma tabela ou um de seus campos, altera a linha corrente da tabela. No caso de um arquivo com vários campos, se alterarmos um campo de cada vez, o gerador produziria um comando de alteração do registro para cada uma destas alterações, o que seria altamente ineficiente. Tentando contornar este problema, para conjuntos estruturados é permitido, num único comando, enumerar todas as atribuições que se deseja fazer sobre os seus campos. Assim, o comando pode atuar sobre uma única variável ou um único conjunto estruturado, mas no último caso, pode atuar sobre um número qualquer de seus campos. A forma geral do comando é a seguinte:

`%change X_1 with Y_1 ,
 X_2 with Y_2 ,`

$$\vdots$$

$$X_n \text{ with } Y_n$$

onde X_i e Y_i são, respectivamente, cadeias da forma

$$E_1 \cdot E_2 \cdot E_3 \cdots E_p \quad \text{e}$$

$$D_1 \cdot D_2 \cdot D_3 \cdots D_q.$$

Vamos chamar cada termo

$$X_i \text{ with } Y_i$$

de *item de atribuição*. Assim, um comando *change* pode ter um ou mais itens de atribuição.

É claro que se E_1 é uma variável simples, $p = 1$. Analogamente, se D_1 é uma variável simples, $q = 1$. Se E_1 é um campo de arquivo ou tabela ou tela, não deve haver ambigüidades, com relação a qual estrutura ele pertence. Os lados esquerdos dos itens de atribuição $X_1, X_2, \dots, X_n$ identificam campos de uma única variável, sendo que o primeiro item de atribuição determina qual variável está sendo modificada. Assim, para X_i , $i > 1$, basta especificar o sufixo que identifica unicamente o campo dentro da estrutura que está sendo modificada.

Toda cadeia X_i deve conter símbolos declarados ao gerador, pois os seus comandos somente podem atuar sobre elementos conhecidos por ele. Já as cadeias Y_i preferencialmente devem conter elementos declarados, mas como o gerador abre a perspectiva de colocação de comandos na própria linguagem Pascal, ele não impede que variáveis desconhecidas estejam presentes em Y_i . Isto introduz uma dificuldade na verificação de tipos. Para superar isto, o gerador supõe, neste caso, que X_i é um campo elementar ou variável simples e faz a atribuição, entendendo que o lado direito é compatível.

Se E_p é um campo elementar ou variável simples, então D_q também deve ser um campo elementar ou variável simples. Por outro lado, se E_p é uma estrutura do tipo Arquivo ou Tabela, ou um campo composto, D_q também deve ser um objeto de um desses tipos. Neste caso, o gerador, durante a tradução, produzirá uma sequência de atribuições, de modo que os campos elementares a partir de E_p recebem os valores de campos elementares a partir de D_q , desde que estes campos tenham os mesmos nomes e tipos. Esta propriedade permite que várias atribuições sejam produzidas a partir de um único comando *change*. Suponha um arquivo A com campos $A_1, A_2, \dots, A_n$, todos elementares e uma tabela T com campos $T_1, T_2, \dots, T_m$, também elementares. O comando

%change A with T

produz os seguintes comandos no programa objeto:

$$A \cdot A_{i_1} := T \cdot T_{j_1};$$

$$A \cdot A_{i_2} := T \cdot T_{j_2};$$

$$\vdots$$

$$A \cdot A_{i_p} := T \cdot T_{j_p};$$

onde os campos A_{i_k} e T_{i_k} têm os mesmos nomes e tipos.

Para implementar este comando, considerando que arquivos e tabelas podem ter campos que são compostos de outros campos, o gerador faz uma dupla busca em árvores. Sejam E e

D duas estruturas de dados que não são variáveis simples e A_E e A_D as árvores relativas aos tipos dos elementos de E e D , respectivamente. O algoritmo que implementa este comando faz o seguinte:

1. Percorre A_E em “pre-ordem”, encontrando um campo elementar E_i .
2. Para cada elemento E_i , percorre A_D em “pre-ordem”, encontrando um campo elementar D_j . Se os nomes e tipos dos campos E_i e D_j são iguais, o gerador produz a atribuição de D_j para E_i .

Se para o campo E_i existem D_j e D_k que satisfazem os requisitos para a atribuição, então é produzido somente para o primeiro que for encontrado na busca da árvore A_D . Isto é válido para um ítem de atribuição. Esta verificação não é feita para distintos ítems de atribuição. Assim, é possível que num mesmo comando de atribuição, um certo campo receba valores de outros dois campos ou variáveis, através de ítems diferentes.

Se E é uma tela, além das atribuições é produzida uma chamada ao procedimento que a desenha no vídeo. Se E é um arquivo, é produzido o comando que regrava o registro modificado. É claro que deve ser um arquivo estruturado e deve ter sido selecionado através de um comando de seleção.

O comando **change** pode também forçar a mudança do elemento corrente de uma tabela, com a seguinte sintaxe.

%change current of T with s

onde T é o nome da tabela e s uma expressão do tipo inteiro.

O comando **insert** é totalmente análogo ao **change** exceto que somente se aplica para inserir um novo elemento no final de um arquivo. O fato destes comandos permitirem uma atribuição múltipla, onde a primeira delas determina a variável do lado esquerdo da expressão, que está sendo tratada, traz uma similaridade com o “with” do Pascal. Esta propriedade é particularmente útil no comando **insert**, pois se a alteração de um campo produz a inserção automática do registro no arquivo e tivéssemos que fazer uma atribuição para cada campo, introduziríamos diversos registros, quando a intenção seria inserir apenas um.

A título de ilustração, segue um exemplo de comando **change** e seu respectivo código, extraído de um exemplo de programa produzido pelo gerador.

```
(*2          %change Notas with Tela *)
NOTAS_Buf_.NUMERO:=TELA_Buf_.NUMERO;
NOTAS_Buf_.NOME:=TELA_Buf_.NOME;
NOTAS_Buf_.NOTA1:=TELA_Buf_.NOTA1;
NOTAS_Buf_.NOTA2:=TELA_Buf_.NOTA2;
NOTAS_Buf_.NOTA3:=TELA_Buf_.NOTA3;
NOTAS_Buf_.NOTA4:=TELA_Buf_.NOTA4;
seek(NOTAS,filepos(NOTAS)-1);
write(NOTAS,NOTAS_Buf_);
```

No exemplo, *Notas* e *Tela* são respectivamente um arquivo estruturado e uma tela, e as atribuições foram produzidas somente para os campos que têm mesmo nome e tipo. *Notas* deve ter sido especificado num comando de seleção. Assim, comando **change** produz automaticamente uma alteração do registro no arquivo.

No exemplo seguinte, temos a oportunidade de observar que os vários itens de atribuição dos comandos **change** e **insert** permitem que a variável *F* receba valores de campos compatíveis de três variáveis distintas, *R*, *S* e *T*. Os campos *C*₁ e *C*₂ foram atualizados por campos de *R*; *C*₃ e *C*₄, por campos de *S*; e *C*₅ e *C*₆, por campos de *T*;

```
(*2      %insert F with R,
          F with S,
          F with T *)
F_Buf_.C1:=R_Buf_.C1;
F_Buf_.C2:=R_Buf_.C2;
F_Buf_.C3:=S_Buf_.C3;
F_Buf_.C4:=S_Buf_.C4;
F_Buf_.C5:=T_Buf_.C5;
F_Buf_.C6:=T_Buf_.C6;
write(F,F_Buf_);
```

No exemplo seguinte, extraído de um trecho de programa obtido pelo gerador, podemos observar a identificação que o primeiro item de atribuição dá a todos os demais itens.

```
(*2      %insert Saida.Nome with 'MINIMA QUANTIDADE',
          Depto with '',
          Setor with '',
          Valor with 0.0,
          Quant with %Min(Entrada.Quant) *)
SAIDA_Buf_.Nome:='MINIMA QUANTIDADE';
SAIDA_Buf_.DEPTO:='';
SAIDA_Buf_.SETOR:='';
SAIDA_Buf_.VALOR:=0.0;
SAIDA_Buf_.QUANT:=Min_SP1_QUANT10_;
writeln(SAIDA,SAIDA_Buf_.NOME,' ':25-ord(SAIDA_Buf_.NOME[0]),' ',
SAIDA_Buf_.DEPTO,' ':10-ord(SAIDA_Buf_.DEPTO[0]),' ',
SAIDA_Buf_.SETOR,' ':10-ord(SAIDA_Buf_.SETOR[0]),' ',
SAIDA_Buf_.QUANT:3,' ',
SAIDA_Buf_.VALOR:5:2);
```

Neste exemplo, o primeiro item de atribuição indica que a variável a ser inserida é um arquivo de nome *Saida* e os demais itens dispensam a qualificação dos campos desta variável. Este exemplo ilustra o emprego de um arquivo do tipo “texto” que está sendo gravado.

Os comandos **change** e **insert** quando atuam sobre cadeias de caracteres (“strings”) podem funcionar como uma função de cópia parcial, através da especificação de uma sub-cadeia da forma seguinte:

```
%change S with W starting_at 10 to 20
```

Neste exemplo os identificadores *S* e *W* devem ser declarados como cadeias de caracteres e o comando **%change** atribui a *S* a sub-cadeia que começa na décima posição e termina na vigésima posição de *W*.

5.5.4 Outros Comandos

Comando de Incremento e Decremento

O comando de incremento é especificado por

`%inc  $V, s$`

onde s é uma expressão opcional inteira ou real, conforme o tipo de V . Estando ausente, a expressão s tem o valor unitário. O código produzido é o trivial $V := V + s$. De forma análoga, é definido o comando de decremento, `dec`.

Comando Clear

Este comando é utilizado quando se deseja limpar o conteúdo de uma tela T , com a sintaxe

`%clear  $T$`

que faz com que o gerador produza o comando de limpeza do vídeo e a inicialização dos campos variáveis de T .

Comando Exit

O comando `exit` é escrito da forma

`%exit`

e, como um comando de transformação, tem o efeito idêntico ao produzido pela sua utilização na parte de ações de uma tabela de decisão e só pode aparecer na etapa principal.

Comando Next

Este comando pode ser usado para uma tabela T , com a seguinte sintaxe

`%next  $T$`

e tem o efeito de tomar a próxima linha da tabela. Tem o efeito análogo ao comando

`%change Current of  $T$  with  $T \cdot \text{Current} + 1$`

Comando de Chamada de sub-processo

As chamadas de sub-processos podem ser utilizadas como comandos de transformação, como ocorre na parte de ações de uma tabela de decisão, com a sintaxe

`%exec subprocess  $SP$`

onde SP é o nome do sub-processo invocado. O código produzido é também análogo ao gerado para estas chamadas na parte de ações de uma tabela de decisão, com o empilhamento de um indicador do ponto de retorno, desvio para o sub-processo e um rótulo para o comando seguinte.

Comando Remove

Existem situações em que a leitura de linhas de arquivos do tipo texto em cadeias de caracteres em Pascal produzem “brancos” à esquerda e/ou à direita da cadeia que interessa ao usuário. O comando **remove** resolve este problema. Por exemplo, o comando

```
%remove left right blanks from S
```

faz com que os caracteres “brancos” nas posições iniciais (**left**) e finais (**right**) de *S* sejam removidos. Este comando é implementado através de uma chamada a uma função compilada em separado.

5.5.5 Expressões

Em diversas especificações ao gerador, são admitidas expressões aritméticas. Elas são sintaticamente verificadas, mas a sua verificação de tipo tem alguns problemas, como mencionamos em alguns pontos. Como o gerador não tem a relação completa das funções utilizadas pelos sistemas em que os usuários trabalham, torna-se difícil uma verificação completa, quando elas estão presentes nas expressões. Além disso, existe a possibilidade do usuário inserir código diretamente na linguagem fonte o que permitiria a inclusão nas expressões de variáveis não declaradas ao gerador. Isto torna ainda mais difícil uma verificação de tipos de uma expressão. Em razão disto, o gerador faz a verificação de tipo somente em expressões nas quais estão presentes, unicamente, variáveis declaradas ao gerador. Fora disto, a verificação é incompleta.

Como dissemos em 5.3.10, os comandos do gerador podem acessar as variáveis e campos do tipo “string” como um todo ou a cada um de seus caracteres. O gerador distingue as duas formas através do campo fictício *char* que é automaticamente atribuído a estas variáveis ou campos. Assim, para uma variável ou campo *S* desse tipo, uma referência ao próprio nome significa o acesso à cadeia como um todo. Desejando acessar o elemento corrente da cadeia basta escrever *S.char*. Esta notação é consistente com a idéia de que *S* pode ser considerado como uma tabela em que cada uma de suas linhas contém um único campo de nome *char*.

5.6 Tabela de Identificadores do Gerador

Se *T* é um arquivo ou tabela e *C* um de seus campos elementar ou não, *C* pode ser acessado, nas especificações ao gerador pela sequência:

$$T \dots C$$

ou simplesmente pelo símbolo *C*, caso não exista ambigüidades. Considerando este fato, resolvemos manter *T* e *C* numa mesma tabela, pois caso contrário, teríamos que fazer uma busca numa tabela de variáveis e, não a encontrando, buscá-la numa tabela de campos. Os nomes de tabelas, processos e transformações correspondem a rótulos no programa gerado e, conseqüentemente, não podem ser usados como nomes de variáveis em Pascal. Portanto deixamos na *TIG* todos os identificadores, isto é, nomes de variáveis, campos e também de sub-processos, tabelas de decisão e transformações. Somente para estes elementos, relacionados com o método I-M-E, é mantida uma tabela, além da *TIG*, cujas entradas são apontadas por um campo da tabela de identificadores. Inversamente, a tabela à parte também, possui um campo que aponta para a

correspondente entrada da *TIG*. Para esses elementos ligados ao método, o gerador precisa armazenar propriedades específicas e para isto são mantidos numa tabela separada, mas poderiam ser armazenados conjuntamente na tabela de identificadores.

Para a manutenção de informações sobre campos e sub-campos de tabelas, arquivos e telas, a *TIG* possui colunas de nomes *Prim_Cmp_Var* e *Prox_Cmp_Var* que apontam respectivamente para o primeiro campo de um elemento e para o próximo campo, se ele faz parte de uma lista de campos. Com estas duas colunas, é possível manter qualquer estrutura de campos e sub-campos, com profundidade arbitrária. Em termos de árvore de definição de um registro, uma das colunas aponta para o primeiro de seus filhos e a outra, para o próximo irmão. Para armazenar informações sobre sinônimos, a tabela possui a coluna de nome *Prox_Sin* que mantém uma lista ligada de sinônimos. Para uma sequência de sinônimos de *T*, o gerador mantém as informações sobre a estrutura de *T* somente na entrada do primeiro da lista. Quando houver referência a um de seus sinônimos, as informações são buscadas no primeiro elemento, que do ponto de vista do gerador, é chamado de *principal*, embora para o usuário todos eles tenham as mesmas propriedades, independentemente da ordem da declaração. Assim, várias colunas da tabela de identificadores do gerador ficam inutilizadas para as linhas que contém sinônimos, visto que as suas propriedades estão resumidas no elemento principal. Usamos uma delas para apontar para o índice em que está o primeiro da lista, isto é, o elemento principal da lista de sinônimos. No nosso caso, apropriamo-nos da coluna *Tam* para esta finalidade. Normalmente, ele armazena o número total de campos que o elemento possui; no caso de telas é o número de campos selecionáveis; para elementos do tipo "string", o número de caracteres que os compõem; para as sequências declaradas com enumeração, o número de seus elementos; se a sequência é declarada por um intervalo, este campo vale zero e esta é a diferença que revela por qual das duas formas uma sequência foi declarada. Quando o gerador está produzindo código para o comando de seleção de uma tela, ele consulta este campo da *TIG* para decidir qual tipo de código deve gerar. Outra utilização de campo com finalidade distinta da inicialmente prevista para a *TIG* ocorre para campos e variáveis do tipo "string". Neste caso, o campo *Prim_Cmp_Var* é inútil para assinalar campos variáveis, pois o identificador armazenado é um tipo elementar. Assim, ele assinala se a variável ou campo necessita ou não de uma variável apontadora para controle do elemento corrente. Se após a análise das especificações, o campo contiver o valor zero, deve ser declarada uma variável auxiliar para apontar o elemento corrente da cadeia. Se contiver o valor -1, a variável ou campo só é referenciado como um todo e não há necessidade de produzir uma variável auxiliar.

As cadeias que constituem os identificadores não são colocadas diretamente na *TIG* mas, em seu lugar, um apontador para uma região da memória que as contém. Esta forma de armazenamento permite alocar o espaço estritamente necessário para os símbolos, não trazendo problemas de sub-utilização de espaço, quando os identificadores possuem poucos caracteres, numa tabela que precisa prever símbolos potencialmente extensos.

A busca por um elemento na *TIG* é feita sequencialmente, a partir dos índices mais altos se o elemento não é um campo de uma determinada estrutura; senão a busca é feita na árvore definida para os elementos desta estrutura.

A *TIG* ainda contém uma coluna de nome *Classe* que classifica os seus elementos em arquivos, tabelas, sequências, constantes, campos elementares, campos compostos (registros que podem ser tabelas), e elementos relacionados com o método, como nomes de tabelas de decisão,

transformações, processos e sub-processos. Cada uma dessas classes admite uma sub-divisão que é mantida numa coluna de nome *Tipo*.

Uma célula da *TIG* tem, além dos campos acima citados, um campo de nome *Pai*, utilizado no caso do elemento ser campo e o seu conteúdo indica o elemento de quem ele é um dos campos. Outras duas colunas *Prim_Cmp_Ctn* e *Prox_Cmp_Ctn* têm funções análogas às usadas para a manutenção da lista de campos e sub-campos, com a diferença de que são utilizadas para os campos constantes das telas.

Para que o gerador possa funcionar corretamente, ainda são necessárias outras informações, que são dependentes dos elementos que estão armazenados. No caso de um arquivo, é necessário, o índice de entrada da *TIG*, onde está o seu nome externo; no caso de uma tela, um apontador para a lista de teclas de transmissão e o valor da tecla de encerramento associada a ela. Excepcionalmente, para a especificação *all* o primeiro elemento da lista de teclas de transmissão contém o valor zero para indicar este fato. Para tabelas, são necessários os valores inferior e superior dos seus índices, que servem também para o caso de sequências declaradas através de intervalos. Para um campo de arquivo do tipo texto precisam ser guardado o seu tamanho, o que é feito na coluna *Tam* e o número de decimais, no caso de valores reais, embora ambas sejam especificações opcionais. Para um campo de tela, além deles, são necessários a sua posição dentro do vídeo, dados por números de linha e de coluna e o atributo do campo, especificando a aceitação da cadeia vazia e o fato de ser unicamente um campo de saída.

Se o símbolo armazenado é uma tabela de decisão, transformação, processo ou sub-processo, existe um índice de entrada para uma tabela específica. Esta tabela que, neste texto, chamaremos de *TRG*, guarda informações relativas aos elementos relacionados com o método. Uma delas é o tipo do elemento, para indicar basicamente se é uma tabela de decisão, transformação ou sub-processo.

A ordem com que os elementos aparecem no arquivo de entrada é quase totalmente arbitrária. Comumente, eles aparecem após serem referenciados e, no final, o gerador precisa percorrer a tabela e verificar se existe algum elemento referenciado e não declarado. Para isto, existe uma coluna na *TRG* de nome *Inicio*, do tipo “booleano” que é alterado para o valor “verdadeiro” quando o gerador detecta o início de um destes elementos. Por outro lado, podem ocorrer elementos, como transformações, definidas e não referenciadas. Isto é detectado por variações nos tipos dos elementos.

A fase *C* de uma etapa tem a sua execução iniciada pela primeira tabela de decisão; ela serve como um mecanismo, em “leque”, que direciona a execução, de acordo com as condições. Eventualmente, tabelas podem ser invocadas sem retorno, através do comando *goto table*. Estas tabelas e a primeira encerram a execução da etapa. Desta forma, ao final destas, devem ser produzidos desvios para a etapa seguinte ou, no caso de estar na etapa *M*, para o ponto que efetiva o loop característico desta etapa, ou seja, deve ser desviado para a base do loop do último comando de seleção da parte principal. Com relação às tabelas que não são a primeira de uma etapa, quando ocorrem antes de serem referenciadas, não é possível saber se elas serão invocadas com ou sem retorno e, portanto, o gerador desconhece, neste momento, qual deve ser o destino do fluxo de execução ao final delas. Para isto, existe uma coluna na *TRG*, de nome *Destino*, que especifica os possíveis destinos do fluxo de execução, após o término da execução. Alguns dos possíveis valores deste campo são:

LOOP: Deve ser desviado para a base do loop do último comando de seleção.

ROT-RET: Deve seguir para a rotina de retorno, onde é desempilhado o valor indicativo do rótulo para o qual deve ser desviado.

ABERTO: Enquadra-se na situação acima e não foi possível decidir o seu destino, que é deixado para o segundo passo da conversão.

Quando a tarefa de decidir sobre o fluxo de execução fica em aberto, precisamos manter na *TRG* informações que possibilitem uma decisão futura. Primeiramente, ela depende da etapa em que a tabela está presente. Para suprir esta necessidade, a *TRG* possui uma coluna que armazena a etapa em que cada um de seus elementos ocorre. Além disso, é preciso armazenar o sub-processo que contém o elemento. Isto serve para subsidiar o comando de desvio para a base do loop do sub-processo que o contém. Quando o elemento da *TRG* é um sub-processo, precisamos guardar a informação de qual rótulo é a base do loop do último comando de seleção da etapa principal.

Para processos e sub-processos, o gerador produz a declaração dos rótulos utilizados na fase de seleção e, para isto, é mantido na *TRG* o número de ordem dos rótulos produzidos. Também é necessário um apontador para uma lista de arquivos, sobre os quais os comandos dos sub-processos atuam. Cada elemento desta lista contém o índice da *TIG* onde está o arquivo, a etapa em que é utilizado e o modo de sua operação. Com base nestas informações, o gerador insere os comandos de abertura para os arquivos que sofrem inserção e de fechamento para todos eles. Para os sub-processos de agrupamento, existe um apontador para uma lista de funções de agregação neles utilizadas, para permitir a correta inserção dos comandos de inicialização e atualização das variáveis de agrupamento.

Ao final da execução de uma tabela de decisão, o fluxo de execução é alterado, conforme a forma como ela é invocada, através de um comando de desvio, inserido no final do código correspondente à tabela. Em consequência disto, uma tabela chamada com retorno (*exec table*) não pode ser chamada, em outro ponto, sem retorno (*goto table*) e vice-versa. Estas diferenças são registradas, como mencionamos, através de variações no tipo dos elementos da *TRG*, mas o problema é que uma tabela, que tenha na parte de ações um comando de chamada sem retorno para uma tabela, não pode ser invocada através de comando com retorno. Este fato é registrado na *TRG*, indicando se uma tabela de decisão contém ou não uma chamada a outra tabela, através do comando *goto table*. Como este comando é um caso particular de chamada com retorno, em que é a última da lista, talvez fosse prudente não permitir esta forma de chamada a outras tabelas.

5.7 Pilha Auxiliar

Como já mencionamos em diversos pontos, as chamadas de transformações, sub-processos e tabelas de decisão com retorno implicam na necessidade de desviar o fluxo de execução ao comando seguinte à chamada, após o encerramento do objeto chamado. Genericamente, podemos ter a chamada de um mesmo elemento em pontos distintos do programa e assim precisamos de um mecanismo dinâmico para controlar as chamadas e retornos. Além disso, é possível que elementos chamados chamem outros elementos, o que requer uma estrutura de pilha para o correto funcionamento do programa gerado. No momento da chamada, antes de produzir um desvio para o sub-processo, transformação ou tabela de decisão invocada, é empilhada uma

marca para o rótulo do comando seguinte. Quando estes elementos invocados são encerrados, a pilha é consultada numa rotina de retorno, que a desempilha e determina para qual rótulo deve seguir. Além disso, mencionamos, em 5.4.2, que o comando `exit`, introduz a complicação adicional de restaurar a pilha ao seu estado do momento em que se iniciou a etapa *M* de um sub-processo. Como a etapa principal só é iniciada quando encerrada a etapa inicial, o estado da pilha no início da etapa *M*, corresponde ao seu estado, no início da execução do próprio sub-processo. Assim, ocorrendo este comando, precisamos restaurar a pilha para o estado em que foi iniciada a execução do sub-processo. Para isto é necessário empilhar uma marca característica do sub-processo em todas as chamadas para estes elementos, por exemplo, o seu índice na *TRG*. A restauração da pilha consiste no desempilhamento consecutivo desde o topo, até encontrar, na pilha a marca do sub-processo corrente. Uma forma alternativa de superar esta dificuldade foi citada também em 5.4.2.

Para gerenciar esta pilha, existe um módulo, compilado em separado, que contém a declaração de uma pilha e de três funções: uma para empilhamento, outra para desempilhamento e outra para restauração da pilha e que funcionam como uma “caixa preta” para o programa gerado. O funcionamento correto deste mecanismo requer que a base da pilha seja inicializada, para que um comando `%exit` no processo principal faça com que o topo da pilha aponte para a sua base.

Uma chamada com retorno a uma tabela de decisão, sub-processo ou transformação produz o código mostrado em 5.4.3.

No final do código produzido para o programa, são gerados os comandos da rotina de retorno, que, de acordo com o valor desempilhado, determina um fluxo de execução distinto, através de um desvio incondicional, como mostrado a seguir.

```
case Pop_ of
  1 : goto label_1.;
  2 : goto label_2.;
    :
  n : goto label_n.;
end;
```

No esquema anterior, `Pop_` é a função de desempilhamento e `label_i_` é o rótulo que identifica o comando seguinte à chamada da rotina que provocou o empilhamento do índice *i*.

Para que este mecanismo funcione é necessário que, no final da execução de uma transformação, sub-processo ou tabela de decisão chamada pelo comando `exec table`, o fluxo de execução seja desviado para os comandos de retorno, o que é feito através da inserção, pelo gerador, de um desvio incondicional no final do código produzido para cada um desses elementos. Este é o motivo pelo qual uma mesma tabela de decisão não pode ser chamada pelos comandos `goto table` e `exec table`.

As chamadas com retorno poderiam ser implementadas através de procedimentos em Pascal o que dispensaria a declaração da pilha, visto que o mecanismo de retorno seria automaticamente gerenciado pela pilha do sistema de execução, mas teríamos dificuldades para tratar o comando `exit`, que pode produzir o desempilhamento de vários elementos da pilha.

Os loops da etapa *M* são implementados através de um desvio incondicional colocado no final da execução da etapa para a base do loop do último comando de seleção da respectiva

etapa. Este desvio incondicional faz com que os comandos da parte principal sejam executados para todos os elementos do conjunto abstrato, enquanto o comando de controle do loop, visto em 5.3, é gerado para cada comando de seleção e implementa o encerramento do loop. Uma questão que deve ser levantada, neste momento, é qual a parte do código da etapa M que encerra a execução de uma iteração, pois este é o local onde deve ser colocado o comando de desvio para implementar o loop. Nem sempre coincide com o final físico da etapa. Se existem tabelas de decisão, isto ocorre no final da execução da primeira tabela de decisão e das demais tabelas que são invocadas, sem retorno, pelo comando **goto table**. Esta é uma das inconveniências deste comando, pois a sua ausência garantiria um único ponto de encerramento da execução de uma iteração da etapa principal. Se não existe a fase C , este ponto coincide com o final da fase de transformações e que também é o fim físico da etapa.

Capítulo 6

Exemplos

6.1 Exemplo 1

Vamos considerar um exemplo simples, onde se deseja criar um arquivo contendo o nome de alunos com as três notas do semestre e de sua respectiva média, entrando com os dados através de uma tela. A seguinte sequência de especificações produz um programa que executa esta tarefa.

```
%declarations
```

```
%screen Tela (Const 'ENTRADA DE DADOS':1,5;  
    Const 'Nome:' :4,5;  
    Const 'Classe:' :6,5;  
    Const 'Numero:' :6,25;  
    Const 'Nota1:' :8,5;  
    Const 'Nota2:' :10,5;  
    Const 'Nota3:' :12,5;  
    Const 'Pressione F1 para encerrar':14,1;  
    Var Nome :string,NON_EMPTY,4,13,30;  
    Var Classe:int,NON_EMPTY,6,13,2;  
    Var Numero:int,NON_EMPTY,6,33,3;  
    Var Nota1:real,NON_EMPTY,8,13,4,1;  
    Var Nota2:real,NON_EMPTY,10,13,4,1;  
    Var Nota3:real,NON_EMPTY,12,13,4,1;  
    Var Msg:string,OUTPUT,18,5,20;  
    Transmit_Key:F2,F3,F4,F5,F6,F7,F8,F9,F10;  
    End_Key:F1)
```

```
%file Notas (Numero,Classe:int;Nome:string[30];  
    Nota1,Nota2,Nota3,Media:real)  
    on 'Notas.tst'
```

```
%-----
```

```
%process Entrada
```

```
%Main Part  
    %Selection
```



```

%Select Tela
%Choice
%decision table Tabela
%Sets
    Nota_Certa | 0..10 : 0..10 : 0..10
%conditions
    Tela.Numero <= 0 | Y N
    Tela.Nota1:Tela.Nota2:Tela.Nota3 | - ~Nota_Certa
%Actions
    %exec Grava | 1
    %exec Erro_Numero | 1
    %exec Erro_Nota | 1
%end table

%Transformations
%transf Grava
    %insert Notas with Tela,
        Media with (Tela.Nota1 + Tela.Nota2 + Tela.Nota3)/3.0
    %clear Tela
%end Grava

%transf Erro_Numero
    %change Tela.Msg with 'NUMERO INVALIDO'
%end Erro_Numero

%transf Erro_Nota
    %change Tela.Msg with 'NOTA INVALIDA'
%end Erro_Nota

%end process

```

Este arquivo submetido ao gerador I-M-E produziu o seguinte código fonte em Pascal.

```

(*1 GERADOR DE APLICATIVOS - VERSAO EXPERIMENTAL 1.0
DEPARTAMENTO DE CIENCIA DA COMPUTACAO - IME - USP *)

(*2 %declarations *)
(*2
%screen Tela (Const 'ENTRADA DE DADOS':1,5;
    Const 'Nome:' :4,5;
    Const 'Classe:' :6,5;
    Const 'Numero:' :6,25;
    Const 'Nota1:' :8,5;
    Const 'Nota2:' :10,5;
    Const 'Nota3:' :12,5;
    Const 'Pressione F1 para encerrar':14,1;
    Var Nome :string, NON_EMPTY, 4,13,30;
    Var Classe: int, NON_EMPTY, 6,13,2;
    Var Numero: int, NON_EMPTY, 6,33,3;
    Var Nota1: real, NON_EMPTY, 8,13,4,1;
    Var Nota2: real, NON_EMPTY, 10,13,4,1;
    Var Nota3: real, NON_EMPTY, 12,13,4,1;
    Var Msg: string, OUTPUT, 18,5,20;
    Transmit_Key: F2, F3, F4, F5, F6, F7, F8, F9, F10;
    End_Key: F1)

%file Notas (Numero, Classe: int; Nome: string[30];

```

```

        Nota1,Nota2,Nota3,Media:real)
        on 'Notas.tst'

%-----
*)
uses Crt,Printer,Screen_,Stack_;
type
T_TELA_=record
    NOME:string[30];
    CLASSE:integer;
    CLASSE_:string[2];
    NUMERO:integer;
    NUMERO_:string[3];
    NOTA1:real;
    NOTA1_:string[4];
    NOTA2:real;
    NOTA2_:string[4];
    NOTA3:real;
    NOTA3_:string[4];
    MSG:string[20];
    TRANSMIT_KEY:char;
end; (* record *)
T_NOTAS_=record
    NUMERO:integer;
    CLASSE:integer;
    NOME:string[30];
    NOTA1:real;
    NOTA2:real;
    NOTA3:real;
    MEDIA:real;
end; (* record *)
var
    TELA_5_:integer;
    TELA_Buf_:T_TELA_;
    NOTAS_Buf_:T_NOTAS_;
    NOTAS_:file of T_NOTAS_;
    CURRENT_SCREEN_:(NIHIL_, TELA_);
    NOTAS_:(CLOSED_,ENTRADA_);
procedure Initialize_TELA____;
begin
    TELA_Buf_.NOME:='';
    TELA_Buf_.CLASSE:=0;
    TELA_Buf_.NUMERO:=0;
    TELA_Buf_.NOTA1:=0.0;
    TELA_Buf_.NOTA2:=0.0;
    TELA_Buf_.NOTA3:=0.0;
    TELA_Buf_.MSG:='';
    TELA_Buf_.TRANSMIT_KEY:=chr(0);
end;
procedure Display_TELA____(Select_:byte);
var I____,J____:integer;
begin
    textbackground(0);textcolor(7);
    clrscr;
    gotoxy(5,1);write('ENTRADA DE DADOS');
    gotoxy(5,4);write('Nome:');
    gotoxy(5,6);write('Classe:');
    gotoxy(25,6);write('Numero:');
    gotoxy(5,8);write('Nota1:');
    gotoxy(5,10);write('Nota2:');
    gotoxy(5,12);write('Nota3:');
    gotoxy(1,14);write('Pressione F1 para encerrar');
    gotoxy(1,23);for I____:=1 to 80 do write(' ');
    textbackground(7);textcolor(0);
    gotoxy(12,4);for I____:=0 to 31 do write(' ');
    gotoxy(12,6);for I____:=0 to 3 do write(' ');
    gotoxy(32,6);for I____:=0 to 4 do write(' ');

```

```

gotoxy(12,8);for I_:=0 to 5 do write(' ');
gotoxy(12,10);for I_:=0 to 5 do write(' ');
gotoxy(12,12);for I_:=0 to 5 do write(' ');
gotoxy(4,18);for I_:=0 to 21 do write(' ');
J_:=1;
gotoxy(13,4);write(TELA_Buf_.NOME);
gotoxy(13,6);
str(TELA_Buf_.CLASSE,TELA_Buf_.CLASSE_);
if(ord(TELA_Buf_.CLASSE_[0])>2)then TELA_Buf_.CLASSE_[0]:=chr(2);
if(TELA_Buf_.CLASSE=0)then
  if(Select_=1)then TELA_Buf_.CLASSE_:='' else J_:=0;
write(TELA_Buf_.CLASSE_);
gotoxy(33,6);
str(TELA_Buf_.NUMERO,TELA_Buf_.NUMERO_);
if(ord(TELA_Buf_.NUMERO_[0])>3)then TELA_Buf_.NUMERO_[0]:=chr(3);
if(TELA_Buf_.NUMERO=0)then
  if(Select_=1)then TELA_Buf_.NUMERO_:='' else J_:=0;
write(TELA_Buf_.NUMERO_);
gotoxy(13,8);
str(trunc(TELA_Buf_.NOTA1*1E1),TELA_Buf_.NOTA1_);
insert(' ',TELA_Buf_.NOTA1_,ord(TELA_Buf_.NOTA1_[0])-0);
if(ord(TELA_Buf_.NOTA1_[0])>4)then TELA_Buf_.NOTA1_[0]:=chr(4);
if(TELA_Buf_.NOTA1=0)then
  if(Select_=1)then TELA_Buf_.NOTA1_:='' else J_:=0;
write(TELA_Buf_.NOTA1_);
gotoxy(13,10);
str(trunc(TELA_Buf_.NOTA2*1E1),TELA_Buf_.NOTA2_);
insert(' ',TELA_Buf_.NOTA2_,ord(TELA_Buf_.NOTA2_[0])-0);
if(ord(TELA_Buf_.NOTA2_[0])>4)then TELA_Buf_.NOTA2_[0]:=chr(4);
if(TELA_Buf_.NOTA2=0)then
  if(Select_=1)then TELA_Buf_.NOTA2_:='' else J_:=0;
write(TELA_Buf_.NOTA2_);
gotoxy(13,12);
str(trunc(TELA_Buf_.NOTA3*1E1),TELA_Buf_.NOTA3_);
insert(' ',TELA_Buf_.NOTA3_,ord(TELA_Buf_.NOTA3_[0])-0);
if(ord(TELA_Buf_.NOTA3_[0])>4)then TELA_Buf_.NOTA3_[0]:=chr(4);
if(TELA_Buf_.NOTA3=0)then
  if(Select_=1)then TELA_Buf_.NOTA3_:='' else J_:=0;
write(TELA_Buf_.NOTA3_);
gotoxy(5,18);write(TELA_Buf_.MSG);
textbackground(0);textcolor(7);
if(J_:=1)then CURRENT_SCREEN_:=TELA_ else CURRENT_SCREEN_:=NIHIL_;
end;
(*2
%process Entrada *)
procedure ENTRADA;
label
  label_1_,
  label_2_,
  label_3_,
  label_4_,
  label_5_,
  ENTRADA_S_1_,
  ENTRADA_I_,ENTRADA_M_,ENTRADA_E_,
  TABELA,
  GRAVA,
  ERRO_NUMERO,
  ERRO_NOTA;
var
  A1_:string;
  Ch1_:char;
  I1_:integer;
  I2_:integer;
  I3_:integer;
begin
  _XX_[0].Sub_:=0;
  _TT_:=0;

```



```

Initialize_TELA___;
CURRENT_SCREEN_:=NIHIL_;
NOTAS_:=CLOSED_;
assign(NOTAS,'Notas.tst');

(*2
  %Main Part *)
ENTRADA_I_:
ENTRADA_M_:
if(NOTAS_:=CLOSED_)then
  begin
    {$I-}reset(NOTAS);{$I+}
    if ioreult <> 0 then rewrite(NOTAS)
    else seek(NOTAS,filesize(NOTAS));
    NOTAS_:=ENTRADA___;
  end;
(*2
  %Selection *)
(*2
  %Select Tela *)
TELA_5_:=0;(* Screen EOF control *)
ENTRADA_S_1_: I1_:=0;(* select command control *)
if((TELA_5_=0))then
  begin
    if(CURRENT_SCREEN_<>TELA___)then Display_TELA___(1);
    Ch1_:=chr(0);
    Vector_Screen_[F2]:=1; Vector_Screen_[F3]:=1; Vector_Screen_[F4]:=1;
    Vector_Screen_[F5]:=1; Vector_Screen_[F6]:=1; Vector_Screen_[F7]:=1;
    Vector_Screen_[F8]:=1; Vector_Screen_[F9]:=1; Vector_Screen_[F10]:=1;
    Vector_Screen_[F1]:=2;
    I2_:=1;
    while (I2_<=6) do begin
      case (I2_)of
        -1:begin
          clrscr;
          CURRENT_SCREEN_:=NIHIL_;
          TELA_5_:= 1;
          I2_:=7;
        end;
        0:begin
          TELA_Buf_.TRANSMIT_KEY:=Ch1_;
          if(TELA_Buf_.TRANSMIT_KEY=chr(0))then
            TELA_Buf_.TRANSMIT_KEY:=F2;
            I2_:=7;
          end;
        1:begin
          A1_:=TELA_Buf_.NOME;
          Le_Strn_(0,A1_,30,13,4,6,I2_,2,Ch1_);
          TELA_Buf_.NOME:=A1_;
        end;
        2:begin
          A1_:=TELA_Buf_.CLASSE_;
          Le_Strn_(1,A1_,2,13,6,6,I2_,2,Ch1_);
          TELA_Buf_.CLASSE_:=A1_;
          val(A1_,TELA_Buf_.CLASSE,I3_);
        end;
        3:begin
          A1_:=TELA_Buf_.NUMERO_;
          Le_Strn_(1,A1_,3,33,6,6,I2_,2,Ch1_);
          TELA_Buf_.NUMERO_:=A1_;
          val(A1_,TELA_Buf_.NUMERO,I3_);
        end;
        4:begin
          A1_:=TELA_Buf_.NOTA1_;
          Le_Strn_(2,A1_,4,13,8,6,I2_,2,Ch1_);
          TELA_Buf_.NOTA1_:=A1_;
          val(A1_,TELA_Buf_.NOTA1,I3_);

```

```

end;
5:begin
  A1_:=TELA_Buf_.NOTA2_;
  Le_Strn_(2,A1_,4,13,10,6,I2_,2,Ch1_);
  TELA_Buf_.NOTA2_:=A1_;
  val(A1_,TELA_Buf_.NOTA2,I3_);
end;
6:begin
  A1_:=TELA_Buf_.NOTA3_;
  Le_Strn_(2,A1_,4,13,12,6,I2_,2,Ch1_);
  TELA_Buf_.NOTA3_:=A1_;
  val(A1_,TELA_Buf_.NOTA3,I3_);
end;
end; end;
if (TELA_5_=0)then I1_:=1;
Vector_Screen_[F2]:=0; Vector_Screen_[F3]:=0; Vector_Screen_[F4]:=0;
Vector_Screen_[F5]:=0; Vector_Screen_[F6]:=0; Vector_Screen_[F7]:=0;
Vector_Screen_[F8]:=0; Vector_Screen_[F9]:=0; Vector_Screen_[F10]:=0;
Vector_Screen_[F1]:=0;
end;
if (I1_=0)then goto ENTRADA_E_;
(*2
  %Choice *)
(*2
  %decision table Tabela
  %Sets
  Nota_Certa | 0..10 : 0..10 : 0..10
  %conditions
  Tela.Numero <= 0 | Y N
  Tela.Nota1:Tela.Nota2:Tela.Nota3 | - Nota_Certa
  %Actions
  %exec Grava | 1
  %exec Erro_Numero | 1
  %exec Erro_Nota | 1
  %end table *)
TABELA:
  if (TELA_Buf_.Numero<=0) then
    begin
      (* rule 1 *)
      Push_(1,3);
      goto ERRO_NUMERO;
    label_1_:
      end
    else
      begin
        if (not(((TELA_Buf_.Nota1>=0)and(TELA_Buf_.Nota1<=10))and((TELA_Buf_.Nota2
        >=0)and(TELA_Buf_.Nota2<=10))and((TELA_Buf_.Nota3>=0)and(TELA_Buf_.Nota3<=
        10)))) then
          begin
            (* rule 2 *)
            Push_(2,4);
            goto ERRO_NOTA;
          label_2_:
            end
          else
            begin
              (* else rule *)
              Push_(3,2);
              goto GRAVA;
            label_3_:
              end;
            end;
            goto ENTRADA_S_1_;
          (*2
            %Transformations *)
          (*2

```

```

%transf Grava *)
GRAVA:
(*2          %insert Notas with Tela,
              Media with (Tela.Nota1 + Tela.Nota2 + Tela.Nota3)/3.0 *)
NOTAS_Buf_.NUMERO:=TELA_Buf_.NUMERO;
NOTAS_Buf_.CLASSE:=TELA_Buf_.CLASSE;
NOTAS_Buf_.NOME:=TELA_Buf_.NOME;
NOTAS_Buf_.NOTA1:=TELA_Buf_.NOTA1;
NOTAS_Buf_.NOTA2:=TELA_Buf_.NOTA2;
NOTAS_Buf_.NOTA3:=TELA_Buf_.NOTA3;
NOTAS_Buf_.MEDIA:=(TELA_Buf_.Nota1+TELA_Buf_.Nota2+TELA_Buf_.Nota3)/3.0;
write(NOTAS,NOTAS_Buf_);
(*2          %clear Tela *)
Initialize_TELA___;
clrscr;
CURRENT_SCREEN_:=NIHIL_;
(*2          %end Grava *)
goto label_4_;

(*2          %transf Erro_Numero *)
ERRO_NUMERO:
(*2          %change Tela.Msg with 'NUMERO INVALIDO' *)
TELA_Buf_.Msg:='NUMERO INVALIDO';
Display_TELA___(0);
(*2          %end Erro_Numero *)
goto label_4_;

(*2          %transf Erro_Nota *)
ERRO_NOTA:
(*2          %change Tela.Msg with 'NOTA INVALIDA' *)
TELA_Buf_.Msg:='NOTA INVALIDA';
Display_TELA___(0);
(*2          %end Erro_Nota *)
goto label_4_;
ENTRADA_E_:
if(NOTAS__=ENTRADA__)then
begin
close(NOTAS);
NOTAS__:=CLOSED_;
end;
goto label_5_;

(*2
%end process *)
label_4_:
case(Pop_) of
1: goto label_1_;
2: goto label_2_;
3: goto label_3_;
end;
label_5_:
end;
begin ENTRADA;clrscr; end.

```

6.2 Exemplo 2

Suponhamos agora que o usuário deseja verificar seqüencialmente os registros digitados para eventuais correções. Desejando interromper a execução, ele poderá fazê-lo pressionando a tecla

F1. Além disso, a tela mostrará a média das notas do aluno, mas o programa não permite que o usuário altere este campo, visto que seu valor será automaticamente recalculado. Neste caso, vamos apresentar somente as especificações para o gerador sem o código gerado, porque ele é semelhante ao do exemplo anterior.

```
%declarations
%screen Tela (Const 'ENTRADA DE DADOS':1,5;
               Const 'Nome:' :4,5;
               Const 'Classe:' :6,5;
               Const 'Numero:' :6,25;
               Const 'Nota1:' :8,5;
               Const 'Nota2:' :10,5;
               Const 'Nota3:' :12,5;
               Const 'Media:' :14,5;
               Const 'Pressione F1 para encerrar':16,1;
               Var Nome :string,NON_EMPTY,4,13,30;
               Var Classe:int,NON_EMPTY,6,13,2;
               Var Numero:int,NON_EMPTY,6,33,3;
               Var Nota1:real,NON_EMPTY,8,14,4,1;
               Var Nota2:real,NON_EMPTY,10,14,4,1;
               Var Nota3:real,NON_EMPTY,12,14,4,1;
               Var Media:real,OUTPUT,14,13,5,2;
               Var Msg:string,OUTPUT,18,5,20;
               Transmit_Key:F1,F2)

%file Notas (Numero,Classe:int;Nome:string[30];
             Nota1,Nota2,Nota3,Media:real)
             on 'Notas.tst'

%-----

%process Altera

%Main Part

%Selection
%Select Notas
%Select Tela thru subprocess Corrige_Notas
%choice
%dec table Encerra
%conditions
Tela.Transmit_Key | F2
%Actions
%exit | 1
%exec Regrava_Notas | 1
%end table

%T
%transf Regrava_Notas
%change Notas with Tela
%end Regrava_Notas

%end process
```

```

%subprocess Corrige_Notas

%Initialization part
%T
    %change Tela with Notas,
        Tela.Msg with ''

%main part
%S
    %Select Tela
%Choice
    %decision table Tabela
    %Sets
        Nota_Errada | ~(0..10:0..10:0..10)
    %conditions
        Tela.Numero <= 0 | Y N
        Tela.Nota1:Tela.Nota2:Tela.Nota3 | - Nota_Errada
    %Actions
        %exec Recalcula_Media | 1
        %exit | 2
        %exec Erro_Numero | 1
        %exec Erro_Nota | 1
    %end table

%Transformations

%transf Erro_Numero
    %change Tela.Msg with 'NUMERO INVALIDO'
%end Erro_Numero

%transf Erro_Nota
    %change Tela.Msg with 'NOTA INVALIDA'
%end Erro_Nota

%transf Recalcula_Media
    %change Tela.Media with (Tela.Nota1 + Tela.Nota2 +
        Tela.Nota3)/3.0
%end Recalcula_Media

%end subprocess

```

6.3 Exemplo 3

Vamos supor para este exemplo que o arquivo produzido pelos programas anteriores contenham os registros por ordem de classe. A seguinte seqüência de especificações produz um arquivo do tipo texto com o nome dos alunos e suas respectivas notas e um arquivo resumo onde cada linha contém o número de uma classe, o número de alunos e a média das notas desta classe.

```
%declarations

%file Notas (Numero,Classe:int;Nome:string[30];
             Nota1,Nota2,Nota3,Media:real)
             on 'Notas.tst'

%file Alunos(Numero,Classe:int,5;Nome:string[30];
             Nota1,Nota2,Nota3,Media:real,5,2)
             on 'Alunos.tst' (text)

%file Resumo(Classe,Numero:int,5;Media:real,5,2) on 'Resumo.tst' (text)
%variable Classe_Anterior:int
%-----

%process Resume_Por_Classe

%Main part
%Selection
%select Notas grouping by Notas.Classe exec subprocess Sub_Resumo
%Transformations
%insert Alunos with Notas
%change Classe_Anterior with Notas.Classe
%end process

%subprocess Sub_Resumo
%Ending part
%Transformations
%insert Resumo.Classe with Classe_Anterior,
                        Numero with %Count,
                        Media  with %average(Notas.Media)
%end subprocess
```

O código produzido pelo gerador para este exemplo é o seguinte.

```
(*1 GERADOR DE APLICATIVOS - VERSAO EXPERIMENTAL 1.0
DEPARTAMENTO DE CIENCIA DA COMPUTACAO - IME - USP *)

(*2 %declarations *)
(*2
%file Notas (Numero,Classe:int;Nome:string[30];
             Nota1,Nota2,Nota3,Media:real)
             on 'Notas.tst'

%file Alunos(Numero,Classe:int,5;Nome:string[30];
             Nota1,Nota2,Nota3,Media:real,5,2)
             on 'Alunos.tst' (text)

%file Resumo(Classe,Numero:int,5;Media:real,5,2) on 'Resumo.tst' (text)
%variable Classe_Anterior:int
%-----
*)
uses Crt,Printer,Stack_;
type
T_NOTAS_=record
```



```

NUMERO:integer;
CLASSE:integer;
NOME:string[30];
NOTA1:real;
NOTA2:real;
NOTA3:real;
MEDIA:real;
end; (* record *)
T_ALUNOS:=record
  NUMERO:integer;
  CLASSE:integer;
  NOME:string[30];
  NOTA1:real;
  NOTA2:real;
  NOTA3:real;
  MEDIA:real;
end; (* record *)
T_RESUMO:=record
  CLASSE:integer;
  NUMERO:integer;
  MEDIA:real;
end; (* record *)
var
  NOTAS_Buf_:T_NOTAS_;
  NOTAS:file of T_NOTAS_;
  CLASSE_P7_:integer;
  CLASSE_A7_:integer;
  ALUNOS_Buf_:T_ALUNOS_;
  ALUNOS:text;
  RESUMO_Buf_:T_RESUMO_;
  RESUMO:text;
  CLASSE_ANTERIOR:integer;
  RESUMO__,
  ALUNOS__,
  NOTAS__: (CLOSED_,RESUME_POR_CLASSE__,SUB_RESUMO__);
  Count_SUB_RESUMO_:integer;
  Total_SUB_RESUMO_MEDIA13_:real;
(*2
%process Resume_Por_Classe *)
procedure RESUME_POR_CLASSE;
label
  label_1_,
  label_2_,
  label_3_,
  label_4_,
  label_5_,
  RESUME_POR_CLASSE_S_1_,
  RESUME_POR_CLASSE_I_,RESUME_POR_CLASSE_M_,RESUME_POR_CLASSE_E_,
  SUB_RESUMO,
  SUB_RESUMO_I_,
  SUB_RESUMO_M_,
  SUB_RESUMO_E_;
var
  I1_:integer;
begin
  _XX_[0].Sub_:=0;
  _TT_:=0;
  RESUMO__:=CLOSED_;
  assign(RESUMO,'Resumo.tst');
  ALUNOS__:=CLOSED_;
  assign(ALUNOS,'Alunos.tst');
  NOTAS__:=CLOSED_;
  assign(NOTAS,'Notas.tst');

(*2
  %Main part *)
  RESUME_POR_CLASSE_I_:

```

```

RESUME_POR_CLASSE_M_:
if(ALUNOS_==CLOSED_)then
  begin
    {$I-}append(ALUNOS);{$I+}
    if ioresult <> 0 then rewrite(ALUNOS);
    ALUNOS_:=RESUME_POR_CLASSE_;
  end;
(*2
  %Selection *)
(*2
  %select Notas grouping by Notas.Classe exec subprocess Sub_Resumo *)
reset(NOTAS);
NOTAS_:=RESUME_POR_CLASSE_;
CLASSE_P7:=1;(* First transaction indication *)
RESUME_POR_CLASSE_S_1_: I1:=0;(* select command control *)
if ((not(eof(NOTAS))))then
  begin
    read(NOTAS,NOTAS_Buf_);
    I1:=1;
  end;
if (I1=0)then
  begin
    if(CLASSE_P7=0)then
      begin
        Push_(1,1);
        goto SUB_RESUMO_E_;
      label_1_:
        end;
        goto RESUME_POR_CLASSE_E_;
      end;
    if((CLASSE_P7=0)and(CLASSE_A7_<>NOTAS_Buf_.CLASSE))then
      begin
        Push_(2,1);
        goto SUB_RESUMO_E_;
      label_2_:
        end;
    if((CLASSE_P7=1)or(CLASSE_A7_<>NOTAS_Buf_.CLASSE))then
      begin
        Push_(3,1);
        goto SUB_RESUMO_I_;
      label_3_:
        CLASSE_P7:=0;
        CLASSE_A7_:=NOTAS_Buf_.CLASSE;
      end;
    inc(Count_SUB_RESUMO_);
    Total_SUB_RESUMO_MEDIA13_:=Total_SUB_RESUMO_MEDIA13_+NOTAS_Buf_.MEDIA;
  (*2
    %Transformations *)
  (*2
    %insert Alunos with Notas *)
  ALUNOS_Buf_.NUMERO:=NOTAS_Buf_.NUMERO;
  ALUNOS_Buf_.CLASSE:=NOTAS_Buf_.CLASSE;
  ALUNOS_Buf_.NOME:=NOTAS_Buf_.NOME;
  ALUNOS_Buf_.NOTA1:=NOTAS_Buf_.NOTA1;
  ALUNOS_Buf_.NOTA2:=NOTAS_Buf_.NOTA2;
  ALUNOS_Buf_.NOTA3:=NOTAS_Buf_.NOTA3;
  ALUNOS_Buf_.MEDIA:=NOTAS_Buf_.MEDIA;
  writeln(ALUNOS,ALUNOS_Buf_.NUMERO:5,' ',
  ALUNOS_Buf_.CLASSE:5,' ',
  ALUNOS_Buf_.NOME,' ':30-ord(ALUNOS_Buf_.NOME[0]),' ',
  ALUNOS_Buf_.NOTA1:5:2,' ',
  ALUNOS_Buf_.NOTA2:5:2,' ',
  ALUNOS_Buf_.NOTA3:5:2,' ',
  ALUNOS_Buf_.MEDIA:5:2);
  (*2
    %change Classe_Anterior with Notas.Classe *)
  CLASSE_ANTERIOR:=NOTAS_Buf_.Classe;
  goto RESUME_POR_CLASSE_S_1_;
RESUME_POR_CLASSE_E_:

```

```

if (NOTAS__=RESUME_POR_CLASSE__) then
  begin
    close (NOTAS);
    NOTAS__:=CLOSED_;
  end;
if (ALUNOS__=RESUME_POR_CLASSE__) then
  begin
    close (ALUNOS);
    ALUNOS__:=CLOSED_;
  end;
goto label_5_;
(*2
%end process *)

(*2
%subprocess Sub_Resumo *)
SUB_RESUMO:
(*2
  %Ending part *)
(*2
  %Transformations *)
SUB_RESUMO_I_:
Count_SUB_RESUMO_:=0;
Total_SUB_RESUMO_MEDIA13_:=0;
SUB_RESUMO_M_:
goto label_4_;
SUB_RESUMO_E_:
if (RESUMO__=CLOSED_) then
  begin
    {$I-}append (RESUMO);{$I+}
    if ioresult <> 0 then rewrite (RESUMO);
    RESUMO__:=SUB_RESUMO__;
  end;
(*2
  %insert Resumo.Classe with Classe_Anterior, *)
(*2
  Numero with %Count,
  Media with %average (Notas.Media) *)
RESUMO_Buf_.Classe:=CLASSE_ANTERIOR;
RESUMO_Buf_.NUMERO:=Count_SUB_RESUMO_;
RESUMO_Buf_.MEDIA:=(Total_SUB_RESUMO_MEDIA13_/Count_SUB_RESUMO_);
writeln (RESUMO, RESUMO_Buf_.CLASSE:5, ' ',
RESUMO_Buf_.NUMERO:5, ' ',
RESUMO_Buf_.MEDIA:5:2);
if (RESUMO__=SUB_RESUMO__) then
  begin
    close (RESUMO);
    RESUMO__:=CLOSED_;
  end;
(*2
%end subprocess *)
label_4_:
  case (Pop_) of
    1: goto label_1_;
    2: goto label_2_;
    3: goto label_3_;
  end;
label_5_:
end;
begin RESUME_POR_CLASSE; clrscr; end.

```


6.4 Exemplo 4

Vamos considerar o problema de intercalar dois arquivos que já estão ordenados por um determinado campo, produzindo um terceiro arquivo. A seguir está especificado um processo que executa esta tarefa, mas como ela é muito comum talvez fosse melhor fornecer um comando que produzisse a intercalação.

```
%declarations

%file Arquivo1(Nome:string[20]) on 'Arquivo1.tst'    (text)
%file Arquivo2(Nome:string[20]) on 'Arquivo2.tst'    (text)
%file Arquivo_Saida(Nome:string[20]) on 'Arquivo3.tst' (text)
%variable Inicio:boolean

%process Intercala

%Initialization part of Intercala
  %Transformations part
    %change Inicio with true

%Main part of Intercala
  %Selection phase of Main part of Intercala process
    %select Arquivo1 when Arquivo1.Nome <= Arquivo2.Nome or Inicio,
      Arquivo2 when Arquivo2.Nome < Arquivo1.Nome or Inicio

%Choice phase of Main part of Itercala process

%decision table Escolhe_Primeiro
  %Conditions
    Arquivo1.Nome <= Arquivo2.Nome | Y  N
  %Actions
    %exec Grava_Arquivo1           | 1
    %exec Grava_Arquivo2           | 1
    %exec Reset_Inicio             | 2  2
  %end table Escolhe_Primeiro

%Transformations phase of Main part
  %transf Grava_Arquivo1
    %insert Arquivo_Saida with Arquivo1
  %end Grava_Arquivo1

  %transf Grava_Arquivo2
    %insert Arquivo_Saida with Arquivo2
  %end Grava_Arquivo2

  %transf Reset_Inicio
    %change Inicio with false
  %end Reset_Inicio

%end process Intercala
```

O código produzido pelo gerador para este processo seria o seguinte.

```
(*1 GERADOR DE APLICATIVOS - VERSAO EXPERIMENTAL 1.0
  DEPARTAMENTO DE CIENCIA DA COMPUTACAO - IME - USP *)

(*2 %declarations *)
(*2
  %file Arquivo1(Nome:string[20]) on 'Arquivo1.tst'    (text)
  %file Arquivo2(Nome:string[20]) on 'Arquivo2.tst'    (text)
  %file Arquivo_Saida(Nome:string[20]) on 'Arquivo3.tst' (text)
  %variable Inicio:boolean
  *)
  uses Crt,Printer,Stack_;
  type
    T_ARQUIVO1_=record
      NOME:string[20];
    end; (* record *)
    T_ARQUIVO2_=record
      NOME:string[20];
    end; (* record *)
    T_ARQUIVO_SAIDA_=record
      NOME:string[20];
    end; (* record *)
  var
    ARQUIVO1_Buf_:T_ARQUIVO1_;
    ARQUIVO1:text;
    ARQUIVO2_Buf_:T_ARQUIVO2_;
    ARQUIVO2:text;
    ARQUIVO_SAIDA_Buf_:T_ARQUIVO_SAIDA_;
    ARQUIVO_SAIDA:text;
    INICIO:boolean;
    ARQUIVO_SAIDA__,
    ARQUIVO2__,
    ARQUIVO1__:(CLOSED_,INTERCALA__);
  (*2
  %process Intercala *)
  procedure INTERCALA;
  label
    label_1_,
    label_2_,
    label_3_,
    label_4_,
    label_5_,
    label_6_,
    INTERCALA_S_1_,
    INTERCALA_S_2_,
    INTERCALA_I_,INTERCALA_M_,INTERCALA_E_,
    ESCOLHE_PRIMEIRO,
    GRAVA_ARQUIVO1,
    GRAVA_ARQUIVO2,
    RESET_INICIO;
  var
    I1_:integer;
  begin
    _XX_[0].Sub_:=0;
    _TT_:=0;
    ARQUIVO_SAIDA__:=CLOSED_;
    assign(ARQUIVO_SAIDA,'Arquivo3.tst');
    ARQUIVO2__:=CLOSED_;
    assign(ARQUIVO2,'Arquivo2.tst');
    ARQUIVO1__:=CLOSED_;
    assign(ARQUIVO1,'Arquivo1.tst');

  (*2
  %Initialization part of Intercala *)
  INTERCALA_I_:
```

```

(*2
  %Transformations part *)
(*2    %change Inicio with true
*)
INICIO:=(TRUE);
(*2
%Main part of Intercala *)
INTERCALA_M_:
if (ARQUIVO_SAIDA_==CLOSED_) then
  begin
    {$I-}append(ARQUIVO_SAIDA);{$I+}
    if ioresult <> 0 then rewrite(ARQUIVO_SAIDA);
    ARQUIVO_SAIDA_:=INTERCALA_M_;
  end;
(*2
%Selection phase of Main part of Intercala process *)
(*2
  %select Arquivo1 when Arquivo1.Nome <= Arquivo2.Nome or Inicio,
    Arquivo2 when Arquivo2.Nome < Arquivo1.Nome or Inicio
*)
reset(ARQUIVO1);
ARQUIVO1_:=INTERCALA_M_;
reset(ARQUIVO2);
ARQUIVO2_:=INTERCALA_M_;
INTERCALA_S_1_: I1_:=0;(* select command control *)
if ((not eof(ARQUIVO1))and((ARQUIVO1_Buf_.Nome<=ARQUIVO2_Buf_.Nome)or(INICIO)))
then
  begin
    readln(ARQUIVO1,
      ARQUIVO1_Buf_.NOME);
    I1_:=1;
  end;
INTERCALA_S_2_:
if ((not eof(ARQUIVO2))and((ARQUIVO2_Buf_.Nome<ARQUIVO1_Buf_.Nome)or(INICIO)))
then
  begin
    readln(ARQUIVO2,
      ARQUIVO2_Buf_.NOME);
    I1_:=1;
  end;
if (I1_=0) then goto INTERCALA_E_;
(*2
  %Choice phase of Main part of Intercala process *)
(*2
  %decision table Escolhe_Primeiro
  %Conditions
  Arquivo1.Nome <= Arquivo2.Nome | Y  N
  %Actions
  %exec Grava_Arquivo1           | 1
  %exec Grava_Arquivo2           | 1
  %exec Reset_Inicio             | 2  2
  %end table Escolhe_Primeiro *)
ESCOLHE_PRIMEIRO:
if (ARQUIVO1_Buf_.Nome<=ARQUIVO2_Buf_.Nome) then
  begin
    (* rule 1 *)
    Push_(1,2);
    goto GRAVA_ARQUIVO1;
label_1_:
    Push_(2,4);
    goto RESET_INICIO;
label_2_:
  end
else
  begin
    (* rule 2 *)

```



```

    Push_(3,3);
    goto GRAVA_ARQUIVO2;
label_3_:
    Push_(4,4);
    goto RESET_INICIO;
label_4_:
    end;
goto INTERCALA_S_1_;

(*2
  %Transformations phase of Main part *)
(*2
  %transf Grava_Arquivo1 *)
GRAVA_ARQUIVO1:
(*2      %insert Arquivo_Saida with Arquivo1 *)
ARQUIVO_SAIDA_Buf_.NOME:=ARQUIVO1_Buf_.NOME;
writeln(ARQUIVO_SAIDA,ARQUIVO_SAIDA_Buf_.NOME,' ' :20-ord(ARQUIVO_SAIDA_Buf_.NOME
[0]));
(*2
  %end Grava_Arquivo1 *)
goto label_5_;

(*2
  %transf Grava_Arquivo2 *)
GRAVA_ARQUIVO2:
(*2      %insert Arquivo_Saida with Arquivo2 *)
ARQUIVO_SAIDA_Buf_.NOME:=ARQUIVO2_Buf_.NOME;
writeln(ARQUIVO_SAIDA,ARQUIVO_SAIDA_Buf_.NOME,' ' :20-ord(ARQUIVO_SAIDA_Buf_.NOME
[0]));
(*2
  %end Grava_Arquivo2 *)
goto label_5_;

(*2
  %transf Reset_Inicio *)
RESET_INICIO:
(*2      %change Inicio with false *)
INICIO:=(FALSE);
(*2
  %end Reset_Inicio *)
goto label_5_;
INTERCALA_E_:
if(ARQUIVO1__=INTERCALA__)then
  begin
    close(ARQUIVO1);
    ARQUIVO1__:=CLOSED_;
  end;
if(ARQUIVO2__=INTERCALA__)then
  begin
    close(ARQUIVO2);
    ARQUIVO2__:=CLOSED_;
  end;
if(ARQUIVO_SAIDA__=INTERCALA__)then
  begin
    close(ARQUIVO_SAIDA);
    ARQUIVO_SAIDA__:=CLOSED_;
  end;
goto label_6_;

(*2
  %end process Intercala *)
label_5_:
  case(Pop_) of
    1: goto label_1_;
    2: goto label_2_;
    3: goto label_3_;
    4: goto label_4_;
  end;

```

```

end;
label_6_:
end;
begin INTERCALA;clrscr; end.

```

6.5 Exemplo 5

A seguir apresentamos as especificações para o carregador sintático descrito em [Se-Me 83]. Dado um conjunto de sub-grafos que descrevem uma gramática, o programa produz uma tabela de terminais, outra de não terminais e uma terceira que é a representação do grafo sintático a ser seguido pelo analisador ali descrito. Os elementos da primeira tabela são armazenados através de espalhamento (“hashing”) e os das duas outras tabelas são armazenados seqüencialmente. O programa seguinte produz no seu final três arquivos do tipo texto que contém estas três tabelas. Se supormos que o arquivo de sub-grafos contém somente linhas que descrevem os seus vértices, as especificações seriam mais simples. Aqui consideramos que o usuário construiu o arquivo que descreve os sub-grafos com linhas de comentário, para facilitar uma conferência. Como o código gerado é relativamente extenso, apresentamos somente as especificações para o gerador sem o código gerado.

```

%declarations
  %file Arquivo
    (Linha:string[80]) on 'Gerador.ttt' (text)

  %table Tabela_Simbolos[0..270](Simbolo:string[20];Colisao:integer)
  %table Tabela_Nt[1..32](Simbolo:string[20];Prim_Trans:integer)
  %table Tabgrafo[0..900](Ter:boolean;Sim,Altr,Sucr,Semr:integer)

  %file Arq_Tabela_Simb(Indice:int,5;Simbolo:string[20];Colisao:int,10)
    on 'Tab.Smb' (text)
  %file Arq_Tabela_Nt(Indice:int,5;Simbolo:string[20];Prim_Trans:int,10)
    on 'Tab.Nt' (text)
  %file Arq_Tabela_Gf(Indice:int,5;Simbolo:string[20];Sim:integer,5;
    Ter:boolean,7;Altr,Sucr,Semr:int,5) on 'Tab.Gf' (text)

  %variable Livre,Modulo,Ant,Maxnt,Maxgrafo,Endereco : int
  %variable Simbolo:string[20]
  %variable Numno,Altr,Sucr,Semr:int
  %variable Tipo,Nulo:string[1]

%process Carregador

%initialization
  %transformations
  %initialize Tabela_Simbolos(Simbolo : ''; Colisao : -1)
  %initialize Tabela_Nt(Simbolo:'';Prim_Trans : 0)
  %initialize Tabgrafo(Sim:-1;Ter:false;Altr:0;Sucr:0;Semr:0)
  %change Modulo with 191
  %change current of Tabgrafo with 0
  %change Nulo with ''

```

```

%change Current of Tabela_Simbolos with 191
%change Tabela_Simbolos.Simbolo with 'INTEGER'
%change Current of Tabela_Simbolos with 192
%change Tabela_Simbolos.Simbolo with 'STRING'
%change Current of Tabela_Simbolos with 193
%change Tabela_Simbolos.Simbolo with 'IDENT'
%change Current of Tabela_Simbolos with 194
%change Tabela_Simbolos.Simbolo with 'NEWLINE'
%change Current of Tabela_Simbolos with 195
%change Tabela_Simbolos.Simbolo with 'REAL'
%change Current of Tabela_Simbolos with 196
%change Tabela_Simbolos.Simbolo with 'EOF'
%change Livre with 197;
%change Ant with 0;
%change Maxnt with 0;

%Main part
%selection
  %select Arquivo where Arquivo.Linha <> Nulo

%Choice

%decision table Decide_Arquivo
  %preparation
    %exec Obtem_Tipo_da_Linha
  %Conditions
    Tipo
  %Actions
    %exec Obtem_Elementos_da_Linha | 1 1 1 1
    %exec table Decide_Terminais_Especiais | 2
    %exec subprocess Insere_Simbolo_Hashing | 2
    %exec subprocess Insere_Nao_Terminal | 2
    %exec Obtem_Novo_No_do_Grafo | 3 3 3
    %exec Insere_Tabgrafo | 4 4 4
    %exec Atualiza_No_como_Terminal | 5 5
    %exec Atualiza_No_como_Nao_Terminal | 5
    %exec table Decide_Sucessor | 6 6 6
    %exec table Decide_Alternativa | 7 7 7
  %end table

%decision table Decide_Alternativa
  %conditions
    Altr = 0 | N
  %Actions
    %exec Acerta_Tabgrafo_Altr | 1
  %end table

%decision table Decide_Sucessor
  %conditions
    Sucl = 0 | N
  %Actions
    %exec Acerta_Tabgrafo_Sucl | 1
  %end table

```



```

%decision table Decide_Terminais_Especiais
%sets
C1 | 'LAMBDA'
C2 | 'OUTROS'
C3 | 'INTEIRO'
C4 | 'ALFA'
C5 | 'IDENT'
C6 | 'NEWLINE'
C7 | 'RREAL'
C8 | 'FIM'
%Conditions
Simbolo | C1 C2 C3 C4 C5 C6 C7 C8
%Actions
Endereco := -1 | 1
Endereco := -2 | 1
Endereco := 191 | 1
Endereco := 192 | 1
Endereco := 193 | 1
Endereco := 194 | 1
Endereco := 195 | 1
Endereco := 196 | 1
%end table

%Transformations

%transf Obtem_Tipo_da_Linha
%change Tipo with Arquivo.Linha starting_at 1 to 1
%end Obtem_Tipo_da_Linha

%transf Obtem_Elementos_da_Linha
%change Simbolo with Arquivo.Linha starting_at 8 to 27
%remove left right blanks from Simbolo
%change Numno with Arquivo.Linha starting_at 28 to 30
%change Altr with Arquivo.Linha starting_at 41 to 43
%change Sucr with Arquivo.Linha starting_at 54 to 56
%change Semr with Arquivo.Linha starting_at 67 to 70
%change Endereco with 0
%end Obtem_Elementos_da_Linha

%transf Obtem_Novo_No_do_Grafo
%change Current of Tabgrafo with Tabgrafo.Current + 1
%end Obtem_Novo_No_do_Grafo

%transf Atualiza_No_como_Terminal
%change Tabgrafo.Ter with true
%end Atualiza_No_como_Terminal

%transf Atualiza_No_como_Nao_Terminal
%change Tabgrafo.Ter with false
%end Atualiza_No_como_Nao_Terminal

%transf Insere_Tabgrafo

```

```

%change Tabgrafo.Sim with Endereco,
    Tabgrafo.Semr with Semr
%end Insere_Tabgrafo

%transf Acerta_Tabgrafo_Sucr
    %change Tabgrafo.Sucr with Sucl + Ant
%end Acerta_Tabgrafo_Sucr

%transf Acerta_Tabgrafo_Altr
    %change Tabgrafo.Altr with Altr + Ant
%end Acerta_Tabgrafo_Altr

%ending part
%transformations
    %change Maxgrafo with Tabgrafo.Current
    %exec subprocess Grava_Tabela_Simbolos
    %exec subprocess Grava_Tabela_Nt
    %exec subprocess Grava_Tabgrafo
%end process

%subprocess Insere_Simbolo_Hashing
%Initialization part
%transformations
    %change Endereco with Hashing_(Simbolo,Modulo);

%Main part
%Selection
    %select Tabela_Simbolos where Current = Endereco
%Choice
    %decision table Decide_Entrada_Hashing
    %Sets
        Vaz      | ''
        TemC     | >0
    %Cond
        Tabela_Simbolos.Simbolo      | Vaz  ~Vaz  ~Vaz  ~Vaz
        Tabela_Simbolos.Colisao      | -    -    ~TemC  TemC
        Tabela_Simbolos.Simbolo=Simbolo | *    Y    N    N
    %Actions
        %exec Prepara_Area_Colisao    |      1
        %exec Selecciona_Proxima_Colisao |      1
        %exec Insere_Terminal          | 1    2
        %exit                          | 2    1    3
    %end table

%Transformations

%transf Prepara_Area_Colisao
    %change Tabela_Simbolos.Colisao with Livre
    %change Current of Tabela_Simbolos with Livre
    %change Endereco with Livre
    %inc Livre
%end Prepara_Area_Colisao

```

```

%transf Selecciona_Proxima_Colisao
  %change Endereco with Tabela_Simbolos.Colisao
%end Selecciona_Proxima_Colisao

%transf Insere_Terminal
  %change Tabela_Simbolos.Simbolo with Simbolo
  %change Tabela_Simbolos.Colisao with 0
%end Insere_Terminal

%end subprocess Insere_simbolo_Hashing

%subprocess Insere_Nao_Terminal
  %Main part
  %selection
    %select (1) Tabela_Nt starting_at 1 to Maxnt where
      Tabela_Nt.Simbolo = Simbolo
  %transformations
    %change Endereco with Tabela_Nt.Current
  %Ending part

  %Choice
  %decision table Insere_Simbolo
    %Conditions
      Endereco > 0
      Tipo
    %Actions
      %exec Aloca_Entrada_Nao_Terminal
      %exec Acerta_Ultimo_No_Sub_anterior
      %exec Atualiza_Simbolo_Nao_Terminal
      %exec Atualiza_Prim_Trans
    %end table

  %Transformations
    %transf Acerta_Ultimo_No_Sub_anterior
      %change Ant with Tabgrafo.Current
    %end Acerta_Ultimo_No_Sub_anterior

    %transf Aloca_Entrada_Nao_Terminal
      %inc Maxnt
      %change Current of Tabela_Nt with maxnt
      %change Endereco with Maxnt
    %end Aloca_Entrada_Nao_Terminal

    %transf Atualiza_Simbolo_Nao_Terminal
      %change Tabela_Nt.Simbolo with Simbolo
    %end Atualiza_Simbolo_Nao_Terminal

    %transf Atualiza_Prim_Trans
      %change Tabela_Nt.Prim_Trans with Tabgrafo.Current + 1
    %end Atualiza_Prim_Trans

%end subprocess

```

Conditions	N	N	Y	Y
Endereco > 0				
Tipo	'C'	'N'	'C'	'N'
Actions				
%exec Aloca_Entrada_Nao_Terminal	1	1		
%exec Acerta_Ultimo_No_Sub_anterior	2		1	
%exec Atualiza_Simbolo_Nao_Terminal	3	2	2	1
%exec Atualiza_Prim_Trans	4		3	


```

%subprocess Grava_Tabela_Simbolos
%Main
%selection
%select Tabela_Simbolos
%transformations
%insert Arq_Tabela_Simb with Tabela_Simbolos,
        Arq_Tabela_Simb.Indice with Tabela_Simbolos.Current
%end subprocess

%subprocess Grava_Tabela_Nt
%Main
%selection
%select Tabela_Nt starting_at 1 to Maxnt
%transformations
%insert Arq_Tabela_Nt with Tabela_Nt,
        Arq_Tabela_Nt.Indice with Tabela_Nt.Current
%end subprocess

%subprocess Grava_Tabgrafo
%Main
%selection
%select Tabgrafo starting_at 1 to Maxgrafo
%choice
%decision table Escolhe_Simbolo_para_Tabgrafo
%Conditions
    Tabgrafo.Ter      | Y  Y  Y  N
    Tabgrafo.Sim      | -1 -2 >=0 -
%Actions
    %exec Escolhe_Terminal |      1
    %exec Escolhe_N_Terminal |      1
    %exec Grava_Lambda    | 1
    %exec Grava_Outros    | 1
%end table

%transformations
%transf Escolhe_Terminal
%change Current of Tabela_Simbolos with Tabgrafo.Sim
%insert Arq_Tabela_Gf with Tabgrafo,
        Arq_Tabela_Gf.Indice with Tabgrafo.Current,
        Arq_Tabela_Gf with Tabela_Simbolos
%end Escolhe_Terminal

%transf Escolhe_N_Terminal
%change Current of Tabela_Nt with Tabgrafo.Sim
%insert Arq_Tabela_Gf with Tabgrafo,
        Arq_Tabela_Gf.Indice with Tabgrafo.Current,
        Arq_Tabela_Gf with Tabela_Nt
%end Escolhe_N_Terminal

%transf Grava_Lambda
%insert Arq_Tabela_Gf with Tabgrafo,
        Arq_Tabela_Gf.Indice with Tabgrafo.Current,

```

```
        Arq_Tabela_Gf.Simbolo with 'LAMBDA'
%end Grava_Lambda

%transf Grava_Outros
    %insert Arq_Tabela_Gf with Tabgrafo,
        Arq_Tabela_Gf.Indice with Tabgrafo.Current,
        Arq_Tabela_Gf.Simbolo with 'OUTROS'
%end Grava_Outros

%end subprocess
```

Comentários Finais

Com a parte de definição de conjuntos de valores, as tabelas de decisão podem descrever os problemas com maior facilidade. O pré-processador de tabelas de decisão implementado não faz a verificação da compatibilidade de tipos entre as variáveis de condição e os elementos dos conjuntos de valores das linhas estendidas, em razão de tratar apenas das partes do programa fonte que contêm tabelas de decisão e conseqüentemente não possui informações sobre os tipos das variáveis envolvidas. Como o algoritmo de conversão do pré-processador foi herdado pelo gerador I-M-E, a deficiência permanece, mas essa verificação pode ser implementada colocando-se o tipo dos elementos de cada conjunto de valores declarado na tabela de decisão ou de cada componente dos conjuntos de n-uplas na tabela de símbolos de conjuntos de valores construída pelo gerador durante a análise de uma tabela de decisão. O comando `goto table` que pode ser especificado na parte de descrição das ações de uma tabela de decisão deveria ser abolido, pois ele pode ser substituído pelo comando `exec table` como a última ação da sequência de ações de uma regra. A sua presença complica a análise das especificações o que foi resolvido com o emprego de dois passos, que poderiam ser reduzidos a um só passo em termos de tabelas de decisão. Além disso, dificulta a geração de código, pois ela provoca o aparecimento de vários pontos de encerramento de uma etapa. Também a especificação de fim de uma transformação poderia ser abolida, pois o início da próxima transformação encerra a anterior.

No gerador I-M-E implementado estão faltando pelo menos dois itens importantes que são as especificações para a geração de relatórios e manipulação de arquivos indexados. O LDT [Se-Ma 88] que opera em ambientes IBM de grande porte permite a especificação de relatórios em módulo específico para esta finalidade. Uma outra possível solução que pode ser estudada é apresentar na parte de declarações uma descrição do relatório que então poderia ser tratado como um arquivo de características ligeiramente diferentes dentro dos processos e sub-processos. O nosso gerador produz programas para serem compilados no sistema *Turbo Pascal*, a partir da versão 4.0, mas uma solução mais eficaz seria escolher uma ambiente mais amplo em termos de suporte que permitisse ao gerador simplesmente produzir chamadas a rotinas disponíveis, por exemplo, para o gerenciamento de arquivos indexados e tratasse os números decimais adequadamente como se exige em processamento administrativo. Seria desejável que o ambiente fornecesse mecanismos de produção de telas para serem usadas pelo gerador e não ficar limitadas àquelas que podem ser descritas na parte de declarações ao gerador. Na verdade, esta solução foi adotada para superar a ausência de um gerador de telas integrado ao projeto. À medida que novos problemas são especificados para o gerador surgem novas propostas de especificação, de modo que ele deve gradativamente incorporar outras possibilidades. Elas devem ser compatíveis com o método I-M-E de desenvolvimento de programas e evitar que o excesso de detalhes torne as especificações parecidas com as de uma linguagem de alto nível.

Outras extensões que devem ser incluídas são a possibilidade de associar dinamicamente um arquivo externo a um arquivo lógico durante a execução; permitir que qualquer tecla individualmente seja declarada como de transmissão de tela e não apenas aquelas entre **F1** a **F10** ou **all**. Infelizmente o comando **exchange** está funcionando apenas para variáveis simples ou campos elementares e precisa ser estendido para ter a generalidade semelhante à do comando **change**. Outros comandos de alteração do elemento corrente de uma tabela como *previous*, *bottom*, *top* devem ser incluídos. Outro ponto a melhorar é possibilitar a declaração de sequências não apenas como um só intervalo ou enumeração de todos os seus elementos, mas misturar estas duas formas numa só sequência ou mesmo permitir a declaração de uma sequência por vários intervalos. É claro que nestes casos, teremos que pagar um preço para o armazenamento desses dados no programa objeto e no código para acesso aos elementos das sequências. No caso de arquivos do tipo texto, precisamos permitir a especificação de cadeias constantes de caracteres a serem gravadas em cada registro (para isto, bastaria estender a declaração de arquivos) e talvez fosse interessante permitir a especificação de linhas de formatos diferentes para um mesmo arquivo.

Durante as experiências que desenvolvemos surgiram inúmeros casos em que tivemos que consultar o elemento que fora anteriormente selecionado num arquivo antes do elemento corrente. Talvez seja desejável introduzir um mecanismo automático de acesso a tal elemento, mas é preciso convencionar o valor que ele comportará durante a primeira iteração. Tal valor poderia inclusive ser consultado para detectar se é ou não a primeira iteração.

Outro aspecto importante é que a tentativa de tornar o gerador mais abrangente acaba tornando os problemas particulares em casos especiais e normalmente o código produzido para atender a generalidade é menos eficiente para casos específicos. Talvez seja desejável produzir código ocupando-se de casos especiais para evitar a ineficiência em situações mais comuns. Uma alteração que deve ser implementada é produzir código para processos e sub-processos como procedimentos ou funções, usando-se rótulos somente para tabelas de decisão e transformações. A pilha auxiliar seria global e os comandos de retorno seriam colocados no trecho final de cada procedimento somente para os rótulos locais. Isto permitiria a implementação de declarações locais, chamadas de sub-processos com parâmetros e recursividade.

É muito útil poder testar se a seleção de um conjunto de dados encontrou algum elemento ou não. Para resolver este problema, precisamos especificar uma variável auxiliar para este controle, como está feito no exemplo do carregador do grafo sintático, mas é recomendável que se permitam o acesso a variáveis automáticas com esta finalidade e a execução de uma transformação ou sub-processo no caso de um comando de seleção não selecionar nenhum elemento. Por exemplo, se *A* é o nome de arquivo, **A.not_found** poderia indicar, se verdadeiro, o insucesso da seleção de *A*.

É possível que a maior dificuldade na utilização do gerador seja a falta de um sistema integrado de edição, compilação e depuração com rastreamento ("trace"). É preciso utilizar um editor para especificar o programa, abandoná-lo temporariamente para a compilação, retornar ao editor, carregar o arquivo que contém as mensagens de erros e corrigi-los no arquivo de entrada que não é o mesmo em que eles são apontados. Esta intensa troca entre os arquivos de trabalho pode acabar provocando situações desagradáveis, como verificar após exaustivas correções que elas foram efetuadas no arquivo errado. Em algumas situações o trabalho de especificar na forma exigida pelo gerador é extenso e repetitivo. Se considerarmos que o volume de erros de digitação é de algum modo proporcional ao tamanho do texto, talvez seja prudente

estabelecer algumas formas de abreviar o arquivo de entrada, ou desenvolver um editor específico para orientar o trabalho de especificação do usuário.

Algumas experiências com o gerador permitem afirmar que pelo menos para problemas com soluções não complexas porém tediosas como o desenvolvimento de programas que envolvem entrada de dados por telas com o controle do teclado através do programa pode ser feito pelo gerador. O método I-M-E e o gerador têm a intenção de permitir ao usuário especificar os seus problemas, à medida do possível, dentro do universo da aplicação, eliminando uma parte dos detalhes de mais baixo nível que são inerentes ao computador e não aos problemas tratados [Se 89]. No exemplo das telas, cabe ao usuário descrever as características dos campos, não precisando preocupar-se com a forma como o computador obterá cada caractere do teclado, a movimentação do cursor no vídeo e o tratamento que deverá ser dado a cada caractere de modo que o campo resultante satisfaça as propriedades desejadas. No caso de arquivos e tabelas, o usuário não precisa fazer a verificação do seu final e do limite superior do seu índice, respectivamente.

A geração automática das iterações proporciona uma maior simplicidade nas especificações e a maneira como o gerador trata as estruturas de dados desobriga o usuário de declarar variáveis temporárias tais como "buffers" para leitura de dados dos arquivos e índices das tabelas.

As especificações ao gerador devem estar estruturadas de acordo com o método I-M-E de desenvolvimento de programas, o que força uma estruturação para os problemas e propicia uma forma mais sistemática de trabalho para o usuário.

Apêndice

O Gerador I-M-E foi implementado usando o analisador sintático desenvolvido no curso regular de Compiladores, conforme descrito em [Se-Me 83]. Em seguida, apresentamos o grafo sintático onde os nós retangulares contêm símbolos não terminais e os demais contêm símbolos terminais. Entre os símbolos terminais estão os caracteres especiais, palavras com letras minúsculas e “I”, “M”, “E”, “S”, “C” e “T” que são as iniciais das etapas e fases do método I-M-E. Os demais símbolos terminais (primeira letra maiúscula, e λ que significa “ignorar símbolo de entrada”) indicam que não é o próprio símbolo que está sendo esperado pelo analisador, mas uma palavra que satisfaz uma particular característica. Na tabela a seguir estão listados estes símbolos terminais e as respectivas cadeias aceitas pelo analisador.

Ident	Identificador
Number	Número inteiro
Real	Número real
Alpha	Cadeia de caracteres delimitada por apóstrofos
Eof	Fim de arquivo
Row	Número inteiro (linha do vídeo)
Column	Número inteiro (coluna do vídeo)
Decimal	Número inteiro (Número de decimais de um número real)
Length	Número inteiro (Tamanho de um campo)
F1-F10	Cadeia da forma ‘F1’, ‘F2’, ..., ‘F10’.
Other	Qualquer símbolo exceto <i>Eof</i> (Fim de arquivo)
λ	Cadeia vazia
Lf	Fim de linha (Linefeed)
Separator	Um caracter especial, normalmente ‘ ’.

No sub-grafo de ‘Condition’ existe um símbolo ‘-¹’ que representa a entrada de condição indiferente (dont’care) e é denotado desta forma para diferenciar do ‘menos’ unário que precede um valor. O símbolo λ é utilizado para executar alguma rotina especial, como no sub-grafo de ‘Condition’ onde no vértice que contém este símbolo é executada uma rotina semântica que verifica se logo após o símbolo ‘-’ existe ou não um espaço em branco, pois isto identifica para os vértices seguintes se o símbolo deve ser interpretado como uma entrada de condição indiferente ou como ‘menos’ unário.

Em vários pontos do grafo existe o símbolo *Lf* quando o analisador espera um ‘fim de linha’. Para isto, o gerador mantém uma variável global que normalmente informa ao analisador léxico [Se-Me 83] que não deve ser passado o símbolo *Lf*, mas que é alterado para fazê-lo em pontos em que é necessário.

Símbolo Inicial

Part

Screen

Dcl

Selection

Choice

Condition

Action

Prep

Transf

Set

Field

Init

Insert-Cmd

TTransf

Change-Cmd

Remove-Blanks

Cmd-Lg

Expr

Union

Sub-Expr

Id-List

Term

Tuple

Difference

Intersection

Complement

Relational

```

graph TD
    Root(( )) --> Pct[%]
    Root --> Ident1[Ident]
    Root --> Num[Number]
    Root --> Real[Real]
    Root --> Alpha[Alpha]
    
    Pct --> Max[max]
    Pct --> Min[min]
    Pct --> Total[total]
    Pct --> Avg[average]
    Pct --> Count[count]
    
    Ident1 --> LP1[(]
    Ident1 --> Dot[.]
    Ident1 --> LS1[[]
    
    LP1 --> Val1[Value]
    Val1 --> Comma1[,]
    Comma1 --> RP1[)]
    RP1 --> Root
    
    Dot --> Curr[current]
    Dot --> Char[char]
    Dot --> Ident2[Ident]
    
    LS1 --> Val2[Value]
    Val2 --> Comma2[,]
    Comma2 --> RS1[)]
    RS1 --> Root
  
```

Figure 1 consists of two block diagrams, (a) and (b). Diagram (a) shows an input signal entering a block labeled 'I'. The output of 'I' splits into two parallel paths: one through a block labeled 'Lf' and another through a block labeled 'Other'. The outputs of 'Lf' and 'Other' are then combined. Diagram (b) shows an input signal entering a block labeled 'transformations'. The output of 'transformations' splits into two parallel paths: one through a block labeled 'transf' and another through a block labeled 'T'. The outputs of 'transf' and 'T' are then combined.

Também não estão no grafo, para simplicidade, as várias palavras reservadas que podem ser utilizadas como sinônimos. Por exemplo, nos pontos do grafo onde está o símbolo **T**, deveríamos ter em mente o subgrafo adicional (b) mostrado na figura anterior. Segue, abaixo, uma lista

desses sinônimos (vide também 4.5):

Initialization	I, Init
Process	Proc
Subprocess	Subproc
Main	M
Ending	E
Selection	S
Choice	C
Transformations	T, Transf
Variable	Var
Integer	Int
Constant	Const
Real	Float

Bibliografia

- [Ah-Ho-Ul 74] Aho, A.V., Hopcroft, J.E. & Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison Wesley (1974).
- [Ah-Se-Ul 86] Aho, A.V., Sethi, R., Ullman, J. D. *Compilers Principles, Techniques, and Tools*. Addison-Wesley (1986).
- [Ah-Ul 72] Aho, A.V. & Ullman, J.D. *The Theory of Parsing, translation and compiling*. Englewood Cliffs, Prentice Hall (1972-1973) Vol. I.
- [Ar 71] Arnold, Henry O. *Utilization of a Decision Table Translator for Basic Program Code*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 12-19.
- [Ba 73] Bayes, A.J. *A Dynamic Programming Algorithm to Optimise Decision Table Code*. The Australian Computer Jornal, Vol. 5, Num. 2 (May,1973) 77-79.
- [Bi-Ra 87] Biswas S. & Rajaraman, V. *An Algorithm to decide feasibility of linear integer constraints occurring in decision tables*. IEEE Transactions on Software Engineering, Vol. 13, Num. 12 (December,1987) 1340-1347.
- [Bj 68] Bjork, Harry. *Decision Tables in ALGOL 60*. BIT Vol.8, Num. 3 (1968) 147-153.
- [Bo-Mu 77] Bondy, J.A. & Murty, U.S.R. *Graph Theory with Applications*. The MacMillan Press Ltd (1977).
- [Br-Re 75a] Breitbart, Yuri & Reiter, Allen. *Algorithms for Fast Evaluation of Boolean Expressions*. Acta Informatica, Vol. 4 (1975) 107-116.
- [Br-Re 75b] Breitbart, Yuri & Reiter, Allen. *A Branch-and-Bound Algorithm to Obtain an Optimal Evaluation Tree for Monotonic Boolean Functions*. Acta Informatica, Vol. 4 (1975) 311-319.
- [Ca 74] Cavouras, John C. *On the Conversion of Programs to Decision Tables*. Comm.Acm, Vol. 17, Num. 8 (August,1974) 456-462.
- [Ca-Ch 67] Callahan, Michael D. & Chapman, Anson E. *Description of Basic Algorithm in DETAB/65 Preprocessor*. Comm.ACM, Vol.10, Num. 7 (July,1967) 441-446.
- [Ch 67] Chapin, Ned. *Parsing of Decision Tables*. Comm. ACM, Vol. 10, Num. 8 (August,1967) 507-510.
- [Ch 70] Chase, Phillip J. *Algorithm 382 - Combinations of M out of N objects*. Comm. ACM, Vol. 13, Num. 6 (June,1970) 368-368.

- [Ch 79] Chinnaswamy, V. *Translation of Decision Tables into Computer Programs with the use of Carnaugh Maps*. Sigplan Notices, Vol. 14, Num. 5 (May,1979) 21-23.
- [Ch 83] Chvalovsky, V. *Decision Tables*. Software - Practice and Experience, Vol. 13, Num. 5 (May,1983) 423-429.
- [Ch-Ra 76] Cheng, Cheng-When & Rabin, Jonas. *Synthesis of Decision Rules*. Comm. Acm, Vol. 19, Num. 7 (July,1976) 404-406.
- [Co 70] Codd, E.F. *A Relational Model of Data for Large Shared Data Banks*. Comm. ACM, Vol. 13, Num. 6 (June,1970) (377-387).
- [Da 71] Dailey, W. H. *Some Notes on Processing Limited-Entry Decision Tables*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 81-89.
- [Da 72] Dathe, Gert. *Conversion of Decision Table by Rule Mask Method without Rule Mask*. Comm. ACM, Vol. 15, Num. 19 (October,1972) 906-909.
- [Da 75] Dailey, W. H. *On Generating Binary Decision Trees with minimum nodes*. Sigplan Notices, Vol. 10, Num. 12 (December,1975) 14-21.
- [Da-We 73] Davies, G & Welland, R. *A pre-processor using Rule mask Techniques for Extended Entry Decision Tables*. Software - Practice and Experience, Vol. 3, Num. 3 (July-September,1973) 227-234.
- [Di 70] Dial, Robert B. *Algorithm 394. Decision Table Translation*. Comm. ACM, Vol. 13, Num. 9 (September,1970) 571-572.
- [DI 89] Dlugosz, John M. *Logic Gem v. 1.0*. Computer Languages, Vol. 6, Num. 5 (May,1989) 131-132.
- [Ga 72] Garey, M.R. *Optimal Binary Identification Procedures*. SIAM Journal on Applied Mathematics, Vol. 23, Num. 2 (September,1972) 173-186.
- [Ga-Jo 79] Garey, Michael R. & Johnson, David S. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York (1979).
- [Ga-Ra 73] Ganapaty, S. & Rajaraman, V. *Information Theory Applied to the Conversion of Decision Tables to Computer Programs*. Comm. ACM, Vol. 16, Num. 9 (September,1973) 532-539.
- [Ha 71] Harrison, William J. *Practically Complete Decision Tables*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 89-93.
- [Hu 73] Humby, E. *Program from Decision Tables*. Macdonald, London - American Elsevier, New York (1973).
- [Hu 83] Hurley, Richard B. *Decision Tables in Software Engineering*. Van Nostrand Reinhold Company, N.Y. (1983).
- [Hy-Ri 76] Hyfil, Laurent & Rivest, Ronald L. *Constructing optimal binary decision trees is NP-Complete*. Information Processing Letters, Vol. 5, Num. 1 (May,1976) 15-17.

- [Ib-Ra 78] Ibramsha, M. & Rajaraman, V. *Detection of Logical Errors in Decision Table Programs*. Comm. ACM, Vol. 21, Num. 12 (December,1978) 1016-1025.
- [Ja 75] Jackson, M.A. *Principles of Program Design*. Academic Press,Inc (1975).
- [Ki 65] Kirk, H.W. *Use of Decision Tables in Computer Programming*. Comm. ACM, Vol 8, Num. 1, (January,1965) 41-43.
- [Ki 66] King, P.J.H. *Conversion of Decision Tables to Computer Programs by Rule Mask Techniques*. Comm.ACM. Vol 9 Num 11, (November,1966) 796-801
- [Ki 68] King, P.J.H. *Ambiguity in Limited Entry Decision Tables*. Comm. ACM, Vol.11, Num.10, (October,1968) 680-684.
- [Ki-Jo 73] King, P.J.H. & Johnson, R.G. *Some Comments on the Use of Ambiguous Decision Tables and Their Conversion to Computer Programs*. Comm.ACM, Vol.16, Num.5, (May,1973) 287-290.
- [Ki-Jo 74] King, P.J.H. & Johnson, R.G. *Comments on the Algorithms of Verhelst for the Conversion of Limited-Entry Decision Tables to Flowcharts*. Comm. ACM, Vol.17, Num:1, (January,1974) 43-44. *Reply of Verheslt.* pag. 45.
- [Ki-Jo 75] King, P.J.H. & Johnson, R.G. *The Conversion of Decision Tables to Sequential Testing Procedures*. The Computer Journal, Vol. 18, Num. 4 (November,1975) 298-306.
- [Ku-Ch 83] Kulikov, M. Ya. & Chervenchuk, V. D. *Correcting the Description of Control Algorithms by Limited Entry Decision Tables*. Cybernetics, Vol. 19, Num. 1 (January-February,1983) 22-28.
- [Ku-Ch 84] Kulikov, M. Ya. & Chervenchuk, V. D. *The Optimization of Decision Tables*. Cybernetics, Vol. 20, Num. 2 (March-April,1984) 204-211.
- [Le 71] Lemoine, Donald J. *An Automatic Decision-Logic-Table Processor*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 40-45.
- [Le 78] Lew, Art. *Optimal Conversion of Extended-Entry Decision Tables with General Cost Criteria*. Comm.ACM, Vol. 21, Num. 4 (April,1978) 269-279.
- [Le 82a] Lesk, Michael. *English for Decision Tables Considered Harmful*. Sigplan Notices, Vol. 17, Num. 7 (July,1982) 88-88.
- [Le 82b] Lew, Art, *On the Emulation of Flow Charts by Decision Tables*. Comm. ACM, Vol. 25, Num. 12 (December,1982) 895-905.
- [Le 83] Lew, Art. *Decision Tables for General Purpose - Scientific Programming Software*. Software - Practice and Experience. Vol. 13, Num. 2 (February,1983) 181-188.
- [Lo 72] London, Keith R. *Decisions Tables*. Auerbach Publishers Inc (1972).
- [Lu-Si-Ko 79] Luchesi, Cláudio & Simon, Imre & Simon, Istvan & Simon, Janos & Kowaltowski, Tomasz. *Aspectos Teóricos da Computação*. Impa - Instituto de Matemática Pura e Aplicada, Rio de Janeiro (1979).

- [Ma 78] Maes, R. *On the representation of program structures by decision tables: a critical assessment*. Computer Journal. Vol. 21, Num. 4 (November,1978) 290-295.
- [Ma-Mo 78] Martelli, Alberto & Montanari, Ugo. *Optimizing Decision Trees Through Heuristically Guided Search*. Comm.ACM, Vol. 21, Num. 12 (December,1978) 1025-1039.
- [Mc 68] McDaniel, Herman. *An Introduction to Decision Logic Tables*. John Wiley & Sons, Inc., New York (1968).
- [Mc 85] McCarthy, Jim. *Logic at a Glance*. Computer Languages, Vol. 2, Num. 7 (July,1985) 35-40.
- [Mc 86] McCarthy, Jim. *Decision Tables and Logic Processing*. Computer Languages, Vol. 3, Num. 11 (November,1986). 73-80.
- [Mc 89] McCarthy, Jim. *Sowing Seeds of Productivity*. Computer Languages, Vol. 6, Num. 6 (June,1989). 34-43.
- [Me-Ba 77] Metzner, John R. & Barnes, Bruce H. *Decision Table Languages and Systems*. Academic Press, Inc. N.Y. (1977).
- [Mi 65] Miller, Raymond E. *Switching Theory. Volume I: Combinatorial Circuits*. John Wiley & Sons, Inc., New York (1965).
- [Mi 77] Milner, D. A. *The Use of Decision Table Logic in a Manufacturing System*. International Journal of Production Research. Vol. 15, Num. 1 (January,1977) 17-26.
- [Mi 85] Miyakawa, Masahiro. *Optimum Decision Trees - An Optimal Variable Theorem and its Related Applications*. Acta Informatica, Vol. 22 (1985) 475-498.
- [Mi 89] Miyakawa, Masahiro. *Criteria for Selecting a Variable in the Construction of Efficient Decision Trees*. IEEE Transactions on Computers, Vol. 38, Num. 1 (January,1989) 130-141.
- [Mi-Ch 89] Misra, A.K. & Chaudhary, B.D. *Modified Structured Decision Table and its Complexity*. Sigplan Notices. Vol.24, Num.6 (June,1989) 32-34.
- [Mo 74] Montalbano, Michael. *Decision Tables*. Science Research Associates, Inc. (1974).
- [Mo 82] Moret, Bernard M.E. *Decision Trees and Diagrams*. Computing Surveys, Vol. 14, Num. 4 (December,1982) 593-623.
- [Mo-Hi 72] Moriya, Shinji & Hiramatsu, Keiji. *Table Program and its Conversion to Computer Program - An Extension of Decision Table*. Information Processing in Japan. Vol. 12 (1972) 51-57.
- [Mo-Th-Go 80] Moret, Bernard M.E. & Thomason, M.G. & Gonzalez, R.C. *The Activity of a Variable and Its Relation to Decision Trees*. ACM Transactions on Programming Languages and Systems, Vol. 2, Num. 4 (October,1980) 580-595.

- [Mu 84] McMullen, B. *Structured Decision Tables*. Sigplan Notices. Vol. 19, Num. 4 (April,1984) 34-43.
- [Mu-Ra 70] Muthukrishnan, C.R. & Rajaraman, V. *On the Conversion of Decision Tables to Computer Programs*. Comm. ACM, Vol.13, Num. 6 (June,1970) 347-351.
- [Ni-Mc 86] Nixon, J. S. & McMullen, W. L. *Decision Tables in Process Control: A Powerfull Development Tool*. ISA (Instrument Society of America) Transactions, Vol. 25, Num. 4 (1986) 55-85.
- [Oe 68] Oerter, G.W. *A New Implementation of Decision Tables for a Process Control Language*. IEEE Transactions on Industrial Electronics and Control Instrumentation. Vol.15, Num.2, (December,1968) 57-61
- [Po 65] Pollack, Solomon L. *Conversion of Limited-Entry Decision Tables to Computer Programs*. Comm. ACM, Vol.8, Num.11, (November,1965) 677-682.
- [Po 71] Pollack, Solomon L. *Comment on the Conversion of Decision Tables to Computer Programs*. Comm. ACM, Vol. 14, Num. 1 (January,1971) 52-52.
- [Po 74] Pooch, Udo W. *Translation of Decision Tables*. Computing Surveys, Vol. 6, Num. 2, (June,1974) 125-151.
- [Po-Hi-Ha 71] Pollack, Solomon L. & Hicks, Harry T.Jr. & Harrison, W. J. *Decision Tables:Theory and Prattice*. John Wiley & Sons. New York (1971).
- [Pr 65] Press, Laurence I. *Conversion of Decision Tables to Computer Programs*. Comm. ACM, Vol.8, Num.6 (June,1965) 385-390.
- [Ra 71] Rabin, Jonas. *Conversion of Limited-Entry Decision Tables in Optimal Decision Trees: Fundamental Concepts*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 68-74.
- [Ra 87] Rajaraman, V. *Validation of Decision Tables Used in Process Control*. IEEE Transactions on Industrial Electronics, Vol. 34, Num. 2 (May,1987) 168-171.
- [Re-Sa-Ya 87] Reilly, Kevin D. & Salah, Akram & Yang, Chao-Chih. *A Logic Programming perspective on decision table - theory and practice*. Data & Knowledge Engineering, Vol. 2, Num. 3 (September,1987) 191-212.
- [Re-So 66] Reinwald, Lewis T. & Soland, Richard M. *Conversion of Limited-Entry Decision Tables to Optimal Computer Programs I : Minimim Average Processing Time*. Journal of the ACM, Vol. 13, Num. 3 (July,1966) 339-358.
- [Re-So 67] Reinwald, Lewis T. & Soland, Richard M. *Conversion of Limited-Entry Decision Tables to Optimal Computer Programs II : Minimum Storage Requirement*. Journal of the ACM, Vol.14, Num.4, (October,1967) 742-755.
- [Sc 85] Schneider, M. L. *Weighted Decision Tables - An alternative Solution for Ambiguity*. The Computer Journal, Vol. 28, Num. 4 (August,1985) 366-371.
- [Sc-Se 76] Schumacher, Helmut & Sevcik, Kenneth C. *The Synthetic Approach to Decision Table Conversion*. Comm. ACM, Vol. 19, Num. 6 (June,1976) 343-351.

- [Se 87] Setzer, Valdemar W. *Banco de Dados*. Editora Edgard Blücher Ltda, Segunda edição (1987).
- [Se 88] Setzer, Valdemar W. *Um Sistema Simples para Documentação Semi-automático de Programas*. Relatório Técnico RT-MAC-8808. Departamento de Ciência da Computação. IME-USP. (Setembro,1988).
- [Se 89] Setzer, Valdemar W. *A Miséria da Computação I a VII*. *Jornal de Software*, 4 (Junho,1989) 12; 5 (Julho,1989) 18; 6 (Agosto,1989) 13; 7 (Setembro,1989) 15; 8 (Outubro,1989) 13; 10 (Dezembro,1989) 10 e 11 (Janeiro,1990) 12.
- [Se-Ch 80] Sethi, I.K. & Chatterjee, B. *Conversion of Decision Tables to Efficient Sequential Testing Procedures*. *Comm. ACM*, Vol. 23, Num. 5 (May,1980) 279-285.
- [Se-Ma 88] Setzer, V.W. & Marussi, Ervino. *LDT: Um Gerador Universal de Aplicações para Processamento de Dados*. Relatório Técnico RT-MAC-8086. Departamento de Ciência da Computação. IME-USP (Junho,1988).
- [Se-Me 83] Setzer, V.W. & Melo, I.S.H. *A Construção de um Compilador*. Rio de Janeiro, Campus (1983).
- [Sh 71] Shwayder, Keith. *Conversion of Limited-Entry Decision Tables to Computer Programs - A Proposed Modification to Pollack's Algorithm*. *Comm. ACM*, Vol.14, Num.2, (February,1971) 69-73.
- [Sh 74] Shwayder, Keith. *Extending the Information Theory to Converting Limited-Entry Decision Tables to Computer Programs*. *Comm. ACM*, Vol.17, Num.9 (September,1974) 532-537.
- [Sh 75] Shwayder, Keith. *Combining Decision Rules in a Decision Table*. *Comm. ACM*, Vol.18, Num. 8 (August,1975) 476-480.
- [Si 71] Silberg, Bruce. *Detab/65 in third - Generation Cobol*. *Sigplan Notices*, Vol. 6, Num. 8 (September,1971) 4-8.
- [Sm-Sh 75] Smillie, K.W. & Shave, M.J.R. *Converting Decision Tables to Computer Programs*. *Computer Journal*, Vol. 18, Num. 2 (May,1975) 108-111.
- [Sp 66] Sprague, V.G. *On Storage Space of Decision Tables*. *Comm. ACM*, Letter to the Editor, Vol 9, Num. 5 (May,1966) 319-319.
- [St 71] Sterbenz, Robert F. *Tabsol Decision Table Preprocessor*. *Sigplan Notices*, Vol. 6, Num. 8 (September,1971) 33-40.
- [St 73] Strunz, Horst. *The Development of Decision Tables via Parsing of Complex Decision Situations*. *Comm. ACM*, Vol. 16, Num. 6 (June,1973) 366-369.
- [Ta 76] Takaoka, Hiroo. *Tabelas de Decisão e sua Implementação*. Dissertação de Mestrado. FEA-USP (1976).
- [Te 82] Terada, Routo. *Desenvolvimento de Algoritmos e Complexidade de Computação*. Terceira Escola de Computação. Departamento de Informática, PUC/RJ (1982).

- [Va 85] Vanthienen, J. *A note on English for Decision Tables Considered Harmful and the nested "if-then-else"*. Sigplan Notices, Vol. 20, Num. 5 (May,1985) 45-47.
- [Ve 66a] Veinott, C.G. *Programming Decision Tables in FORTRAN, COBOL or ALGOL*. Comm. ACM, Vol.9, Num. 1, (January 1966) 31-35.
- [Ve 66b] Veinott, Cyril G. *More on Programming Decision Tables*. Comm. ACM, Letters to the Editor. Vol.9, Num. 7, (July 1966) 485-485.
- [Ve 72] Verhelst, M. *The Conversion of Limited-entry Decision Tables to Optimal and near-optimal Flowcharts:Two new Algorithms*. Comm. ACM, Vol.15, Num. 11, (November 1972) 974-980.
- [Vi 79] Vivian, Maria E. Bruno. *Sobre a Eliminação de Recursão em Programas*. Dissertação de Mestrado. IME-USP (1979).
- [We 71] Weisbard, Michael F. *DETAP/55: A Decision Table Preprocessor for Generating Single-Paragraph, Fully-Nested Cobol Code*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 45-53.
- [Wi 76] Wirth, Niklaus. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, N.J. (1976).
- [Wo-Ha 71] Woods, C. Geoffrey & Hawes, Mary K. *Optimized code Generation from Extended-Entry Decision Tables*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 74-80.
- [Ya 71] Yasui, Toshio. *Some Aspects of Decision Table Conversion techniques*. Sigplan Notices, Vol. 6, Num. 8 (September,1971) 104-111.
- [Za 84] Zaki, Ahmed S. *Conversion of Limited Entry Decision Tables to Computer Programs*. International Journal of Computer Mathematics, Vol. 15, Num. 1 (1984) 3-22.