"PRÉ-PROCESSADOR PARA LINGUAGEM NÃO ITERATIVA EM SISTEMAS CONVERSACIONAIS"

Estevam Gilberto De Simone

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.).

Aprovada por:

Célio C. Guimarães

Pierre-Jean Lavelle

Nelson Maculan Filho

RIO DE JANEIRO

ESTADO DO RIO DE JANEIRO - BRASIL

FEVEREIRO DE 1976

A Célia e ao

Nelson

AGRADECIMENTOS

Ao Prof. Nelson Maculan Filho cuja confiança e incentivo estão na origem deste trabalho.

Ao Prof. Célio Cardoso Guimarães pela excelente orientação recebida.

Aos colegas alunos, professores e funcionários dos Programas de Engenharia de Sistemas e Engenharia Biomédica, em especial ao Prof. Pierre-Jean Lavelle.

À Angela, na datilografia, e ao Marquinho e Roberto, na prancheta, pela dedicação acima de qualquer expectativa.

Aos meus amigos e a Célia, minha companheira, por tudo o que tenho recebido.

À COPPE pelo apoio financeiro no decorrer de todo o trabalho.

Toda manhã, a fim de ganhar a vida,
Vou para o mercado onde se vendem mentiras.
Esperançoso,
Entro na fila dos vendedores.

B. Brecht.

R E S U M O

O presente trabalho estuda o problema de definir uma linguagem de programação para ensino de computação e o processo de compilação a adotar para sua utilização em modo conversacional.

Dada a incompatibilidade entre este tipo de sistema e as principais linguagens de programação é estudado e proposto um compromisso entre as restrições impostas à linguagem e as restrições impostas ao sistema.

A partir da crítica de uma linguagem com objetivos semelhantes, a "LSE 15 - Langage Symbolique d'Enseignement", são propostas a sintaxe e a semântica da nova linguagem, que é um descendente do ALGOL 60.

São analisados os principais métodos de adequação de linguagens não iterativas a sistemas conversacionais. Sendo então proposto um pré-processador que fragmenta o texto fonte em trechos e condições tais que permitem sua compilação através de um compilador "batch" ligeiramente modificado. O conjunto de caracteres mantidos sob vigilância do pré-processador é minimizado bem como sua intercomunicação com o compilador e a necessidade de recompilação de trechos já admitidos. Para esses efeitos o pré-processador armazena os fragmentos em estrutura em árvore binária, para a qual são descritos seis algoritmos, que constituem o

núcleo principal do trabalho: geração e percurso na árvore; inserção, eliminação e substituição de fragmentos; recompilação.

A B S T R A C T

This work deals with the problem of defining a programming language oriented toward the teaching of computation and a compiling procedure appropriate for its usage in a conversational mode.

Due to the incompatibility of this kind of system with the most programming languages, a compromisse between restrictions on the language and on the system is proposed.

The proposed syntax and semantics of the new language are suggested by a critical analysis of another language, the "LSE 15 - Langage Symbolique d'Enseignement", created with similar purposes. The new language is a descendent of the ALGOL 60.

The main methods for accomodating non-iterative languages to conversational systems are analysed. This suggests a pre-processor that breaks the source-text into pieces and conditions, thus allowing its compilation by a sligthly modified batch compiler. The set of characters under pre-processor's control, the pre-processor-compiler interaction and the need of recompiling processed pieces are minimized. To achieve this goal the pre-processor stores the pieces in a binary tree structure. The central part of this work are six algorithms for such binary structures, namely: tree generation; traversal; insertion, deletion and substitution of pieces; recompilation.

I N D I C E

CAPÍTULOS PÁGINAS

CAPÍTULO I 1

I.Introdução 1

CAPÍTULO II 6

II.Conceituação do Problema 6

II.1. Linguagens de Programação e Modo Conversacional 7

II.2. Técnicas de Adequação 10

II.3. Análise de Utilização de Sistemas Conversacionais 14

II.4. Linguagens tipo ALGOL e Sistemas Conversacionais 21

CAPÍTULO III 26

III. Linguagem para Ensino de Computação 26

III.1. LSE 15: Liguagem de Comando do Monitor... 26

III.2. Estrutura do Programa 32

III.3. Esquema de Subprogramação 34

III.4. Valores,Tipos,Variáveis e Escopos 36

III.5. Expressões e Comandos Atributivos 39

III.6. Controle da Sequência do Programa 41

III.7. Instruções de Entrada e Saída.... 46

III.8. Inserção de Comentários 46

III.9. Passagem de Parâmetros 47

CAPÍTULOS PÁGINAS

III.10. Analisador Sintático 49

III.11. Admissão de Programas 51

III.12. ALGOL como Linguagem de Ensino de Computação 52

CAPÍTULO IV 65

IV. Pré-Processador para Linguagem não Iterativa em Sistema Conversacional 65

IV.1. Fragmentação do Programa 65

IV.2. Representação da estrutura de Programas em Árvores Binárias 73

IV.3. Edição de Comandos com Fragmentação 75

IV.4. Geração da Árvore Associada e Alterações no Programa 85

IV.5. Rotulação de Comandos 108

IV.6. Edição e Compilação de Fragmentos quais - quer 110

CAPÍTULO V 115

V. Extensões ao Pré-Processador. Conclusões ... 115

V.1. Extensões ao Pré-Processador 115

V.1.1. Criação de Blocos de Comandos já Compilados 115

V.1.2. Reutilização de Comandos em Blocos Eliminados 116

V.1.3. Edição de Caracteres na Linha 119

V.1.4. Reutilização do Texto Fonte 119

V.1.5. Edição de mais de um Comando por Linha... 121

CAPÍTULOS PAGINAS

V.1.6. Tabulação de Comentários 121

V.2. Conclusões 122

BIBLIOGRAFIA - Autores 125

BIBLIOGRAFIA - Linguagens 129

APÊNDICE A - Linguagem de Comando do Monitor LSE 15 131

APÊNDICE B - Árvore Gramatical da Linguagem LSE 15 133

APÊNDICE C - Árvore Gramatical da Linguagem Proposta para ensino de Computação 143

CAPÍTULO I

INTRODUÇÃO

O presente trabalho é uma parcela dos esforços desenvolvidos pelo Programa de Pós-Graduação em Engenharia de Sistemas e Computação da COPPE/UFRJ, para criação de um Laboratório de Ensino de Computação.

Esse projeto tem por objetivo transferir para o aluno de computação as vantagens e as facilidades advindas do desenvolvimento tecnológico recente, tanto em equipamento como em técnicas de programação. A constatação básica do projeto é que, de uma maneira geral nas universidades brasileiras, o aluno de computação não encontra uma estrutura de serviços de computação adequada aos seus interesses de aprendizado.

O desenvolvimento da indústria de minicomputadores e computadores de médio porte, expresso pelo aumento de poder de computação associado ao barateamento de seus custos, bem como o desenvolvimento análogo da indústria de equipamento periférico, vem permitir a montagem de sistemas de tempo compartilhado ("Time-Sharing") baseados numa configuração composta por um minicomputador dotado de sistema de interrupção, com memória auxiliar em disco, uma impressora, opcionalmente uma leitora de cartões e um número variá

vel de terminais de vídeo ou teletipo. Com esse equipamento é possível a elaboração de sistemas para operação em modo conversacional.

É evidente que, em termos de ensino de computação, onde o tempo gasto na execução do programa feito pelo aluno é efetivamente desprezível face ao tempo gasto para concepção, diagnóstico e correção desse mesmo programa, os sistemas conversacionais oferecem vantagens significativas.

Por outro lado, os minicomputadores na configuração acima descrita oferecem suficiente poder de cálculo para suprir as necessidades de processamento administrativo de pequenas universidades e mesmo para suprir as necessidades de cálculo científico, até certo limite.

Esses motivos levaram o Programa de Pós-Graduação em Engenharia de Sistemas e Computação da COPPE/UFRJ a voltar seu interesse para o estudo das características de computação em tempo real, de sistemas operacionais de tempo compartilhado, de linguagens específicas para o ensino de computação e para a compilação em modo conversacional.

Como parte integrante desse projeto global de pesquisa o presente trabalho versará sobre uma proposta de definição de uma linguagem de ensino de computação e dos problemas encontrados em adaptá-la a um sistema conversacional.

A idéia da elaboração de uma nova linguagem com tal objetivo baseou-se na firme convicção que boa parte das dificuldades dos iniciantes na área decorre da grande quantidade de infor

mações forçadamente memorizadas e do excessivo grau de complexidade das linguagens atualmente em uso, em prejuízo da absorção e fixação do reduzido número de conceitos realmente significativos . Essas falhas são consequências, principalmente, da utilização de linguagens projetadas com objetivos radicalmente diversos, associados a péssimas implementações.

Alguns dos parâmetros do projeto dessa linguagem de ensino, foram prioritariamente estabelecidos:

a) a linguagem deveria favorecer uma ação conjunta e simultânea dos processos de concepção, codificação, documentação e depuração dos programas;

b) a linguagem deveria introduzir o conceito de comando composto, sem necessariamente introduzir o conceito de "estrutura em blocos";

c) a linguagem deveria introduzir os conceitos de subprogramação e recursividade;

d) a linguagem deveria ser compatível com o conceito de "programação estruturada";

e) a linguagem deveria conter, no mínimo comandos dos tipos "IF...THEN...ELSE...", "FOR...UNTIL...", "WHILE...DO..." e "GO TO...", limitando por outro lado a variedade dos demais comandos;

f) as expressões aritméticas deveriam ser expressas na forma mais usual e encontradiça em linguagens de programação;

g) os comandos deveriam ser escritos em forma semelhante à frase com significado equivalente em português;

h) seria conveniente que a linguagem dispusesse de comandos para operações alfanuméricas.

A partir deste corpo mínimo de requisitos,que configuram a linguagem proposta como um descendente bastante simplificado do ALGOL 60 [*Naur,63*], partiu-se para a definição sintática e semântica da linguagem e para o projeto de seu compilador ou analisador sintático e interpretador, dentro de um sistema conversacional.

Entretanto, a utilização de uma linguagem não iterativa em sistema conversacional constitui um problema ainda não satisfatoriamente resolvido [*Lorho,72*]. A descrição dos métodos e soluções aplicadas a este problema, tanto sua resolução particular para a linguagem em questão como o estudo de sua aplicação geral em linguagens com "estrutura em blocos", constitui o corpo deste trabalho.

Toda a pesquisa necessária se desenvolveu no Laboratório de Automação e Simulação de Sistemas da COPPE/UFRJ, cujo equipamento consta de:

a) minicomputador CII - MITRA 15, com 64 Kbytes de memória principal;

b) unidade de disco magnético, cabeça móvel, para 2,3 Mbytes;

c) leitora de cartões;

d) terminais de vídeo e teletipos.

CAPÍTULO II

CONCEITUAÇÃO DO PROBLEMA

A utilização de técnicas de tempo compartilhado deu origem a radical transformação no processo de comunicação entre usuários e sistemas computacionais, denominada geralmente modo conversacional.

Nesse processo de comunicação a principal característica alterada foi a necessidade de submeter ao sistema a totalidade de um programa - ou, na melhor das hipóteses, partes muito bem definidas de um programa - e, a seguir, aguardar a totalidade das respostas do sistema, sejam elas diagnósticos ou resultados.

Em modo conversacional é permitido ao usuário apresentar parte de seu programa, recebendo imediatamente as respostas referentes ao conjunto de partes já apresentadas. É também permitido eliminar ou substituir partes, inserir novas partes entre as já existentes, alterar sua ordem ou, simplesmente, dar prosseguimento ao processo de apresentação.

Forçosamente, modificação de tal profundidade na interação usuário-sistema veio acarretar severas mudanças na concepção, projeto e implementação de sistemas de computação, incluíndo as linguagens de programação.

II.1. Linguagens de Programação e Modo Conversacional

Uma possível classificação das linguagens de programação [*Cheatham,72*] em sua relação com sistemas conversacionais as separa em dois grupos: linguagens iterativas e não iterativas. É também observado que estas denominações não se baseiam na possibilidade ou frequência de utilização das linguagens em tais sistemas, mas no grau de dificuldade e nas restrições que se impõem a operação dos processos de edição, compilação, carga e execução em módulos não necessariamente distintos. Trata-se, portanto, de um critério de adequação ao tipo do sistema.

Apenas para fixar as denominações acima, que serão doravante utilizadas, citamos a classificação do mesmo autor para algumas das mais populares linguagens de programação:

a) linguagens não iterativas: FORTRAN, COBOL, ALGOL 60, ALGOL 68, PL/I, PASCAL;

b) linguagens iterativas: BASIC, JOSS, APL , PPL, LISP.

No apêndice bibliográfico poderão ser encontradas referências para cada linguagem citada.

É conveniente que procuremos conceituar melhor a distinção entre as duas classes de linguagens.

Com alguma dose de simplicidade poder-se-á chamar iterativa à linguagem cujos programas tenham por elemento constitutivo básico linhas, estruturadas segundo regras bem definidas e gozando das seguintes propriedades:

a) as linhas são rotuladas (normalmente numeradas);

b) o conjunto das linhas constitui uma estrutura ordenada de acesso aleatório, cujo índice de ordenação e via de acesso é o rótulo;

c) cada linha possui uma estrutura sintática cuja análise independe das estruturas de quaisquer outras linhas;

d) cada linha possui um conteúdo semântico que independe do conteúdo de qualquer outra linha.

A propriedade de ordenação associada às propriedades de independência conferem às linhas a capacidade de constituir elementos distintos e identificáveis tanto no espaço (em uma massa de cartões) como no tempo (em um sistema conversacional).

Por outro lado, essas mesmas propriedades

escondem debaixo de sua aparente simplicidade um formidável grau de dificuldade para sua realização através de uma linguagem e a incapacidade de certas linguagens, ditas iterativas, atenderem aos requisitos acima é uma clara razão para a existência de técnicas sofisticadas - como a gerência de variáveis durante o tempo de execução, por exemplo - destinadas a suprir tais deficiências.

Com igual dose de simplicidade, poder-se-á chamar não iterativa à linguagem cujos programas têm por elemento constitutivo básico comandos, caracterizados por:

a) não obrigatoriedade de rótulo;

b) o conjunto de comandos constitui uma estrutura de acesso sequencial, definida a partir da própria apresentação espacial ou temporal dos comandos;

c) cada comando possui uma estrutura sintática que depende dos demais comandos do programa;

d) cada comando possui um conteúdo semântico que pode depender do conteúdo dos demais comandos, geralmente antecessores.

Algumas das estruturas típicas das linguagens de programação usuais são claros exemplos de incompatibilidade com

a utilização iterativa: a definição de tipos, valores e dimensões e, de uma maneira geral, todas as "declarações"; a possibilidade de utilizar comandos compostos ou blocos no interior de outros comandos; a divisão do programa em secções com função especifica e, até mesmo, o conceito de subprogramação.

II.2. Técnicas de Adequação de Linguagens não Iterativas a Sistemas Conversacionais

Como as mais conhecidas e utilizadas linguagens de programação são não iterativas, diversas técnicas foram desenvolvidas para adequá-las ao modo conversacional. Para efeito de exposição estas técnicas serão agrupadas em três classes, bastante gerais:

a) Partição do tempo de edição e do tempo de execução

Consiste basicamente, em permitir a edição do programa em partes que não são nem linhas nem comandos, tal como definidos anteriormente, mas no que poderíamos denominar linhas físicas, rotuladas e de acesso aleatório. Estas linhas físicas correspondem a um número pré-determinado de caracteres, geralmente definido a partir das características do terminal, e agrupados conforme o livre desejo do programador. Normalmente, uma gran

de variedade de comandos do editor possibilitam amplas facilidades de edição. Entretanto o processo de compilação só será ativado quando a edição estiver completa e será efetuado de modo não conversacional.

Esta técnica promove a adequação necessária a custa de grande sacrifício das possibilidades de utilização do modo conversacional. A maioria absoluta dos sistemas oferecidos pelos fabricantes são deste tipo, que não constituem, propriamente, o objeto de interesse deste trabalho.

b) Redefinição da linguagem

Consiste, basicamente, em promover as mínimas alterações necessárias na sintaxe e semântica da linguagem não iterativa, com o objetivo último de obter a coincidência entre comandos e linhas e maximizar a independência entre os comandos. A nova linguagem obtida deve refletir da melhor maneira possível as características da linguagem original. Um exemplo clássico de utilização desta técnica é a derivação do BASIC a partir do FORTRAN , ressalvadas sua diversidade de objetivos e suas dessemelhanças. Como teremos oportunidade de observar nos capítulos seguintes, este processo verdadeiramente cirúrgico de transformação pode dar margem ao surgimento de linguagens irreconhecivelmente mutiladas.

c) Compilação incremental

Este método visa proporcionar ao usuário o

máximo das vantagens permitidas pelo modo conversacional, sem alterar a linguagem. Consiste, basicamente, em ativar o processo de compilação ao final de cada trecho do texto fonte (que geralmente irão corresponder as linhas físicas) e considerar esses trechos , os incrementos, como um conjunto de elementos independentes, passíveis de serem criados, modificados ou suprimidos sem levar em conta o contexto onde se inserem. O objeto tratado pelo compilador é, efetivamente, o incremento, sendo gerados os pseudo-códigos e os diagnósticos possíveis com apenas tais informações. Evidentemente, esse tipo de análise sintática não garante a validade do incremento analisado e o pseudo-código gerado será forçosamente incompleto, quando tratarmos de linguagens não iterativas.

Um tal tipo de sistema é organizado, em duas partes distintas: um analisador sintático para os incrementos ("syntax line checker") que verifica sua validade, gera um pseudo-código intermediário e constroi os encadeamentos necessários para situar esse incremento entre os demais; um interpretador, que trata os incrementos na ordem especificada pelo encadeamento e executa o pseudo-código porém agora, e apenas nesse momento, relacionando-o com o conteúdo semântico dos demais incrementos, por exemplo ligando a utilização de uma variável à sua declaração de tipo.

Em consequência a sintaxe da linguagem, admitida pelo analisador sintático, deverá sofrer algumas alterações: por exemplo, é necessário que o analisador aceite a instrução "ELSE..." pois não pode verificar se esta segue a uma instrução condicional "IF...THEN...". Além disso teremos criado um profundo desba

lanceamento entre as cargas de trabalho do analisador sintático e do interpretador [*Lorho,72*]: enquanto o analisador sintático limita-se a uma tarefa de verificação e codificação, o interpretador, além da interpretação propriamente dita, encarrega-se de verificar a validade dos encadeamentos e deve se responsabilizar pela correção semântica do programa. E esse tratamento não é trivial.

É evidente que a compilação incremental é o processo típico de compilação para linguagens iterativas que obedeçam estritamente as condições postuladas anteriormente: quando o incremento corresponde à linha, e estas são sintática e semanticamente independentes, as restrições citadas se anulam.

Das três técnicas apresentadas a compilação incremental é a que mais se aproxima de uma efetiva solução para o uso de linguagens não iterativas em sistemas conversacionais. Entretanto, as desvantagens decorrentes são particularmente críticas quando se pretende adotar tal processo em sistema voltado para o ensino. A simples verificação sintática, por ser incompleta, pode ser prejudicial para o programador principiante, que necessita da maior quantidade possível de diagnósticos, porém focalizados sobre trechos curtos e simples do programa. É extremamente indesejável, que erros sintáticos simples - como um erro de digitação do nome de uma variável - seja detetado simultaneamente a erros de execução causados pela estrutura lógica do programa.

Algumas propostas mais recentes procuram determinar novas técnicas capazes de contornar tais problemas. O interpretador CPL/I [*Blaizot,72*] com o objetivo de apurar as incompa

tibilidades semânticas no interior de uma instrução, propõe o que denomina "compilação mista": os incrementos não mais corresponderiam ao fragmento entrado pelo usuário, mas estes fragmentos seriam armazenados e agrupados pelo compilador até constituir o incremento, que no caso corresponderia ao bloco do PL/I. Cada incremento ou bloco seria portanto formado por um conjunto de fragmentos submetidos e estes fragmentos seriam passíveis de alteração, mas causariam a recompilação de todo o bloco. A grande vantagem obtida ao preço desse acréscimo de trabalho é a possibilidade de efetuar a análise semântica dentro de cada bloco.

Mesmo este tipo de solução, que envolve uma programação sofisticada do compilador e alto custo de processamento, especialmente em memória ocupada pelo compilador e tempo de recompilação, além de necessitar o interpretador, não parece adequada ao problema de compor um simples e compacto sistema conversacional para ensino.

II.3. Análise da Utilização de Sistemas Conversacionais

Neste parágrafo o mesmo problema será atacado a partir de uma perspectiva mais pragmática e, de certa forma, anterior ao julgamento e a escolha da técnica a utilizar. O único critério possível para se decidir até onde sacrificar as potencialidades da linguagem ou do sistema adotado - visto que parece segura a existência de incompatibilidade entre ambos - é através de

uma avaliação correta do que pode ser mais valioso para facilitar e tornar mais produtivo o processamento de programas, tanto no que concerne ao trabalho humano, quanto a tarefa da máquina.

A ênfase principal recairá sobre a simplificação e facilitação do encargo do programador uma vez que o objetivo visado é a utilização do sistema para fins educacionais, onde a carga computacional pode ser considerada irrelevante, dentro de certos limites. Este mesmo argumento permite passar ao largo da já extensa polêmica acêrca da maior ou menor eficiência do uso de sistemas "time-sharing" versus "batch" ou "compiladores" versus "interpretadores" [*Bauer,67*] [*Sackman,68*].

Até o presente o uso de sistemas conversacionais de computação não evoluiu no sentido de obter máxima eficiência a curto prazo. Poder-se-iam citar como causas a mística que envolve o crescimento do poder de cálculo dos computadores [*Martin,70*] e a própria situação - digamos de "inferioridade" - do programador frente ao tempo de resposta da máquina [*Eklind,68*], para justificar porque foram tão superestimadas as possibilidades atuais de uso de tal sistema. A verdade é que encontramos farta bibliografia sobre uso de sistemas conversacionais com interação gráfica ou de voz humana, sistemas de ensino que pretendem substituir totalmente o professor, editores e terminais de vídeo altamente sofisticados ou bancos de dados, que respondem perguntas em linguagens naturais. Mas pouco, ou quase nada, foi realmente criado com o objetivo de facilitar a tarefa dos programadores de hoje.

O que há de realmente prático e utilizável

em sistemas conversacionais para programação é oferecido pelos próprios fabricantes, porém muito aquém do potencial do sistema. A situação atual apresenta, paradoxalmente, complexos sistemas que se propõem tarefas sofisticadas frente a sistemas apenas nominalmente conversacionais.

É necessário, portanto, fixar os pontos essenciais desse modo de utilização e verificar sua utilidade ou conveniência para o trabalho de programação. A filosofia orientadora da maior parte deste trabalho pode ser expressa pelo seguinte conjunto de premissas:

1. Resposta clara e imediata a partes do programa

Isto não implica em eliminar necessariamente o compilador "batch" [*Barron, 71*]. Implica, isto sim, em eliminar o tempo de espera entre duas respostas do sistema, o que talvez se constitua na principal vantagem do modo conversacional.

2. Não há necessidade que estas partes sejam de absoluta livre escolha do programador

Não há sentido em se pretender projetar um sistema onde qualquer programa possa ser dividido em pedaços absolutamente arbitrários, entrados na máquina em qualquer ordem e a

seguir automaticamente agrupados, ordenados, analisados e, ainda mais, executados. Mesmo porque não há necessidade de tanto. O que os programadores desejam é submeter seu programa em ordem, divididos em partes que para eles mesmos sejam conexas e tenham significado, podendo ser relativamente extensas. Essa divisão em fragmentos - digamos "bem comportados" - não é uma distorção de raciocínio com intuito de justificar a simplificação da tarefa da máquina: é uma decorrência natural da maneira do programador apreender seu próprio programa.

3. As alterações que se tornarem necessárias podem custar ao programador algum trabalho.

Este argumento vem criticar frontalmente o uso de sofisticados editores de programas. A possibilidade, por exemplo, do programador fazer voltar o cursor de seu terminal para alguma linha remota e, então, acrescentar no meio da frase a sequência 'BEGIN' é uma comodidade desnecessária, cujo preço é a recompilação de todo o programa. Não podemos perder de vista que acrescentar uma linha em um programa, mesmo sob certas restrições é preferível a reapresentar todo o programa.

4. As alterações que se tornarem necessárias podem ser feitas posteriormente.

Trata-se de um ponto muito importante .

A concepção dos sistemas conversacionais e da maioria das linguagens iterativas prevê a possibilidade de, em se encontrando erro na linha entrante que obrigue a substituição de linha anterior, efetuar a correção imediatamente. A consequência imediata é que nenhuma informação pode ser guardada, sem grande custo, a respeito da situação de cada linha em seu contexto. Teremos oportunidade de observar nos capítulos seguintes como essa facilidade de edição se incompatibiliza com "estruturas em bloco".

Mas a pergunta que cabe neste ponto é porque o programador não pode adiar por alguns instantes - até que certa estrutura seja completamente editada, por exemplo - a efetivação da correção ?

5. O programa é suposto editado em ordem , ressalvadas algumas facilidades de alteração de linhas.

Este ponto é consequência direta do ítem anterior. Se for permitido ao programador alterar linhas em qualquer trecho do programa em qualquer instante, então a ordem de entrada pode ser qualquer. Tal excêntrico estilo de programação não é o usual e muito menos o desejável.

6. Ao programador não se deve exigir esforço algum devido às características do sistema.

Isto significa que, qualquer que seja a solução adotada, o trabalho decorrente deverá ser efetuado pela máquina. Este automatismo preconizado talvez seja a maior fonte de trabalho para o sistema. Mas não faria qualquer sentido criar-se um meio ambiente especial para uma linguagem não iterativa já existente, se isso implicar em maior carga para o usuário que o sistema tradicional.

7. O sistema deve propiciar uma apresentação clara e simplificada dos programas.

Uma condição necessário a um editor para linguagens tipo ALGOL é que possa proporcionar todas as facilidades ao programador para examinar e seguir seu programa, preferivelmente acompanhando a própria importância sintática das frases da linguagem. Algumas técnicas com esse objetivo são bastante conhecidas [*Mckeeman, 65*] [*Scowen,71*]: abertura de novas linhas para comandos, chamadas a subprogramas e declarações; posicionamento de rótulos; ajustamento da margem esquerda conforme a profundidade do bloco; partição dos comandos compostos condicionais; enfim, um conjunto de técnicas objetivando a edição do programa segundo um estilo claro e consistente de apresentação. Este processo (denominado "indentation" em inglês) está se popularizando em português sob o escabroso neologismo "indentação".

8. O sistema deve respeitar a documentação como parte integrante do programa.

Merece citação talvez o mais claro e conciso texto a respeito de documentação de programas [*Hoare,73*]:

> *"A idéia de que documentação é algo que se adicione ao programa depois que estiver em uso parece ser errada por princípio e contraproducente na prática. Em vez disso, a documentação deveria ser encarada como uma parte integral dos processos de programação e codificação. As boas linguagens de programação deveriam encorajar e auxiliar o programador a escrever programas claros e auto-documentados (...). A facilidade com que programas são lidos é imensuravelmente mais importante que a facilidade com que são escritos (...). A facilidade de leitura por parte de seres humanos tem sido desprezada em favor da legibilidade pela máquina e, às vezes, ainda prejudicada para facilitar a abreviatura da escrita, obtida através de uma enorme série de convenções implícitas."*
>
> (Trad. do A.)

Este compromisso do sistema com o conteúdo não diretamente significativo do programa, parece vir justificar a necessidade de se preservar o texto fonte, tal como originalmente

criado pelo programador. O custo de se arquivar devidamente o conjunto de sinais e caracteres extra-linguísticos deverá ser equivalente ao custo de salvar o programa todo, se não for maior. O sistema deverá proporcionar também toda a facilidade para o usuário desenvolver seu próprio estilo de programação e comentário e, nos capítulos seguintes teremos ocasião de precisar até onde esse incentivo pode ser levado.

II.4. Linguagens tipo ALGOL e Sistemas Conversacionais

A utilização de linguagens tipo ALGOL em sistemas conversacionais encontra sua dificuldade principal em definir um critério adequado de fragmentação do texto fonte.

Acompanharemos o problema com os exemplos a seguir. Note-se que qualquer das três disposições são permitidas e admissíveis, representando o mesmo programa.

FRAGMENTAÇÃO 1:

```
if
a > b
then
     begin   j : = j
                    + 1;  m: =
                              m + 2
     end ;
```

FRAGMENTAÇÃO 2:

```
if a > b then begin j : = j + 1 ;  m : = m + 2   end ;
```

FRAGMENTAÇÃO 3:

```
if a > b then
       begin
       j : = j + 1 ;
       m : = m + 2 ;
       end ;
```

Estas três disposições são apenas exemplos dentre a infinidade de fragmentações cabíveis, admissíveis e corretas para este mesmo programa. Trata-se agora de determinar o grau adequado de flexibilidade concedida ao programador, sem tornar impossível a tarefa da máquina.

Em primeiro lugar convém relembrar que a resposta do sistema a admissão de um fragmento não terá maior sentido se esse mesmo fragmento não possuir um mínimo de significado. O primeiro exemplo de fragmentação expõe claramente tal fato:qual a utilidade do diagnóstico do compilador, se possível fosse diagnosticar qualquer coisa, ao receber a sequência ' + 1 ; m:=' ?

Em linguagens tipo ALGOL o comando é a menor unidade sintática com certo grau de significado associado. Embora partes de comandos - como expressões lógicas, por exemplo -

possam ter determinado sentido próprio, é na leitura do conjunto do comando que seu significado encontra plena justificação. Note-se que ao nos referirmos a significado neste ponto, estamos nos referindo ao significado apreendido pelo programador, ao significado visto sob a ótica de quem lê o programa, e não ao conteúdo semântico de certo símbolo para a linguagem.

Nessas condições o segundo exemplo de fragmentação parece ser adequado. Mas não pode ser esquecido que em linguagens com estruturas em blocos o programa em si pode ser um comando único. A possibilidade de alteração em trechos não muito extensos deve ser, portanto, assegurada.

Pode-se formular, então, a questão principal que constituirá o âmago do problema: qual o ponto ótimo de fragmentação, intermediário entre as duas condições opostas, que efetuaria o compromisso entre a excessiva fragmentação, que facilita as modificações mas retira o significado dos diagnósticos e dificulta o processo de compilação, e excessiva compactação, que simplifica o processo de compilação, mas torna penoso o processo de substituição das partes ?

Em primeiro lugar, qualquer que seja o grau de fragmentação escolhido, este deverá depender forçosamente da linguagem. Ainda mais, deverá ser efetivado segundo regras estritas de modo a permitir que o processo seja automático, executado diretamente pela máquina e independente da vontade e, se possível, do conhecimento do usuário.

A partir das considerações acima será propos

ta a existência de um programa fragmentador com as seguintes características principais:

a) este programa será parte intrínseca do editor do sistema, e estará intimamente associado ao compilador;

b) a análise da própria sequência de entrada fornecerá as informações necessárias à fragmentação;

c) o programa, na medida em que necessita , informações sobre a estrutura da linguagem, será particular para cada linguagem;

d) o programa deverá proporcionar ao usuário a possibilidade de alterações do texto já entrado , mas poderá criar determinadas restrições;

e) o programa deverá ser "invisível" para o usuário, ou seja, o programador deverá cumprir a mesma série de ações (apertar a mesma sequência de teclas, por exemplo) que cumpriria caso o fragmentador não existisse;

f) como o programa estará em atividade em momentos particularmente críticos para sistemas de tempo compartilhado deverá necessariamente ser muito simples e eficiente.

No capítulo seguinte será feita a análise de uma linguagem iterativa, a LSE 15 - "Langage Symbolique d'Enseignement" - que é fruto da aplicação da técnica de redefinição da linguagem. A partir da crítica e reformulação dessa linguagem será definida a linguagem para ensino, sobre a qual se construirá o programa fragmentador aqui proposto.

CAPÍTULO III

LINGUAGEM PARA ENSINO DE COMPUTAÇÃO

A linguagem LSE 15 é a versão para o minicomputador MITRA 15 da "Langage Symbolique d'Enseignement", definida pela Ecole Supérieure d'Eletricité com o intuito de criar uma linguagem padrão para a inicialização e sensibilização à Ciência de Computação [*Mitra,73*]. Sua concepção prevê a utilização da linguagem em modo conversacional em todas as fases do trabalho: edição do programa, compilação, depuração, execução e arquivamento em biblioteca. A linguagem está inserida em sistema operacional de tempo compartilhado especificamente destinado a seu uso, dotado de monitor para o atendimento de até 16 terminais e de sistema de arquivos em disco para memória ou dados.

Esse conjunto de especificações torna a LSE 15 excelente matéria-prima para análise e definição de uma linguagem para ensino de computação. Nos parágrafos seguintes teremos oportunidade de criticar detalhadamente cada um dos aspectos desse sistema para, a partir dessa crítica, alterar, reformular e redefinir certas características indesejáveis, bem como aproveitar suas melhores criações.

III.1. LSE 15: Linguagem de Comando do Monitor

A linguagem de comando para o monitor e sistema de arquivos, cuja sintaxe se encontra detalhada no apêndice A, dispõe das seguintes facilidades:

a) comandos para conexão usuário-sistema:

a.1 BONJOUR: início de sessão;

a.2 AU REVOIR: fim de sessão;

a.3 IDENTIFICATION: para controle da contabilidade e proteção de arquivos;

a.4 ABREGER: para abreviar a linguagem de comando;

a.5 IN EXTENSO: para anular o efeito do anterior.

b) para edição e compilação:

b.1 LISTER: impressão do texto fonte;

b.2 PERFORER: perfuração do texto fonte;

b.3 NUMERO: impressão do número das linhas

b.4 EFFACER: supressão de linhas.

c) para a execução:

c.1 EXECUTER: execução do programa;

c.2 CONTINUER: reinício da execução do pro

grama, após a instrução PAUSE da linguagem LSE, ou toque de interrupção (ESC) para o monitor;

c.3 REPRENDRE: reinício da execução em ponto diverso do ponto de parada;

c.4 PAS A PAS: execução com parada após cada linha;

c.5 NORMAL: anula o efeito do anterior.

d) para gestão de arquivos:

d.1 RANGER: carga em disco do pseudo-código do programa corrente;

d.2 APPELER: carga em memória de pseudo-código;

d.3 SUPRIMER: eliminar arquivo de diretório;

d.4 MODIFIER: substituição de programa do diretório pelo programa corrente;

d.5 CATALOGUER: entrada em diretório de área de dados do programa;

d.6 DECODER: decodificação e arquivamento do texto fonte do programa em execução.

d.7 ENCODER: carga em memória do pseudo-código obtido a partir da compilação de texto fonte arquivado.

O funcionamento do sistema operacional do ponto de vista do usuário se dá conforme o esquema da figura 1. Os "estados" citados correspondem às seguintes séries de atividades do sistema:

a) ESTADO DORMENTE: o terminal está aberto ao usuário porém desconectado logicamente do sistema e admite apenas o comando BONJOUR;

b) ESTADO MONITOR: o terminal está aberto ao usuário e o contato será feito com o monitor que se encarregará de convocar o analisador sintático, o interpretador ou suas próprias rotinas, conforme a entrada recebida;

c) ESTADO EXECUÇÃO: o terminal está aberto ao usuário que se comunicará diretamente com seu programa;

d) ESTADO COMPILAÇÃO: o terminal está fechado para o usuário e está sendo efetuada a análise sintática e geração de pseudo-código para a linha ou uma saída sobre o terminal;

e) ESTADO SISTEMA: o terminal está fechado para o usuário e está sendo executado um

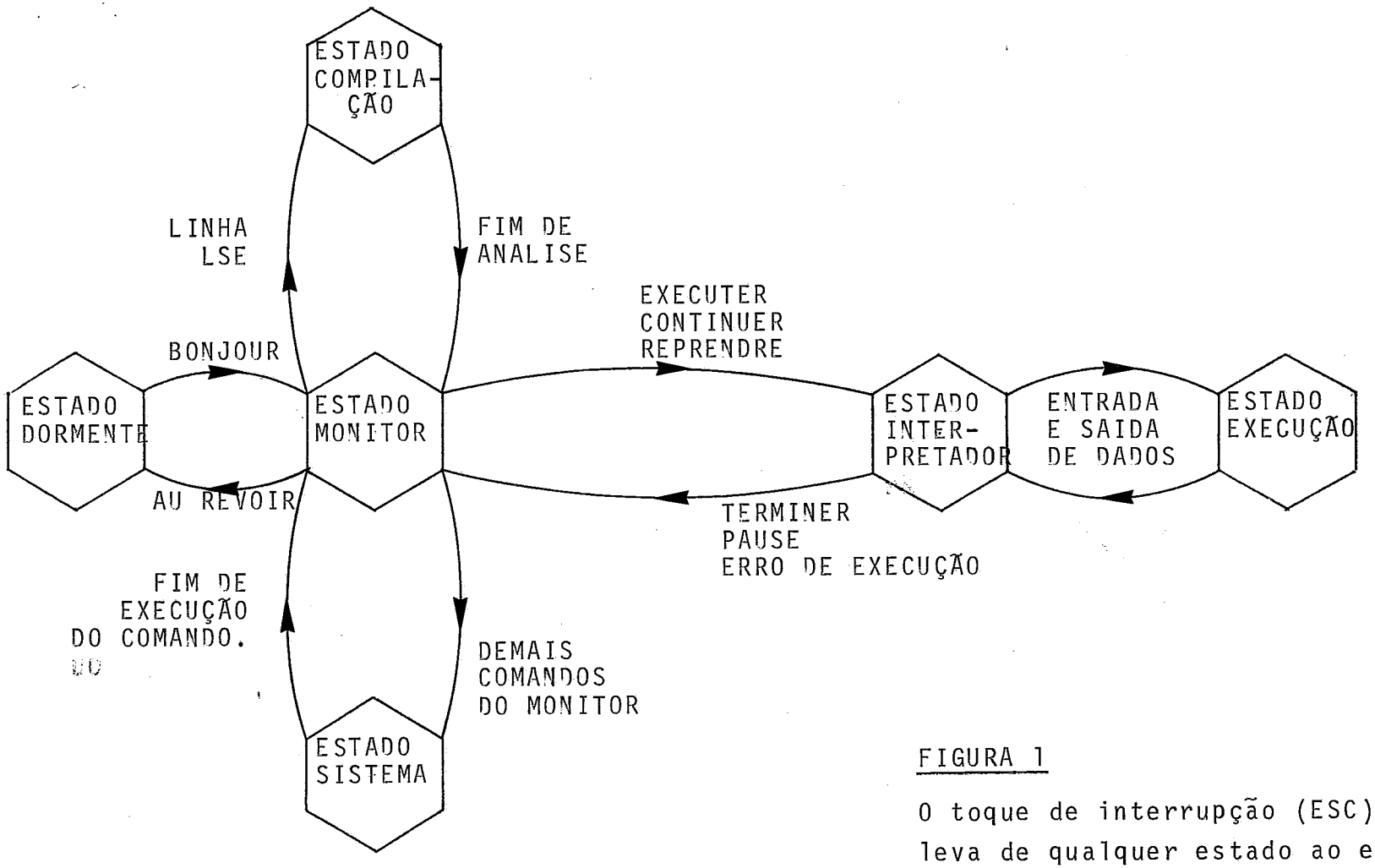


FIGURA 1

O toque de interrupção (ESC) leva de qualquer estado ao estado monitor.

comando do sistema;

f) ESTADO INTERPRETADOR: o terminal está fechado para o usuário e está sendo executado um comando do programa;

O toque de interrupção (ESC) leva sempre ao estado monitor, porém não substitui o comando BONJOUR.

O estado monitor admite ainda o modo "machine de bureau", através do qual o usuário pode efetuar o cálculo de qualquer expressão aritmética simples, substituindo uma máquina de calcular, ou referir-se a alguma variável do programa corrente para afetá-la ou solicitar sua impressão. Este modo não será considerado no presente estudo.

Embora não esteja dentro do escopo deste capítulo, que se voltará exclusivamente para a análise da LSE 15 propriamente dita, é necessário que se constate que o monitor e o sistema de gestão de arquivos suprem comodamente as necessidades previsíveis para atividades de ensino. As facilidades oferecidas para o processo de edição e compilação, a possibilidade de efetuar a execução parcial ou em passos do programa e, especialmente, um sistema de arquivos protegidos e organizados em diretórios prefixados por número de conta e senha, são plenamente adequados para assegurar o suporte adequado aos usuários. Se algumas alterações se tornarem necessárias serão principalmente no sentido de excluir alguns desses comandos do monitor, excessivamente especializados pa

ra a linguagem LSE 15.

E, finalmente, uma observação muito importante: o texto fonte do programa LSE é descartado pelo sistema, que se dá ao trabalho de decodificar o pseudo-código quando é solicitada a listagem de qualquer linha, exceto a entrante. A análise e as consequências desse fato serão discutidas nas secções seguintes.

III.2. LSE 15: Estrutura do Programa

A linguagem LSE 15, cuja descrição sintática completa se encontra no apêndice B, é descendente direta do ALGOL 60. Daquela linguagem guarda a feitura de seus principais comandos, especialmente dos comandos compostos - delimitados por 'DEBUT' e 'FIN' - embora sem designar o escopo das variáveis e rotinas característica da estrutura em blocos do ALGOL.

Nessas condições a linguagem se caracteriza como não iterativa e sua utilização em sistema conversacional apresentou aos projetistas do sistema exatamente o problema analisado neste trabalho, e que mereceu de sua parte uma solução do tipo descrito no capítulo anterior como alteração da sintaxe da linguagem.

A parte da árvore gramatical correspondente a estrutura do programa em LSE 15 - extraída do apêndice B e obedecendo as mesmas convenções descritas no apêndice A - está exposta na figura 2.

|1| <programa> ::=

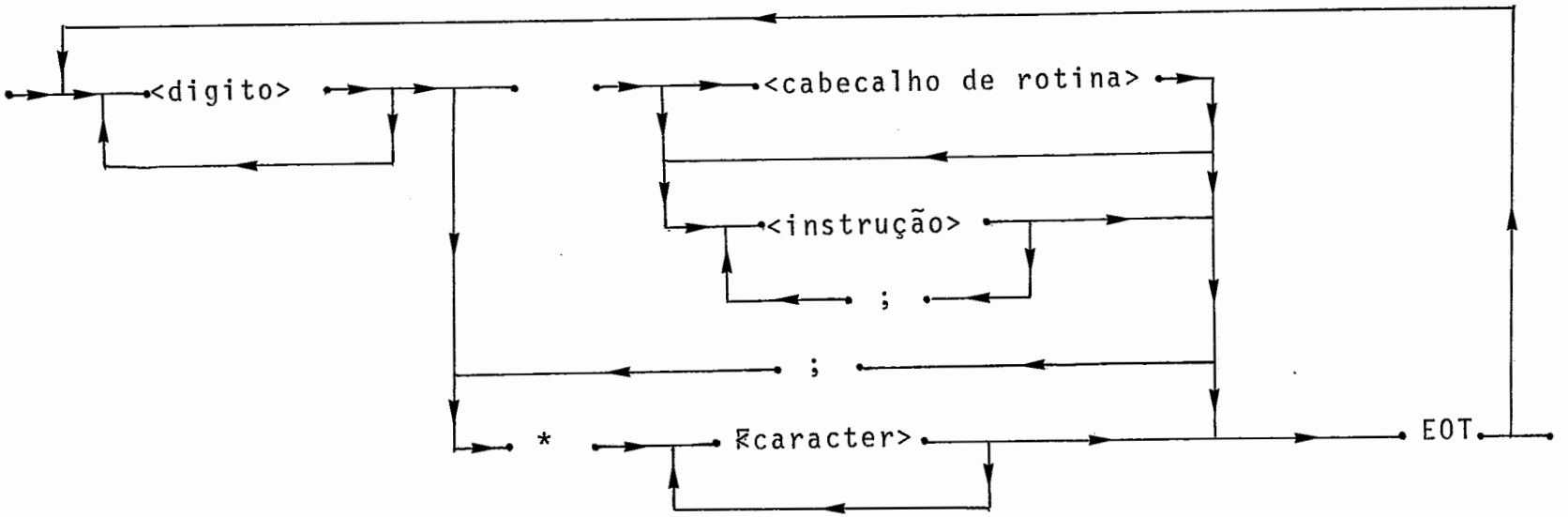


FIGURA 2 Estrutura de um programa em LSE 15.

A primeira é imediata observação a respeito da estrutura do programa LSE 15 é que, ao contrário do ALGOL 60 , sua definição não é recursiva. Cada parte do programa não admite em seu interior partes igualmente construídas. Mais ainda: como a linguagem não possui caracteres para contrôle das linhas do terminal, um programa em LSE 15 é um conjunto de linhas físicas ordenadas segundo seu rótulo numérico. Em termos práticos, isso significa que qualquer instrução, mesmo as que admitem instruções compostas em seu interior, devem conter no máximo 72 caracteres, se for usado um terminal teletipo, por exemplo.

Por outro lado, não existe precedência de apresentação ou de número de rótulo entre as instruções e as declarações. Isto implica em verificação da sintaxe e semântica dos tipos e subscritos das variáveis durante a fase de execução, além de conferir a qualquer declaração escopo global ao programa. A única exceção a esta regra é a declaração de subprograma.

III.3. LSE 15; Esquema de Subprogramação

A obrigatoriedade de escrever comandos compostos em uma única linha física obriga a definir a subprogramação de maneira diversa do ALGOL 60, onde um subprograma é constituído por uma declaração ("procedure heading") seguida de um bloco. Se usada esta forma todas as rotinas em LSE 15 teriam no máximo o comprimento de uma linha física.

Como se pode depreender da árvore gramatical, a solução adotada foi admitir a presença do cabeçalho da rotina em linha separada dos comandos de seu interior. Entretanto essa reorganização - que, em última análise, é um problema sintático sobre a melhor forma de escrever a definição do subprograma - veio acarretar mudanças profundas no conteúdo semântico dos comandos da rotina. Como cada linha física em LSE 15, excetuando-se as declarações, é um comando acessível e executável e não existe qualquer restrição sobre onde colocar a definição do subprograma podemos ter situações dos seguintes tipos: desvios de um ponto do programa para o interior de uma rotina, sem que se efetue a passagem de parâmetros; supressão do cabeçalho da rotina, quando os comandos do subprograma passam a ser comandos do programa principal, e os parâmetros se tornam variáveis; supressão da instrução de saída da rotina, quando os comandos do programa principal de rótulo numérico mais alto se tornam comandos do subprograma.

Na verdade, a atuação do subprograma pode ser dividida em três partes, supondo que esteja sendo corretamente utilizada: o cabeçalho da rotina é responsável pela alteração das tabelas da gerência dinâmica das variáveis (recordar sempre que esta gerência é feita pelo interpretador, em tempo de execução), de modo a adequar os argumentos aos parâmetros; os comandos no interior da rotina são comandos normais, e seu pseudo-código gerado durante a compilação, não é alterado; as instruções 'RETOUR' e 'RESULTAT' modificam as tabelas de variáveis em sentido inverso a entrada na rotina, além de providenciar o retorno adequado.

Apenas como citação o esquema de subprograma ção do LSE 15 admite ainda a declaração de uma variável global no interior de uma rotina, o uso de variáveis locais, a saída "extraordinaire" [*Mitra,72*] de uma rotina, quando o programa continua em ponto diverso do ponto de chamada e a chamada recursiva da própria rotina.

III.4. LSE 15: Valores, Tipos, Variáveis e Escopos

A linguagem admite valores numéricos e alfanuméricos. Embora aceite expressões lógicas no interior de certos comandos os valores lógicos não são definidos.

O único tipo de valor numérico permitido é o tipo real.

As variáveis podem ser variáveis simples ou arranjos de até duas dimensões.

O escopo de uma variável pode ser global ao programa ou local a um subprograma.

Essa combinação de opções feitas para a definição de variáveis em LSE 15, permitiu que a linguagem reduzisse a apenas três as declarações possíveis: a declaração de variável alfanumérica ('CHAINE'); a declaração de variável subscrita ('TABLEAU'); e a declaração de variável local ('LOCAL').

As seguintes regras são automaticamente assu

midas ("default"):

a) uma variável não declarada subscrita é variável simples

b) uma variável não declarada alfanumérica é numérica

c) uma variável não declarada local é global

d) uma variável não declarada é, em consequência das regras anteriores, simples, numérica e global.

Como a verificação semântica da adequação entre a declaração de uma variável e sua utilização é função cumprida pelo interpretador, durante a fase de execução, a verificação sintática não é efetuada durante a fase de compilação. Isto significa que o exemplo abaixo é aceito como correto até o instante da execução da linha 20.

Exemplo:

```
10    TABLEAU  I [3,3];
20    I ← I + 1 ;
```

Outra observação importante a ser feita é

que a linguagem dispõe de comando para desocupação de espaço em memória (LIBERER) que permite eliminar do programa variáveis globais, se o comando for executado no programa principal, ou locais, se for executado no interior de um subprograma. O identificador liberado pode ser reutilizado, pois como vimos é durante a fase de execução que é feita a alocação de memória. Nesse caso o valor anterior é perdido e a variável é reinicializada. Esta possibilidade de reutilização permite, por exemplo, que um mesmo comando tenha, seu significado totalmente alterado dependendo da ordem de execução dos demais comandos.

As diversas situações acima descritas decorrem de uma importante característica da linguagem: as declarações em LSE 15 são, sob certos aspectos, "executáveis". O analisador sintático gera pseudo-código para cada uma delas e apenas quando o interpretador "executa" esse código é que a declaração passa a ter validade. Acompanhemos a situação com o exemplo seguinte:

TEXTO FONTE

```
10   I ← 0 ;
20   A ← 1 ;
30   AFFICHER A ;
40   SI I ≠ 0 ALORS TERMINER ;
50   LIBERER A ;
60   CHAINE A ;
70   A ← 'UM' ;
80   I ← 1 ;
90   ALLER EN 30 ;
```

SAÍDA IMPRESSA

1.0

UM

Note-se que as linhas 20 e 70 utilizam o mesmo identificador para duas variáveis distintas, e a linha 30 tem dois significados. Isso só é possível porque o analisador sintático não verifica a discordância dos tipos entre a variável das linhas 20 e 70. O interpretador ao executar a linha 20, cria área de memória para a variável numérica simples 'A' e na linha 30 imprime o seu valor. Ao executar a linha 50 esta variável é eliminada e outra variável, agora alfanumérica é criada ao "executar" a linha 60. As declarações em LSE 15 são, portanto, "instruções"para a gerência dinâmica de memória.

III.5. LSE 15: Expressões e Comandos Atributivos

A linguagem admite dois tipos de expressões: aritméticas ou literais.

As expressões aritméticas obdecem ao modelo usual de precedência de operadores e permitem utilizar como primários funções internas ou funções definidas pelo usuário (rotinas com o comando 'RESULTAT')e expressões condicionais do tipo
' SI <exp.lógica> ALORS <exp.arit.> SINON <exp.arit.>'.

As dez funções aritméticas internas cobrem perfeitamente as necessidades usuais de cálculo simples.

As expressões literais admitem como operador único o operador concatenação visto que o esquema utilizado para o tratamento de sequências alfanuméricas está baseado principalmente no uso de funções com efeitos colaterais ("side-effects"),em sistema algo semelhante ao PL/I.

Estas funções literais e litero-aritméticas, em número de doze, efetuam tarefas bastante complexas sobre sequências alfanuméricas como obter a posição em uma dada sequência do primeiro caracter não constante de uma subsequência fornecida como parâmetro, iniciando a comparação a partir de certo ponto e terminando a comparação segundo critérios definidos através de parâmetros.

A consequência desta forma de tratamento de sequências que, ao invés de fornecer comandos e estruturas para acesso e alterações na sequência (como ponteiros, comandos de comparação e troca, etc), tenta obter um conjunto de funções com tarefas pré-estabelecidas, é que, na tentativa de prever a maior parte dos casos usuais, se construa um conjunto complexo de funções, dotadas de complicado repertório de parâmetros, com atribuições múltiplas para uma mesma função e obrigando a memorização de inúmeros efeitos colaterais que, em algumas aplicações são o objetivo desejado, em outras, porém, são desnecessários, redundantes e perigosos.

Na LSE 15 temos 12 funções diferentes, variando de um a quatro parâmetros, em 13 esquemas distintos de para

metrização, com resultados numéricos ou literais, conforme o caso, e ainda, dependendo do tipo do parâmetro recebido (variável aritmética ou literal, por exemplo), ou da ausência de certos parâmetros, a função tem comportamento completamente diverso. Nessas condições é praticamente impossível aplicar totalmente toda a potencialidade de tratamento de sequências da LSE 15 sem um período relativamente longo de aprendizado e prática.

O comando atributivo da linguagem atua indiferentemente sobre expressões aritméticas ou literais, efetuando a conferência dos tipos de atributos, e, no caso de variáveis alfanuméricas, providenciando automaticamente a alocação de espaço para a sequência atributo.

III.6. LSE 15: Contrôle da Sequência do Programa

A linguagem dispõe de 8 tipos distintos de instruções para controle da sequência do programa, as quais iremos nos referir por:

1) 'ALLER EN'
2) 'SI...ALORS'
3) 'SI...ALORS...SINON'
4) 'FAIRE...TANT QUE'
5) 'FAIRE...JUSQUA'
6) 'TERMINER'
7) 'PAUSE'
8) 'EXECUTER'

Primeiramente, é necessário lembrar que, embora a linguagem não defina rótulos ou instruções rotuladas, na medida em que cada linha recebe seu número identificador, esse número é usado como apontador para as instruções de contrôle de sequência, referindo-se sempre a única ou a mais esquerda das instruções de uma linha, não sendo as demais acessíveis. Este fato, a primeira vista restritivo, na verdade permite ter acesso a todos os pontos do programa independentes entre si, bastando colocar cada instrução em uma linha. Impede apenas o acesso a uma instrução no interior de um bloco 'DEBUT...FIN', que é um estilo de programação exaustivamente provado indesejável.

A instrução 'ALLER EN' significa algo mais que o polêmico "GO TO" trajado à francesa. Em LSE 15 o desvio é permitido para qualquer linha, exceto a definição de subprograma . Para o interior do subprograma por exemplo. As implicações deste fato foram abordadas no parágrafo sobre subprogramação.

As instruções 'SI...ALORS' e 'SI...ALORS... SINON', definidas de maneira muito poderosa, com expressões lógicas admitindo operadores 'OU' e'ET' e primários alfanuméricos, perde quase totalmente seu potencial de utilização pela obrigatoriedade de comprimir todo o comando em uma única linha. Se necessitarmos utilizar, por exemplo, o comando 'SI...ALORS...SINON' com dois comandos compostos, seremos obrigados a escrever no mínimo 37 caracteres para o rótulo, comando e delimitadores 'DEBUT...FIN'. Em um terminal teletipo restariam 35 caracteres para escrever a expressão lógica e, no mínimo 4 instruções. Outra maneira de visualizar o po

der dessa restrição é notar que é impossível compor dois comandos 'SI...ALORS...SINON' encadeados (permitido pela linguagem e de uso muito comum) por absoluta falta de espaço.

O comando 'FAIRE' tanto na forma 'FAIRE ... TANT QUE' como na forma 'FAIRE...JUSQUA' , da mesma maneira que a definição de subprograma é o ponto nevrálgico e esclarecedor do problema de definir o fragmento básico de um programa em um sistema conversacional. Limitar estes comandos ao comprimento máximo de uma linha significa, praticamente, inutilizar a linguagem. Portanto é necessário sacrificar a clareza e a concisão do comando "FOR" do ALGOL 60, para montar um esquema semelhante ao comando "DO" do FORTRAN, para ganhar "espaço físico" dentro do programa , que, paradoxalmente, é apenas um problema de apresentação. Nova - mente, as dificuldades encontradas na análise sintática vêm acarretar prejuízos na própria construção e significado dos programas.

A figura 5 apresenta a estrutura de funcionamento dos comandos 'FAIRE...TANT QUE' e 'FAIRE...JUSQUA'. Note-se que, embora a grafia do comando nos induza a pensar em duas formas distintas - como o "FOR...UNTIL" e o "WHILE...DO" [Ashcroft, 71] - trata-se de um mesmo e único comando - apenas o tipo do teste varia - sem o poder teoricamente provado das estruturas de contrôle citadas [Peterson,73] [Wulf,71].

As instruções 'PAUSE' e 'TERMINER' comportam-se como seus próprios nomes nos fazem esperar. A instrução ' EXECUTER' entretanto permite fazer o lançamento de um novo programa, carregado a partir da memória auxiliar, mas não permite qualquer

E_0 FAIRE E_1 POUR $V \leftarrow E_2$ PAS E_3 JUSQUA E_4

FOR $I \leftarrow E_2$ STEP E_3 UNTIL E_4 DO B

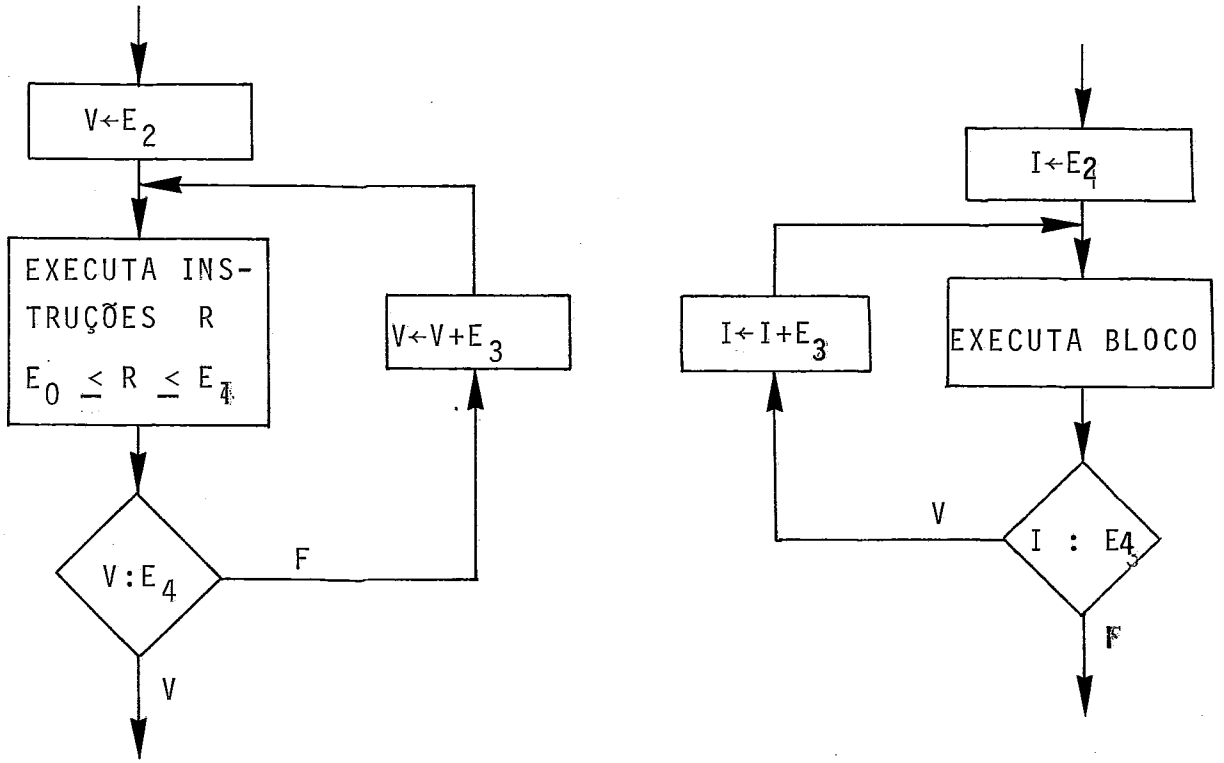
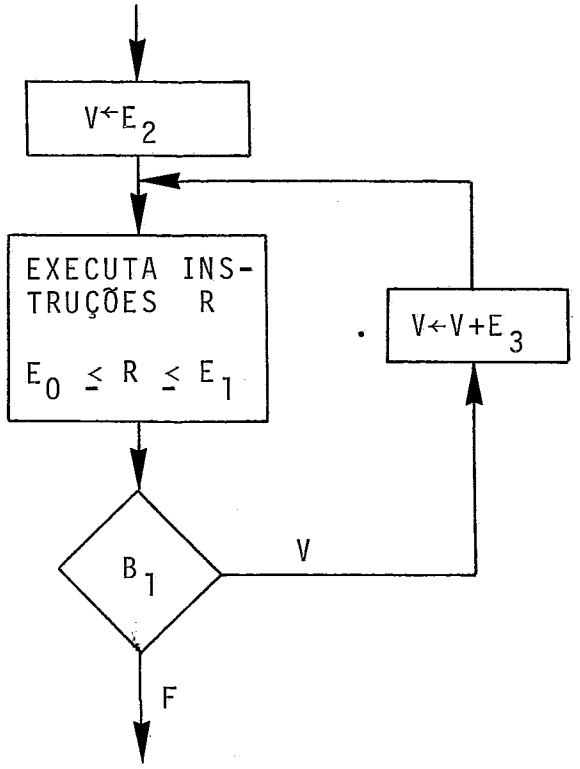


FIGURA 3 Comparação entre "FAIRE...JUSQUA" e "WHILE...DO".

E_0 FAIRE E_1 POUR V E_2
PAS E_3 TANT QUE B_1



WHILE B_1 DO B

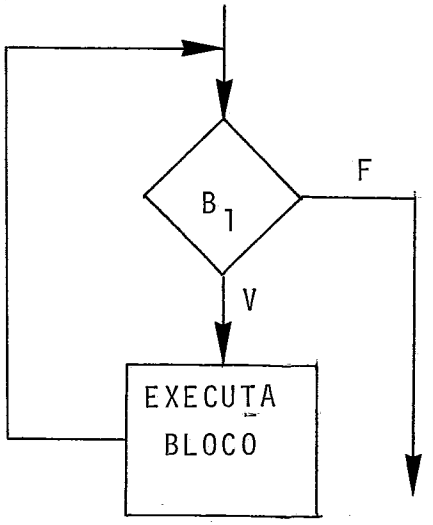


FIGURA 3 (cont.) Comparação entre "FAIRE...TANT QUE" e "WHILE...DO"

tipo de comunicação entre os programas. Todos os dados do programa original são perdidos e sua execução não continua após o final da execução do programa chamado.

III.7. LSE 15: Instruções de Entrada e Saída

As 4 instruções de entrada e saída fornecem eficiente comunicação entre programa e terminal ou programa e arquivos.

A instrução 'LIRE' se encarrega de receber seqüências alfanuméricas ou números em formato livre, efetuando os testes e analisando sintaticamente os caracteres entrados, via terminal.

A instrução 'CHARGER' efetua transferência analoga a partir de arquivo em disco com acesso direto.

As instruções 'AFFICHER' e 'GARER' permitem a saída no console ou em arquivo de acesso direto de dados formatados, segundo um esquema de formatação semelhante ao utilizado nas linguagens de programação populares.

III.8. LSE 15: Inserção de Comentários

Esta inserção é feita utilizando-se o sistema

"fim de linha": após um sinal determinado ('*'), o restante da linha é utilizado para o comentário. Trata-se de um sistema simples e eficiente quando o "restante da linha" tem comprimento maior que alguns poucos caracteres.

A linguagem de comando não dispõe de instrução para tabulação dos comentários, em coluna pré-determinada.

III.9. LSE 15: Passagem de Parâmetros

Chamaremos de parâmetros formais, ou simplesmente parâmetros, os identificadores presentes à definição do subprograma. Chamaremos de argumentos, em vez de parâmetros reais, aos identificadores presentes a uma chamada qualquer do subprograma.

1ª Forma: Passagem de valor ("call by value")

Quando um parâmetro é definido "LOCAL", por ocasião da chamada é efetuado o cálculo do argumento (seja ele numérico ou alfanumérico) e seu resultado é transferido para uma recém criada variável local com nome idêntico ao parâmetro, e o argumento original (mesmo que seja uma simples variável global) não é mais acessível.

Um esquema análogo é utilizado quando o parâmetro é uma variável subscrita local. Nesse caso, os valores transferidos serão os valores da variável subscrita utilizada como argu

mento, sendo vedada a utilização de expressões.

2ª Forma: Passagem de endereço ("call by reference")

Quando o parâmetro é não local, por ocasião da chamada é feita a adequação dos endereços de argumento e parâmetro (mesmo que este seja uma variável subscrita) e as operações descritas no subprograma envolvendo o parâmetro são efetuadas sobre o argumento. Evidentemente os argumentos devem ser variáveis.

3ª Forma: Passagem de nome de subprograma

É possível indicar parâmetros como sendo nome de subprograma. Este nome fictício colocado no parâmetro será substituído pelo nome constante do argumento, por ocasião da chamada. As referências feitas ao subprograma parâmetro no interior da rotina serão substituídas por referências ao subprograma argumento no instante de execução, sendo então efetuada a verificação de seus parâmetros, cuja adequação é responsabilidade do programador.

4ª Forma: Passagem de Rótulos

É possível indicar como parâmetro uma expressão aritmética que servirá como rótulo. Esta "facilidade" se destina a "aumentar o poder"do comando 'RETOUR EN <expressão aritmética>'. Com isto é possível ao programador não apenas retornar do

subprograma em ponto distinto do ponto de chamada, como fazer variar esse ponto de retôrno e, até, calculá-lo !

O exemplo abaixo contém todas as formas significativas possíveis:

```
25 PROCEDURE  &S (H,I) LOCAL H,I ;
27 RESULTAT (EXP(H) - EXP(I))/H
35 PROCEDURE  &C (H,I) LOCAL H,I ;
37 RESULTAT (1/H + 1/I)/H
45 PROCEDURE &ACUM (&F,N,DIF) LOCAL I ;
47 FAIRE 49 POUR I ← 1 PAS 2 JUSQUA N ;
49 DIF ← DIF + &F (I,I-1) ;
51 RETOUR ;
65 * PROGRAMA PRINCIPAL
67 LIRE N ;
69 &ACUM (&S,N,A) ;
71 &ACUM (&C,N,B) ;
73 AFFICHER [F10.6]A-B ;
75 TERMINER
```

III.10. LSE 15: Analisador Sintático

O analisador sintático da linguagem descrita nos parágrafos anteriores deve forçosamente ser muito simplificado.

Isto porque praticamente todas as dificuldades que seriam encontrados no processo de compilação foram previamente contornadas, modificando-se a linguagem. Sumariamente:

a) o fragmento do texto fonte submetido a análise sintática é uma linha física, definida pelo programador, com significado completo dentro das restrições já descritas. O analisador não necessita determinar o início ou o final do incremento, nem guardar informações das linhas já analisadas;

b) as linhas são executadas em ordem de numeração crescente, respeitadas as instruções de contrôle de fluxo, sendo porém armazenadas conforme a ordem de entrada. Uma simples lista sequencial ordenada dos números de cada linha e do endereço de armazenamento dos fragmentos do pseudo-código é suficiente para o trabalho do interpretador;

c) mesmo as declarações geram pseudo-código, composto por instruções para a gerência dinâmica de variáveis, como vimos anteriormente;

d) o processo de análise propriamente dita é facilitado pelo alto grau de redundância e farto uso de separadores, advindos da sintaxe próxima da construção equivalente em francês.

e) na versão operacional do sistema LSE 15 for

necida para o minicomputador MITRA 15, o analisador sintático é um conjunto relativamente pequeno de rotinas, com menos de 6 Kbytes de código objeto (incluindo todas as tabelas) capaz de transformar o texto fonte em pseudo-código, conforme notação polonesa, mas com limitada capacidade de geração de diagnósticos.

III.11. LSE 15: Admissão de Programas

Como aparentemente, foi um dos objetivos do sistema, o processo de edição de programas é efetuado com total simplicidade. A inserção, eliminação ou substituição de linhas se dá pela simples admissão de nova linha:

a) caso o número da nova linha coincida com o número de linha já existente, a nova linha substitui a antiga (se a nova linha estiver em branco teremos, portanto, a eliminação);

b) caso o número da nova linha seja menor que o número de alguma linha já existente, mas distinto dos demais, a nova linha é inserida na posição correta, conforme a ordem crescente dos números.

c) a numeração é de livre escolha do programador sendo, portanto, de sua responsabilidade prever a necessidade de inserção ou providenciar a renumeração.

III.12. ALGOL como Linguagem de Ensino de Computação

No presente parágrafo tentar-se-á isolar os componentes essenciais do ALGOL, visando compor um subconjunto simples e conciso, porém dotado das características principais das linguagens de programação usuais. A escolha de uma linguagem não iterativa para servir de base da análise, se deve ao fato de serem de tal tipo a maioria absoluta das linguagens de programação populares, devendo tal situação prevalecer por ainda algum tempo.

Evidentemente, estes argumentos não justificam a escolha do ALGOL 60. Processo semelhante foi aplicado ao FORTRAN quando da criação do BASIC [*Kemeny,71*]. Processo parecido teve por objeto o PL/I [*Lorho,72*], embora as características desta "ommibus language" [*Cheatham,72*] forçassem a eliminação de boa parte da linguagem por impossibilidade de implementação ou absoluta inaplicabilidade. O próprio FORTRAN já foi alvo de outras tentativas de alteração, visando sua "conversacionalidade" [*Barron,71*]. O próprio ALGOL já foi submetido a experiências diversas, como vimos na LSE 15 ou nas várias implementações de "ALGOL conversacional", oferecidas por fabricantes. Sua escolha pode ser precisamente justificada pela citação seguinte [*Hoare,73*]:

> *"Quanto mais pondero sobre as regras que deveriam nortear o projeto de linguagens e sobre as técnicas que poderiam colocá-las em prá*

tica, maior é minha satisfação e admiração pelo ALGOL 60. Eis uma linguagem tão a frente de sua época que significou um passo adiante não só sobre todas as suas predecessoras, como sobre praticamente todas as suas sucessoras".

(Trad.do A.)

Sem dúvida, alguma margem de indecisão ainda restaria quanto a utilizar PASCAL [*Wirth,71*] em lugar de ALGOL. Foram fundamentais, entretanto, os seguintes argumentos:

a) maior popularidade do ALGOL, evidenciada pela sua aceitação como linguagem "oficial" de alguns fabricantes;

b) maior simplicidade do ALGOL, considerando que algumas características do PASCAL - como a estruturação de tipos - seriam de difícil implementação, além de constituirem um objeto, talvez, sofisticado para uma linguagem de ensino;

c) e, principalmente, o ALGOL apresenta todos os casos possíveis para o estudo da compilação de programas fragmentados em

linguagens com estrutura em blocos.

Dessa forma optou-se por manter o estudo e o projeto dentro de características mais simples e com objetivos, digamos, mais humildes.

Os itens a seguir procuram fazer o inventário das características essenciais das modernas linguagens de programação - a maioria delas introduzidas pelo próprio ALGOL - e hoje indispensáveis a bagagem de qualquer programador. Uma linguagem, porém, é mais que um simples conjunto de instruções e todos os cuidados serão tomados visando formar um todo conexo e coerente, embora simples e conciso.

A LSE 15, criticada nos parágrafos anteriores, será tomada como linguagem - mãe, principalmente por possuir as principais simplificações propostas para o ALGOL 60. Embora algumas alterações necessárias a LSE 15 sejam de tal porte que seu resultado vêm constituir efetivamente uma nova linguagem, apenas as adições ou supressões serão descritas, exclusivamente com o intuito de evitar a repetição tediosa dos elementos comuns.

1. Estrutura do programa. Recursividade.

Mais importante que a utilização efetiva da recursão como ferramenta de programação, deve ser considerada a familiarização do programador com tal instrumento. A linguagem deve admitir níveis limitados de recursão em sua estrutura geral não sendo necessário que sua definição seja plenamente recursiva, tal co

mo o bloco do ALGOL 60. O custo de admissão, de entidades locais a blocos - sejam variáveis, rótulos ou subprogramas - seria excessivamente oneroso para um sistema que se propõe ser simples e deseja ser mais demonstrativo que poderosamente operacional. Portanto, é conveniente que o usuário entre em contato com uma estrutura em blocos simplificada, onde sejam permitidos comandos e proibidas declarações ou definições de subprogramas. É conveniente, também que para os subprogramas seja admitida uma estrutura em blocos mais completa, com definição de entidades locais. A figura 4 apresenta, resumidamente, uma proposta dessa espécie e a figura 5 apresenta, esquematicamente, um exemplo de programa típico.

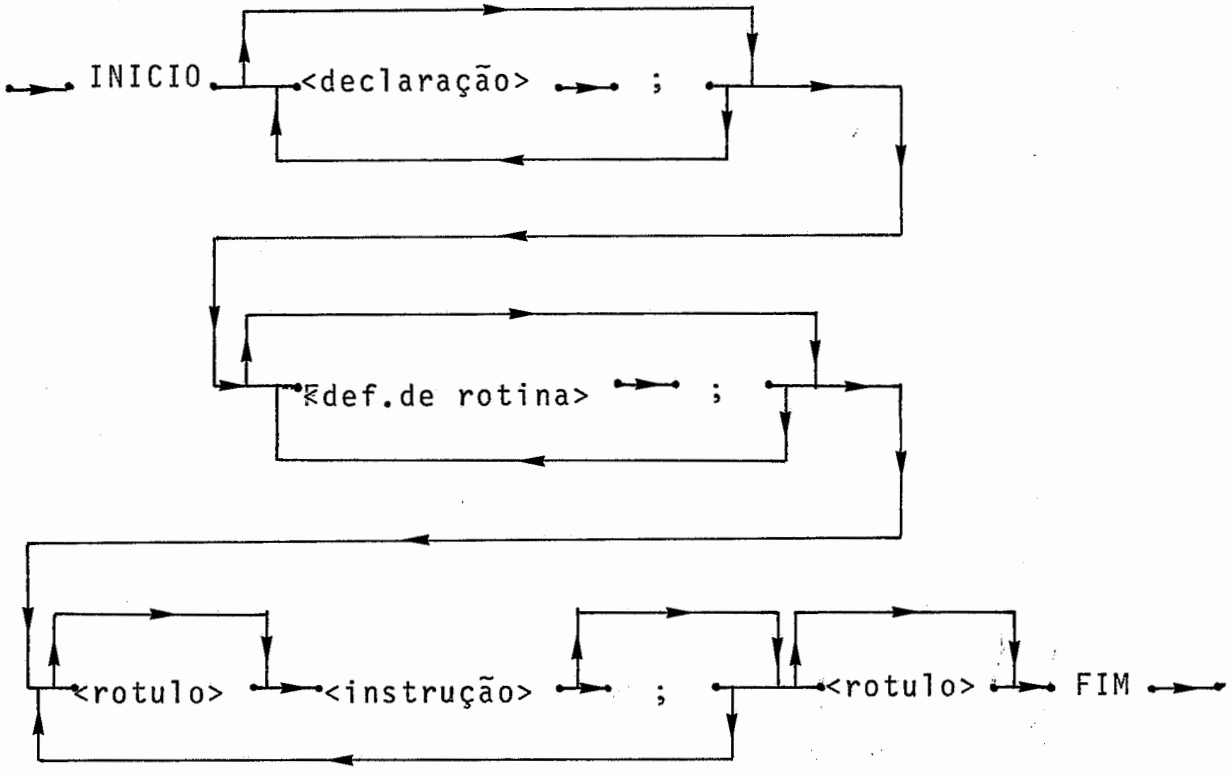
A estrutura dos programas da LSE 15 fica, portanto, totalmente substituída. Note-se que é exatamente neste ponto que os principais problemas de implementação se propõem: a definição da estrutura fundamental da linguagem é tipicamente não iterativa e o uso de blocos completos em subprogramas vem tornar a execução dos comandos não sequencial .

2. Valores, tipos, variáveis e escopos.

Com algumas modificações será mantida a escolha de tipo e valores da LSE 15. Embora não existam valores alfanuméricos na linguagem de referência do ALGOL 60, sua presença foi acrescida à grande maioria das implementações. Não pode faltar porém uma definição de valores lógicos, fundamentais para as moder - nas técnicas de programação, especialmente para evitar o uso do comando "GO TO". Isso implica em acrescentarmos uma declaração para

FIGURA 4 Estrutura de programa proposta para a linguagem de ensino de computação.

|1| <programa> ::=



```
INICIO
DECLARAÇÃO A;
DECLARAÇÃO B;
:
DECLARAÇÃO N;
ROTINA A ( )
        INICIO
        DECLARAÇÃO AA;
        :
        DECLARAÇÃO AN;
        COMANDO A1;
        :
        COMANDO AN;
        FIM;
:
ROTINA N ( )
        INICIO
        DECLARAÇÃO BA;
        :
        DECLARAÇÃO BN;
        COMANDO B1;
        :
        COMANDO BN;
        FIM;
COMANDO 1;
COMANDO 2;
:
COMANDO N;
FIM
```

FIGURA 5 Exemplo esquemático de programa com a estrutura proposta para a linguagem.

variáveis lógicas, assim como alterações no comando atribuição e nas regras de construção das expressões lógicas. O apêndice C onde a nova linguagem está definida apresenta as alterações.

A utilização de um único tipo - 'REAL' - para as variáveis numéricas pode ser encarada como uma simplificação bastante forte, porém aceitável para uma linguagem de ensino que , em primeira aproximação, não se propõe aprofundar o problema de definição de tipos. A eficiência do tratamento do cálculo em ponto flutuante nos modernos equipamentos permite assegurar desempenho satisfatório.

A presença do tipo 'MATRIZ' ("array", "tableau") é uma constante em todas as linguagens de programação não especiais e sua limitação a duas dimensões pode ser considerada mais como fator de implementação que como propriedade definitiva da linguagem.

Optou-se por não admitir qualquer declaração implícita de variável, tal como originalmente no ALGOL 60. A economia de declarações compensada pela fartura de convenções implícitas, do ponto de vista do programador é de uma eficiência, no mínimo discutível. Mas, do ponto de vista de uma linguagem com estrutura em blocos, permitir que uma variável seja implicitamente declarada a partir de um bloco mais interno é um contra-senso. Portanto , definiu-se o seguinte conjunto de regras:

a) quanto a escopos

1. Globais, quando declarados no bloco mais ex

terno.

2. Locais a rotinas, quando declaradas no corpo do subprograma

b) quanto a tipos

1. Alfanuméricos, lógicos, reais ou matrizes quando explicitamente declarados.

Note-se que os tipos matriz, alfanumérico e lógico são mutuamente exclusivos.

O comando 'LIBERER' da LSE 15 é de utilização perigosa e deve preferivelmente ser abolido. É normal que o programador principiante obtenha muitas vezes más soluções para seus problemas, e não possa executá-las por falta de espaço. Esse impedimento é uma penalidade normal e a falta de espaço é apenas um dado do problema, que não deve ser contornado fornecendo-lhe uma ferramenta delicada como o compartilhamento de memória, usada com enorme cuidado - e não menos dificuldades - por programadores experientes. Se o sistema como um todo, em determinada implementação, tem problemas de falta de espaço, a alocação dinâmica de memória deve ser automática e de sua exclusiva responsabilidade, não sendo visível ou acessível a partir de uma linguagem com objetivos de ensino. Note-se que o programador já dispõe de certo grau de flexibilidade através do uso de variáveis locais.

3. Expressões e comandos atribuitivos.

As expressões aritméticas, tal como definidas na LSE 15, abrangem o domínio usual de utilização.

As expressões literais, além de contar com o operador concatenação, devem ter reduzida a variedade e simplificada a parametrização das funções que operam sobre sequências alfanuméricas. Um jogo reduzido de funções desse tipo, pode ser proposto da seguinte forma:

a) Função "varredura"(APTR): é uma função litero-aritmética, que recebe como parâmetro uma sequência fonte, uma sequência procurada e um valor inicial e devolve o valor numérico da posição do primeiro caracter na sequência fonte da subsequência idêntica a sequência procurada. Exemplificando:

Se: SF ← 'VALOR_DESEJADO'
SP ← '_'
temos: APTR (SF,SP,1) = 6

Não sendo encontrada subsequência idêntica a função retorna valor zero, visto que o comprimento das sequências é tratado automaticamente pelo sistema, não sendo as vezes conhecido pelo programador.

b) Função "partição"(SUBS): é uma função li

teral que recebendo como parâmetros uma sequência fonte, uma posição inicial e uma posição final fornece a subsequên - cia compreendida entre estas posições , inclusive. Exemplo:

Se : SF ← 'ISTO_E_UMA_SEQUÊNCIA'

N ← APTR (SF, '_', 1)

temos: SUBS (SF,N,APTR(SF,'_',N+1)='_E_'

Estas duas funções associadas ao operador concatenação devem permitir a grande maioria das operações mais comuns sobre sequências. Na verdade a função APTR é versão simplificada da idéia de apontador ("pointer"). Foi evitada, todavia, a introdução da linguagem de mais essa entidade pelos mesmos motivos que redundaram na escolha dos tipos e visando, também, manter mínimo o número de declarações diferentes possíveis. O apêndice C apresenta a sintaxe completa da nova estrutura das expressões literais.

Com a introdução dos valores lógicos, o comando atribuição passa a admitir nova forma, também detalhada no apêndice C.

4. Instruções

Ficam mantidas as instruções de desvio definidas na LSE 15, do tipo "GO TO" e"IF...THEN...ELSE" do ALGOL. O comando "go to" fica provisoriamente definido na forma geral "GO TO <rótulo >" e, se o esquema de edição dos programas, segundo sua

nova estrutura, exigir que "<rótulo>" seja algo diverso do número da linha, tal como na LSE 15, cabe a ele resolver os problemas surgidos, respeitando o princípio de não sobrecarregar o usuário.

Os comandos iterativos das formas "FOR... UNTIL" e "WHILE...DO" encontram vasta bibliografia em seu apoio e devem ser adotados conforme a semântica proposta na figura 5 e sintaxe definida no apêndice C.

O comando "PAUSE" é indispensável a qualquer sistema de tempo compartilhado. O comando "EXECUTER" tal como definido na LSE 15, pode ser eliminado, visto que sua função é análoga ao comando "EXECUTER" da linguagem de comando do sistema.

As instruções de entrada e saída do LSE 15, atendem perfeitamente as necessidades.

5. Esquema de subprogramação. Parâmetros.

A subprogramação e passagem de parâmetros são mantidas, exatamente como na LSE 15, exceto por uma restrição: não são permitidos desvios para seu interior e nem desvios de seu interior para qualquer ponto externo. As rotinas, tal como na LSE 15, admitem valor intrínseco porém, a forma sintática 'RESULTAT' é indesejável, pois na nova estrutura de programa não poderia ser utilizada em uma linha qualquer. O mesmo resultado pode ser obtido com as formas:

a) ROTINA

b) FUNÇÃO REAL

c) FUNÇÃO LÓGICA

d) FUNÇÃO LITERAL

O valor fica então atribuído a uma variável global de mesmo nome que a função.

A passagem de rótulos como parâmetros bem como o comando 'RETOUR EN' são eliminados.

6. Comentários

O comentário em "fim de linha" é sem dúvida o mais seguro e prático sistema de documentação. Evita o perigoso uso de sinais especiais no fim do comentário, ou o uso de convenções muito longas (por exemplo, "COMMENT") ou, anula, o desperdício de toda uma linha [*Hoare,73*]. O comando tabulação na linguagem do sistema, além de imprescindível, deve ser usado como forte incentivo automático para o programador principiante. Por exemplo, detetando o editor o fim de um comando (';') antes da coluna tabulada, é feito o salto imediato e impresso o sinal convencionado. Caso o programador não deseje comentários nessa linha, um sinal especial será necessário para obter nova linha. O sistema completo, dentro desse espírito está detalhado no capítulo seguinte.

O resumo de alterações descrito neste capítulo vem constituir, associado a definição do LSE 15, o esqueleto de uma linguagem de programação voltada para o ensino de computação, via sistemas de tempo compartilhado. Uma proposta de gramática pa-

ra tal linguagem compõe o apêndice C. Não se pretende com isso definir completa e absolutamente tal linguagem. O objetivo é, basicamente, compor um conjunto coerente de proposições, da forma mais realista e adequada às finalidades de ensino capaz de servir de base para a análise da interelação eficiente dos sistemas conversacionais com a grande maioria das linguagens, projetados sem considerar tal caso.

Na verdade, as críticas e as idéias para alteração do que foi proposto neste capítulo são extremamente desejáveis.

CAPÍTULO IV

PRÉ-PROCESSADOR PARA LINGUAGEM NÃO ITERATIVA EM SISTEMA CONVERSACIONAL

As designações "pré-processador" ou "editor" para o programa descrito neste capítulo são utilizadas apenas para maior clareza da apresentação. Correspondem, na verdade, a parcelas do processo de compilação de programas submetidos em fragmentos. De uma maneira geral, será suposta a existência de um interpretador como depositário e executor da saída gerada por este sistema. Entretanto, uma das consequências principais do processo adotado é que este pode ser adaptado a uma compilação usual, gerando código objeto em vez de pseudo-código. A linguagem não iterativa tomada como referência está descrita no apêndice C e no Capítulo III.

IV.1. Fragmentação do Programa

As regras de fragmentação expostas a seguir são sugeridas pela maneira usual de escrever e apreender um programa em ALGOL. Um dos objetivos visados foi também, manter sob vigilância do editor a menor quantidade possível de caracteres.

Será associado ao processo de fragmentação um esquema de numeração dos fragmentos. Este processo destina-se apenas

a servir de referência e ligação entre as figuras e o texto, não fazendo parte, como posteriormente será descrito, do pré-processador. A numeração se fará mediante uma enupla de números inteiros associada a cada fragmento. Estas enuplas de comprimento variável, utilizam o número de elementos de cada uma como fator adicional de distinção. A enupla $(a_1.a_2...a_n)$ é igual a enupla $(b_1.b_2...b_m)$ se e somente se:

a) $m = n$;

b) $a_i = b_i$, para todo $1 \leq i \leq m$

Vamos distinguir imediatamente fragmentos e linhas. Define-se como fragmento qualquer trecho de programa entre dois quaisquer dos símbolos: "INICIO", "FIM" ou " ; " . Chamar-se-á linha à linha física do terminal, visível pelo usuário. O pré-processador fará a adequação entre fragmentos e linhas saltando para o começo de nova linha ao encontrar qualquer dos três símbolos mantidos sob vigilância. Entretanto, em alguns casos especiais citados posteriormente, o usuário pode mudar de linha dentro do mesmo fragmento. Um fragmento pode, portanto, corresponder a uma ou mais linhas, normalmente apenas uma. Nos exemplos deste capítulo cada fragmento corresponderá a uma única linha, de modo que a numeração será usada indistintamente para ambos.

As regras, a seguir, referenciam os símbolos da linguagem " ; ", "INICIO" e "FIM". Para maior facilidade do pro-

cesso de edição pode ser convencionado desde já que o símbolo, "INICIO" será representado por um único caracter ' : ', bem como o símbolo "FIM" será representado por um único caracter ' . '. O editor pode se encarregar de substituir automaticamente os caracteres ' :' e ' . ' pelas sequências 'INICIO' e 'FIM', respectivamente. Esta providência evitará a presença de analisador léxico no editor. Entende-se, também, por "linha aberta" a linha do terminal onde estão correntemente sendo editados caracteres. Por oposição, "linha fechada " é qualquer trecho do programa, normalmente um fragmento, que tenha sido anteriormente editado em uma única linha.

As regras de fragmentação e numeração são as seguintes:

a) todo caracter não referenciado por estas regras é colocado ao lado direito de seus predecessores, no fragmento corrente;

b) o primeiro caracter do programa - forçosamente ' : ' - compõe o fragmento (0);

c) após cada símbolo ' : ' colocado na linha $(a_1.a_2...a_m.0)$ onde $m \geq 0$, é aberta nova linha e inicia-se novo fragmento denominado $(a_1.a_2...a_m.1)$;

d) o símbolo ' : ' surgido enquanto o frag

mento, $(a_1.a_2...a_n)$ estiver aberto, encerra este fragmento, fecha a linha correspondente, abre novo fragmento e nova linha denominados $(a_1.a_2...a_n.0)$, onde será colocada a sequência 'INICIO';

e) após o símbolo ' ; ' colocado no fragmento $(a_1.a_2...a_n)$ são abertos novo fragmento e nova linha denominados $(a_1.a_2...a_n + 1)$, exceto no caso previsto na regra g;

f) o símbolo ' . ' surgido em qualquer posição, exceto a primeira, de um fragmento $(a_1.a_2...a_n)$ fecha este fragmento e a linha correspondente, abre novo fragmento e nova linha, onde é colocada a sequência 'FIM', e denominada $(a_1.a_2...a_n+ 1)$;

g) todos os caracteres, exceto ' ; ', surgidos após o símbolo ' . ' ou a sequência 'FIM' no fragmento $(a_1.a_2...a_n)$, fecham este fragmento e são colocados em novos fragmentos e linha demominados $(a_1.a_2...a_{n-1}+1)$. O símbolo ' ; ', porém, quando surgido após o símbolo ' : ', ou a sequência 'FIM' no fragmento $(a_1.a_2...$

a_n), é mantido nesse fragmento e abre novos fragmento e linha com o nome ($a_1 . a_2 \ldots a_{n-1}+1$). O símbolo ' : ' não é permitido imediatamente após o símbolo ' . ' ou a sequência 'FIM'.

O diagrama da figura 6 demonstra o funcionamento deste conjunto de regras e a figura 7 é um exemplo de sua aplicação.

Deve-se dizer que a proibição de se colocar consecutivamente o par "FIM" "INICIO" é uma restrição coerente com a definição sintática da linguagem, e é bastante conveniente adiantar a verificação devido ao seguinte:

a) supondo que o editor não detetasse a presença do par consecutivo ' . : ' teríamos, a seguinte situação: o primeiro símbolo ' . ' estaria no fragmento ($a_1.a_2 \ldots a_n$); quando o editor recebesse o símbolo ' : ', o fragmento ($a_1.a_2 \ldots a_n$) seria fechado (regra g), enviado para compilação digamos não apresentasse erro; o símbolo ' : ' deveria ser então colocado no fragmento ($a_1.a_2 \ldots a_{n-1} + 1.0$) conforme a combinação das regras g e b. Mas neste

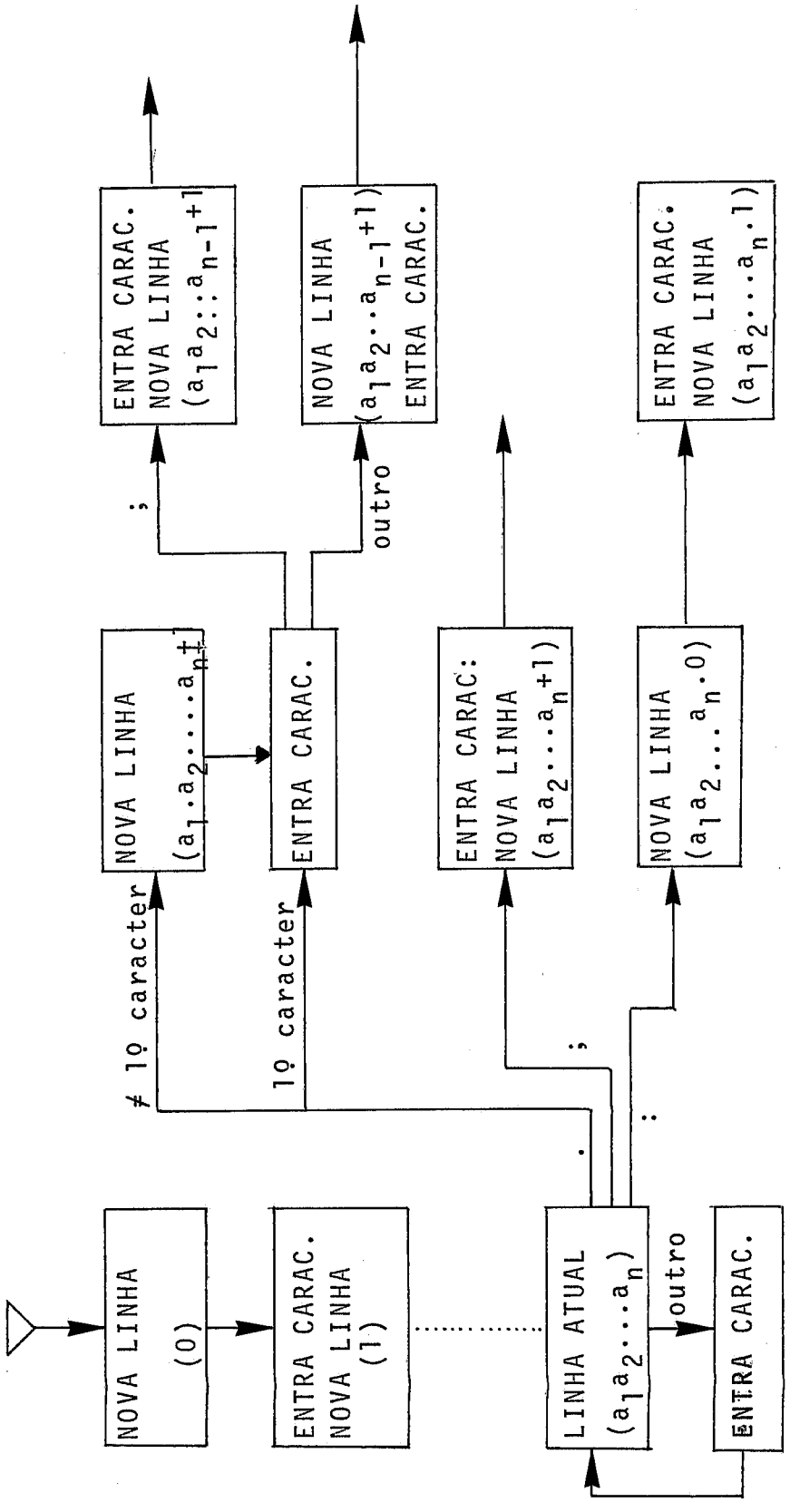


FIGURA 6 Diagrama do processo de fragmentação e numeração

Sequencia original

: SE B1 ENTAO : PARA V1←E1 PASSO E2 ATE E3 FACA : S1; S2;.; SE B2 ENTAO S3 SENAO : S3; S4; . ; . SENAO S5; S6; SE B3 ENTAO S7 SENAO SE B5 ENTAO S8 SENAO : S9; S10; . ; .

Sequencia fragmentada e numerada

REGRA	LINHA	
b	(0)	:INICIO
c,a	(1)	SE B1 ENTAO:
d	(1.0)	INICIO
c,a	(1.1)	PARA V1←E1 PASSO E2 ATE E3 FACA:
d	(1.1.0)	INICIO
c,a	(1.1.1)	S1;
e,a	(1.1.2)	S2;
e,f	(1.1.3)	.FIM;
g,a	(1.2)	SE B2 ENTAO S3 SENAO:
d	(1.2.0)	INICIO
c,a	(1.2.1)	S3;
e,a	(1.2.2)	S4.
f	(1.2.3)	FIM;
g	(1.3)	.FIM
g,a	(2)	SENAO S5;
e,a	(3)	S6;
e,a	(4)	SE B3 ENTAO S7 SENAO SE B5 ENTAO S8 SENAO:
d	(4.0)	INICIO
c,a	(4.1)	S9;
e,a	(4.2)	S10;
e,f	(4.3)	.FIM;
g	(5)	.FIM

FIGURA 7 Exemplo de programa fragmentado

SITUAÇÃO ANTERIOR

```
(6.4.2)          S1;
(6.4.3)          S2;
(6.4.4)          S3;
(6.4.5)

ENTRADA          . :              ( "FIM" "INICIO" )
```

SITUAÇÃO OBTIDA

```
(6.4.2)          S1;
(6.4.3)          S2;
(6.4.4)          S3;
(6.4.5)          .FIM:            ( já compilada e correta )
(6.5)                             ( já compilada e correta )
(6.5.0)          :INICIO          ( êrro )
```

FIGURA 8 Exemplo da necessidade de deteção do par "FIM""INICIO" pelo editor.

ponto além de termos criado e fechado o fragmento ($a_1.a_2...a_{n-1}+1$) totalmente vazio (e logo sem erro !) teremos de corrigir esse erro a partir do fragmento aberto ($a_1.a_2...a_{n-1}+1.0$). A figura 8 ilustra o caso.

Para outras construções não permitidas pela gramática, como por exemplo, o par consecutivo ' : : ', a correção desse erro pode ser normalmente tratada pelo processo de fragmentação descrito e sua verificação foi deixada a cargo do compilador.

IV.2. Representação da Estrutura de Programas em Árvores Binárias

No parágrafo anterior foi associado ao processo de fragmentação um esquema de numeração destinado a facilitar a compreensão e a visualização da estrutura fragmentada. Entretanto, a representação dessa estrutura no editor pode ser feita através da utilização de árvores binárias [*Knuth,72*].

Diferentemente do LSE 15 visto no capítulo anterior, onde os fragmentos eram ordenados sequencialmente segundo o número da linha e corresponderiam a uma tabela de acesso aleatório , os fragmentos de programa obtidos segundo as regras do pa

rágrafo anterior podem dar origem a uma árvore binária.

Chamaremos de "FILHO" e "IRMAO" aos dois tipos de ligações ("links") possíveis em uma árvore binária. Chamaremos de nó corrente, representado por Q, o nó ao qual estamos associando os caracteres entrantes. Chamaremos de "PAI", representado por P, o nó que satisfaça a seguinte propriedade:

Q = IRMAO (IRMAO(IRMAO(...IRMAO(FILHO(P))...)))

Em uma árvore binária, dado Q, P é único. Chamaremos de "IRMAO IMEDIATO", representado por R, o nó que satisfaça:

Q = IRMAO (R)

Chamaremos de "RAIZ" ao nó especial, sem "IRMAO", e cujo "FILHO" será Q quando o primeiro caracter do programa for admitido. A representação NOVO(X) indica que um novo nó foi gerado e será apontado por X.

A árvore será gerada, adotando-se o procedimento conforme o caracter recebido:

a) ' : ' - P ← Q
NOVO (R)
IRMAO [P] ← R
NOVO (Q)
IRMAO [R] ← Q

b) ' ; ' - R ← Q
NOVO (Q)
IRMAO [R] ← Q

c) ' . ' - R ← P
NOVO (Q)
IRMAO [R] ← Q

d) outro caracter - nada é feito.

A figura 9 mostra a árvore correspondente ao programa da figura 7 . Nesta figura as ligações verticais são do tipo "FILHO" e as horizontais do tipo "IRMAO". As enuplas estão representadas apenas para facilidade de verificação.

Note-se que cada valor da altura da árvore corresponde ao nível de aninhamento dos blocos do programa e, também, ao número de componentes da enupla associada. Esse valor será doravante designado por profundidade.

A deteção do caracter seguinte ao símbolo ' . ' não foi detalhada aqui, devendo ser explicitada mais adiante.

IV.3. Edição de Comandos com Fragmentação

Vamos analisar neste parágrafo o funcionamento do editor para os comandos da linguagem de referência e suas

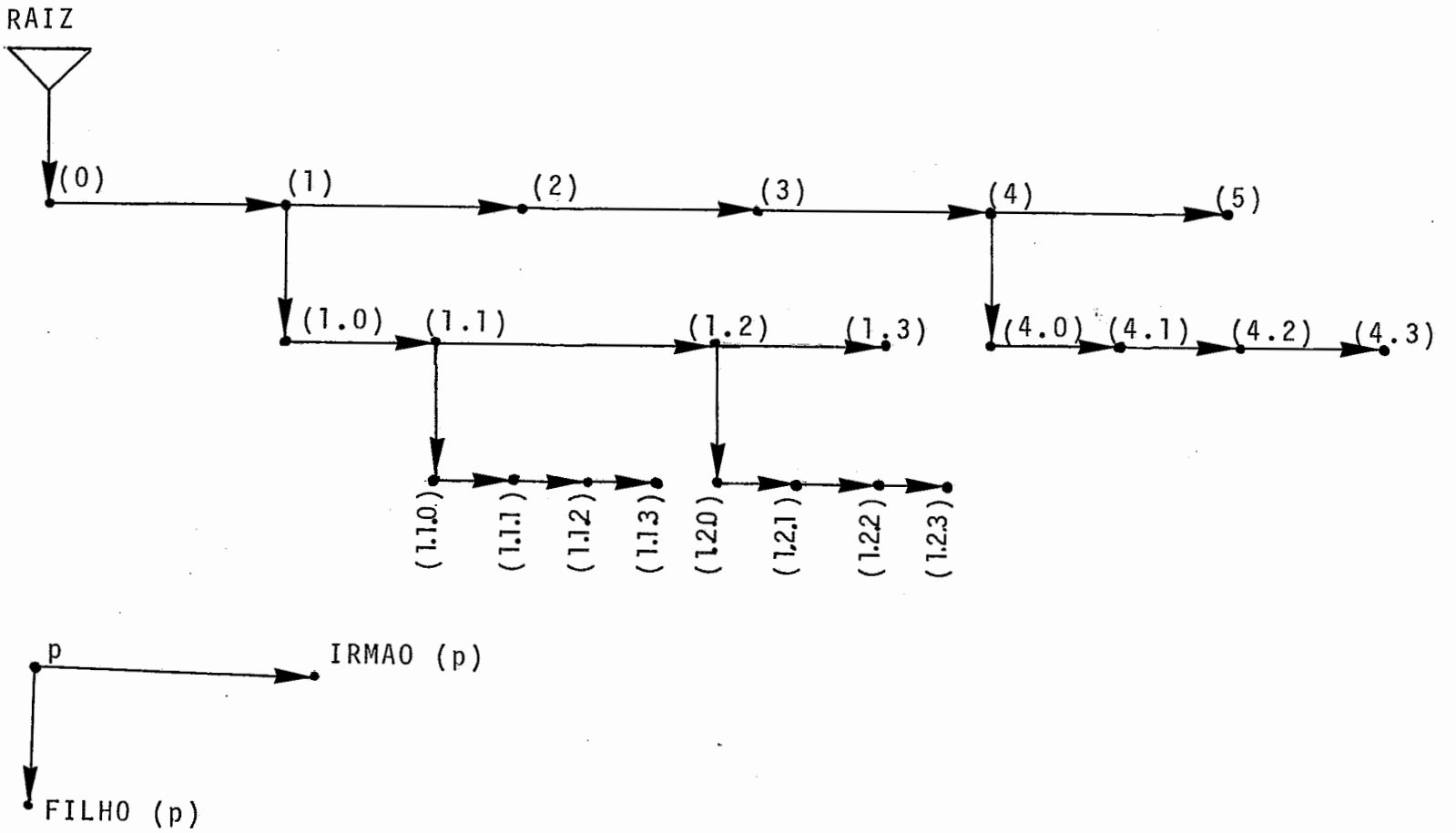


FIGURA 9 Arvore associada ao exemplo da figura 7.

consequências para o processo de compilação. Isto significa que em primeira aproximação não será tratado o problema da edição de declarações ou rotinas, mas apenas o corpo propriamente executável do programa principal.

Algumas características adicionais devem ser acrescidas ao conjunto editor-analisador sintático de modo a estender a flexibilidade do usuário na edição de comandos. Até este ponto ainda não foi levantada a limitação do comprimento dos comandos frente ao máximo de caracteres admitidos em uma linha física.

Para que isto seja obtido é necessário admitir a "transparência" semântica dos caracteres responsáveis pelo salto para o início de nova linha no terminal. Apenas como exemplo, vamos supor que estes caracteres sejam "RETURN"([R]) e "LINE FEED" ([LF]). Se algum outro tipo de terminal utilizar caracteres diversos o procedimento será análogo.

Para todos os efeitos sintáticos e semânticos, a presença dos caracteres [R] e [LF] é equivalente a presença do caracter branco ("SPACE"). Esta equivalência é tarefa do compilador.

Para efeitos de impressão sobre o terminal, no momento da digitação ou quando a listagem da linha for solicitada, os caracteres provocarão o salto para o início da nova linha , seguido de uma sequência de brancos, de modo a adequar a largura da margem esquerda ao estilo de apresentação do programa. É responsa-

bilidade do editor inserir tantos caracteres brancos quanto necessários.

Por exemplo, sendo a sequência digitada:

'SE_A>B_OU_ (R)(LF) ___C<D_ENTAO_'

será impresso sobre o terminal:

'_____SE_A>B_OU_'

'________C<D_ENTAO_'

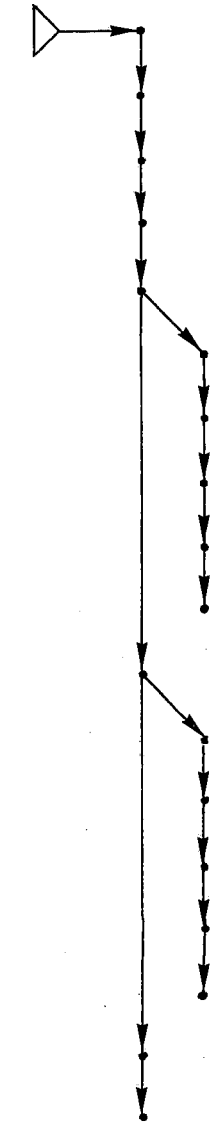
e será equivalente para a compilação à sequência:

'SE_A>B_OU_____C<D_ENTAO_'

Note-se que a mudança de linha comandada pelo usuário não provoca a mudança de fragmento, não altera a árvore associada e não causa o início da compilação por partes. É apenas um instrumento do programador para incrementar a legibilidade de seu texto, sem causar efeitos maiores ao programa.

O comprimento dos comandos torna-se independente da linha física do terminal e fica restrito apenas por decisões de implementação.

Vamos agora utilizar o exemplo da figura 10 para analisar a edição e a compilação de comandos. Serão focalizados principalmente o comportamento detalhado do editor e as conse



NUMERO	LINHA
(0)	INICIO
(1)	REAL A,B,C,D,R1,R2;
(2)	LEIA A,B,C;
(3)	D←B↑2-4*A*C;
(4)	SE D>=0 ENTAO:
(4.0)	INICIO
(4.1)	R1←(-B-RQ(D))/(2*A);
(4.2)	R2←(-B+RQ(D))/(2*A);
(4.3)	ESCREVA[2F8@4]R1,R2.
(4.4)	FIM
(5)	SENAO:
(5.0)	INICIO
(5.1)	R1←-B/(2*A);
(5.2)	R2←RQ(-D)/(2*A);
(5.3)	ESCREVA[F8@4,'+-',F8@4,'I']R1,R2.
(5.4)	FIM;
(6)	PAUSA;
(7)	.FIM

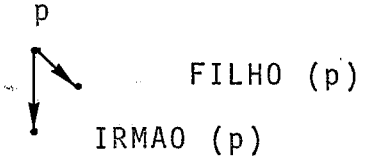


FIGURA 10 Exemplo de programa editado e fragmentado

A numeração consta apenas para referencia.

quências desse processo para a estrutura do compilador. Como a análise referir-se-á apenas aos comandos, será iniciada na linha (2). Vamos supor também que os comandos sejam submetidos na ordem apresentada na figura 10 .

A entrada dos comandos (2) e (3) se processa de maneira análoga e corresponde a entrada de comandos terminados em ' ; '. Exemplificando o processo com o fragmento (2):

a) a linha (2) foi aberta pelo ' ; ' final do fragmento (1);

b) a sequência 'LEIA A,B,C' é entrada caracter por caracter, limitando-se o editor a verificar se o entrante é diferente de ' ; ' ' : ' ou ' . ' ;

c) ao detetar o caracter ' ; ' o editor o coloca na linha e fornece o fragmento (2) completo ao compilador.

d) em caso de erro no fragmento os diagnósticos são emitidos, a sequência entrada é descartada e a linha (2) reaberta: Note que o compilador deve voltar ao estado anterior a análise desse fragmento . Para este caso essa exigência não apre-

sentará maiores problemas, pois o fragmento (2) admite análise sintática relativamente independente dos fragmentos anteriores.

e) caso não seja encontrado erro o código ou pseudo-código gerado é armazenado (a estrutura desse armazenamento bem como a conveniência de preservar o texto fonte serão discutidas posteriormente) e o editor abre a linha e o fragmento(3).

É importante notar que o compilador não dispõe de informações sobre o fornecimento fragmentado do programa. Do ponto de vista do compilador tudo se passa como se houvesse chegado ao fim de um registro (por exemplo, em cartão) e solicitasse, porém agora ao editor, a sequência do texto fonte.

As linhas (3), (4.1), (4.2), (4.3), (5.1), (5.2) e (5.3) do programa da figura 10 são em tudo semelhantes a linha recém analisada, embora algumas delas se encontrem em outra profundidade. Este tipo de fragmentação é o mais simples encontrado pois é constituído por um único comando sem comandos compostos em seu interior. Podemos admitir que após a análise de um fragmento deste tipo o compilador se encontra no "mesmo estado" que antes de sua compilação. É importante frisar que o editor sabe disso utilizando-se precisamente da mesma informação que o compilador:a pre

sença dos caracteres ' ; ' .

A linha (4) apresenta um fragmento de novo tipo: sendo encerrada pelo caracter ' : ' em lugar de ' ; ' irá gerar um nó filho em lugar do nó irmão, com implicações distintas. O processo detalhado é o seguinte:

a) a linha (4) está aberta pelo ' ; ' do fragmento (3);

b) a sequência 'SE_D>=0_ENTAO_:' é entrada caracter por caracter;

c) ao receber o símbolo ' : ' o editor envia o fragmento completo ao compilador;

d) em caso de erro os diagnósticos são emitidos, a sequência descartada e a linha (4) reaberta. O procedimento de recuperação do estado do compilador deverá ser análogo ao utilizado para o exemplo anterior;

e) caso não haja erro, o código é armazenado, a linha (4.0) e o nó correspondente são abertos, o editor escreve a sequência 'INICIO', fecha o nó e a linha sem chamar o compilador e a seguir abre o nó e a linha (4.1).

Um interessante ponto de distinção entre o processo de compilação por partes aqui descrito e o processo usual de compilação é que o compilador, utilizando a estrutura da árvore, pode prescindir de outras informações para completar aa análise do comando.

Em um compilador usual, a sequência 'IF <B> THEN BEGIN' não pode gerar o código completo pois faltam informações para resolver as referências para frente; basicamente, o endereço de desvio em caso da condição <B> ser falsa.

Utilizando-se a estrutura em árvore os endereços de desvio são conhecidos: o "FILHO" do nó corrente para condição verdadeira, e o "IRMAO" do nó corrente para condição falsa. O compilador pode gerar pseudo-código exatamente nessas condições para um interpretador que trabalhe coerentemente sobre a estrutura em árvore. Ou então, um algoritmo para percorrer a árvore quando transformá-la em uma estrutura linear usual, trocando as ligações "IRMAO" e "FILHO" por instruções de desvio, quando necessário.

O fragmento (4.4) corresponde a um terceiro tipo de problema a ser tratado. A presença do símbolo ' . ' indica ao editor e ao compilador que o comando composto aberto pelo símbolo ' : ' predecessor (linha (4.0)) está encerrado. Isto significa o encerramento da subarvore corrente e a volta a profundidade um nível acima. Entretanto, a estrutura da linguagem admite comandos com mais de um comando composto em seu interior. No

caso da linguagem de referência, apenas o "SE...ENTAO...SENAO". Com o processo de fragmentação adotado, partes do mesmo comando se encontram em fragmentos distintos. Para contornar esse problema é necessário sinalizarmos os nós correspondentes a esses fragmentos com o aviso especial "CONTINUA" (este aviso poderá ser, por exemplo, um "bit" da ligação "IRMAO"). Não há necessidade de se arguir o compilador para identificar este tipo de situação, pois a informação necessária é a ausência de ' ; ' após o símbolo ' . '. Assim sendo a linha (4.4) seria editada:

a) a linha (4.4) está aberta pelo ' ; ' do fragmento (4.3). Note-se que a presença deste caracter, coerentemente com a gramática, é desnecessária, no caso;

b) o caracter ' . ' é armazenado e abre-se novamente o terminal, ainda na linha (4.4);

c) se o caracter seguinte fosse ' ; ' (como na linha análoga (5.4)) a sequência ' . ; ' seria enviada ao compilador e a seguir seria aberta a linha (5). Em caso de erro continuariamos na linha(4.4);

d) como o caracter é outro (exceto ' ; ') a linha (4.4) é fechada e enviada sem o

último caracter recebido para o compilador. Em caso de erro perde-se a sequência toda e a linha (4.4) é reaberta . Em caso de não haver erro é aberta a linha (5) e sinalizados os nós (4) e (5) com "CONTINUA" (IRMAO < 0);

É importante notar que também neste ponto é conhecido o endereço de desvio após a execução do comando composto que sucede o 'ENTAO': se ' . ' estiver associado ao nó Q, o endereço será o nó M que satisfaça Q = FILHO (R) e M = IRMAO (IRMAO (R)).

IV.4. Geração da Árvore Associada e Alterações no Programa

Os algoritmos número 1 e número 2 descrevem respectivamente o processo de geração na árvore associada, e o processo de visita aos nós dessa árvore. A figura 11 apresenta uma árvore completa.

Estes processos estão intimamente associados aos processos de inserção ou substituição de fragmentos, e as restrições sofridas pelo programador em acessar certo fragmento em determinado instante irão determinar as alterações que o programa pode sofrer.

Fica desde já convencionado que, uma vez que o usuário passa atingir certo nó da árvore representado por Q, correspondente a linha $(a_1.a_2...a_n)$, poderá efetuar qualquer das seguintes operações:

a) INSERÇÃO - consistindo em abrir novo nó R, tal que R = IRMAO (Q). Este nó corresponderá a linha $(a_1.a_2...a_n+m)$, acrescida ao programa;

b) ELIMINAÇÃO - consistindo em eliminar da árvore o nó Q e a subárvore que o tem como raiz, bem como cortar do programa todas as linhas associadas aos nós eliminados;

c) SUBSTITUIÇÃO - consistindo em eliminar o nó Q e sua subárvore e em seguida reabri-lo sem subárvore, mantendo as ligações anteriores.

O algoritmo de geração da árvore tem por objetivo, em última análise minimizara necessidade de recompilação de trechos já entrados.

Cada nó da árvore, possui a seguinte estrutura:

±	COSTURA
	CODIGO
	FONTE
±	FILHO
±	IRMAO

onde:

a) "CODIGO" é um apontador para o fragmento do código objeto;

b) "FONTE" é um apontador para o fragmento do texto fonte;

c) "FILHO" e "IRMAO" são as ligações definidas anteriormente;

d) "COSTURA" é a ligação para percurso na árvore em sentido inverso.

Vamos admitir que o usuário disponha de três comandos do editor para possibilitar o contrôle do percurso na árvore: "ACIMA" (↑),"ABAIXO"(↓) e "A DIREITA" (→). A figura

O algoritmo número 2 efetua o contrôle desses caracteres. Do ponto de vista do usuário "ACIMA" significa obter a linha anterior na mesma profundidade, "ABAIXO" significa obter a linha posterior na mesma profundidade e "A DIREITA" significa descer

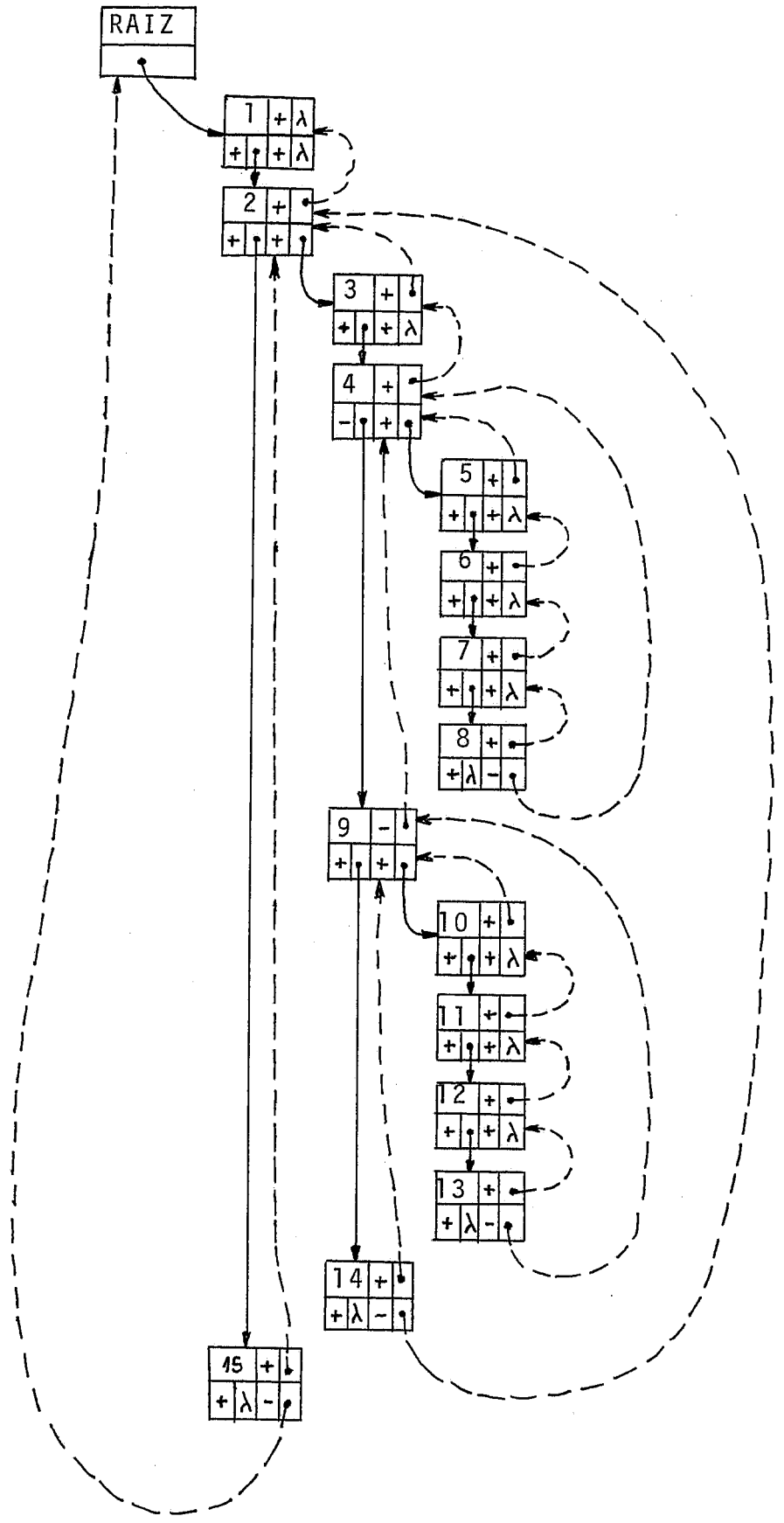
FIGURA 11 (vide página seguinte)

Arvore associada a um programa completo

Ao lado dos nós a sequencia que compõe cada linha

A estrutura simplificada de cada nó obedece ao padrão abaixo:

Nº DO NÓ	±	COSTURA
± IRMAO	±	FILHO




```
'INICIO'
'SE B1 ENTAO'
    'INICIO'
    'SE B2 ENTAO'
        'INICIO'
        'S4;'
        'S5;'
        'FIM'
    'SENAO'
        'INICIO'
        'S6;'
        'S7;'
        'FIM;'
    'FIM;'
'FIM;'
```

ALGORITMO Nº 1 GERAÇÃO DA ÁRVORE

```
% ESTE ALGORITMO ESTA ESCRITO NA LINGUAGEM DE REFERENCIA
% VIDE APENDICE C
%
% ALGUMAS ROTINAS NAO ESTAO DEFINIDAS SENDO APENAS DESCRITAS
%
% ROTINA LEITURA    LE 1 CARACTER E O COLOCA NA VARIAVEL
%                   CARACTER E NO BUFFER DA LINHA
%                   INCREMENTA N
%
% ROTINA NOVO (X)   OBTEM NOVO NO DE UMA LISTA DE NOS DISPONIVEIS
%                   E O APONTA POR X
%
% ROTINAS ERRO E CONFIRME   IMPRIMEM MENSAGENS
%
% VARIAVEIS UTILIZADAS
%
% Q = NO CORRENTE
% R = IRMAO IMEDIATO DE Q
% PROF = PROFUNDIDADE
% P[PROF] VETOR DOS ASCENDENTES DO NO Q
%          CORRESPONDE AO CONCEITO DE DISPLAY
% IRMAO[X],COSTURA[X],FILHO[X]= VETORES LIGAÇAO
% N = POSICAO DO CARACTER NA LINHA
% CARACTER = ULTIMO CARACTER ENTRADO
```

ALGORITMO Nº 1 (cont.) GERAÇÃO DA ÁRVORE

```
% OS SIMBOLOS '.',';',':' REPRESENTAM OS CODIGOS DESSES CARACTERES
% LAMBDA = VALOR POSITIVO ESPECIAL
%
% AS VARIAVEIS NAO ESTAO DECLARADAS
%
:INICIO
ROTINA CRIAIRMAO (X):                  % OBTEM NOVO NO APONTADO
    INICIO                             % POR X
    R ← X;                             % POSICIONA R
    NOVO (Q);                          % COMPLETA LIGACOES
    IRMAO[R] ← Q;                      %
    COSTURA[Q] ← R;                    %
    N ← 0;                             %
    .FIM;                              %
%
%
ROTINA DOISPONTOS:                     % TRATA O SIMBOLO INICIO
    INICIO                             %
    P[PROF] ← Q;                       % NOVO PAI
    NOVO (R);                          % ABRE NO INICIO
    FILHO[P[PROF]] ← R;                % CRIA FILHO
    CRIAIRMAO(R);                      % NOVO NIVEL PARA O NO
    PROF ← PROF + 1;                   % CORRENTE
    .FIM;                              %
```

ALGORITMO Nº 1 (cont.) GERAÇÃO DA ÁRVORE

```
%
%
ROTINA PONTO:                    % TRATA O SIMBOLO FIM
    INICIO                       %
    SE N ≠ 0 ENTAO CRIAIRMAO(Q)SENAO;  % NOVA LINHA
    LEITURA;                     % LE PROXIMO CARACTER
    PROF ← PROF - 1;             % SOBE NIVEL
    SE CARACTER = ':' ENTAO ERRO SENAO;      %
    FILHO[Q] ← - P[PROF];        % FECHA
    COSTURA[FILHO[P[PROF]]] ← P[PROF];       % SUBARVORE
    CRIAIRMAO(P[PROF]);          % ABRE PROXIMO NO
    SE CARACTER ≠ ';' ENTAO;     %
        INICIO                   % SINALIZA
        IRMAO[R] ← - IRMAO[R];   % CONTINUACAO
        COSTURA[Q] ← - COSTURA[Q];           %
        .FIM                     %
    SENAO;                       %
    .FIM;                        %
%
% PROGRAMA PRINCIPAL
%
NOVO(Q);                         % ABRE PRIMEIRO NO
RAIZ ← Q;                        % LIGACAO INICIAL
P[0] ← 0;                        % RAIZ = NO 0
```

ALGORITMO Nº 1 (cont.) GERAÇÃO DA ÁRVORE

```
PROF ← 1;                          %
N ← 0;                             % PRIMEIRO CARACTER
LEITURA;                           %
ENQUANTO CARACTER ≠ ':' FACA:      %
    INICIO                         %
    ERRO;                          % OBRIGATORIO
    N ← 0;                         % INICIO
    LEITURA;                       %
    .FIM;                          %
CRIAIRMAO(Q);                      % NOVO NO
%
% LOOP PRINCIPAL
%
ENQUANTO PROF > 0 FACA                                          %
    SE CARACTER = ':' ENTAO DOISPONTOS SENAO                    %
        SE CARACTER = ';' ENTAO CRIAIRMAO SENAO                 %
            SE CARACTER = '.' ENTAO PONTO SENAO;                %
.FIM
```

ALGORITMO Nº 2 PERCURSO

```
% ESTE ALGORITMO ESTA ESCRITO NA LINGUAGEM DE REFERENCIA
% VIDE APENDICE C
%
% UTILIZA MESMAS CONVENCOES E ROTINAS DO ALGORITMO 1
% '↓','↑','→' SAO OS CODIGOS DESSES SIMBOLOS
%
:INICIO
ROTINA SALTO(X):                    % EFETUA O SALTO
    INICIO                          % SEGUNDO A LIGACAO
    SE X[Q]≠ LAMBDA ENTAO           % PARAMETRO
       Q ← ABS (X[Q])SENAO;         %
    .FIM;                           %
%
%
ROTINA PARABAIXO                    % TRATA O CARACTER ↓
    INICIO                          %
    R ← Q;                          %
    SALTO (IRMAO);                  %
    SE R = Q E FILHO[Q]<0 ENTAO:        % MAL SUCEDIDO
       INICIO                           %
       SALTO(FILHO);                    % TENTA ALCANCAR
       PROF ← PROF - 1;                 % A LINHA FISICAMENTE
       SALTO(IRMAO);                    % CONTIGUA
                                        %
       .FIM                             %
```

ALGORITMO Nº 2 (cont.) PERCURSO

```
    SENAO;                          %
    .FIM;                           %
%
%
% LOOP PRINCIPAL
%
SE CARACTER = '↓' ENTAO PARABAIXO SENAO          %
    SE CARACTER = '↑' ENTAO:                     %
        INICIO                                   %
        SE FILHO[COSTURA]Q]] = Q ENTAO           % NO INICIO
            PROF ← PROF - 1 SENAO;               % MUDA DE NIVEL
        SALTO(COSTURA):                          %
        .FIM                                     %
    SENAO SE CARACTER = '→' ENTAO:               %
        INICIO                                   % TESTA SE O
        SE FILHO[Q] ≠ LAMBDA ENTAO               % SALTO SER A
           PROF ← PROF + 1 SENAO;                % EXECUTADO
        SALTO (FILHO);                           %
        .FIM;                                    %
%
%
```

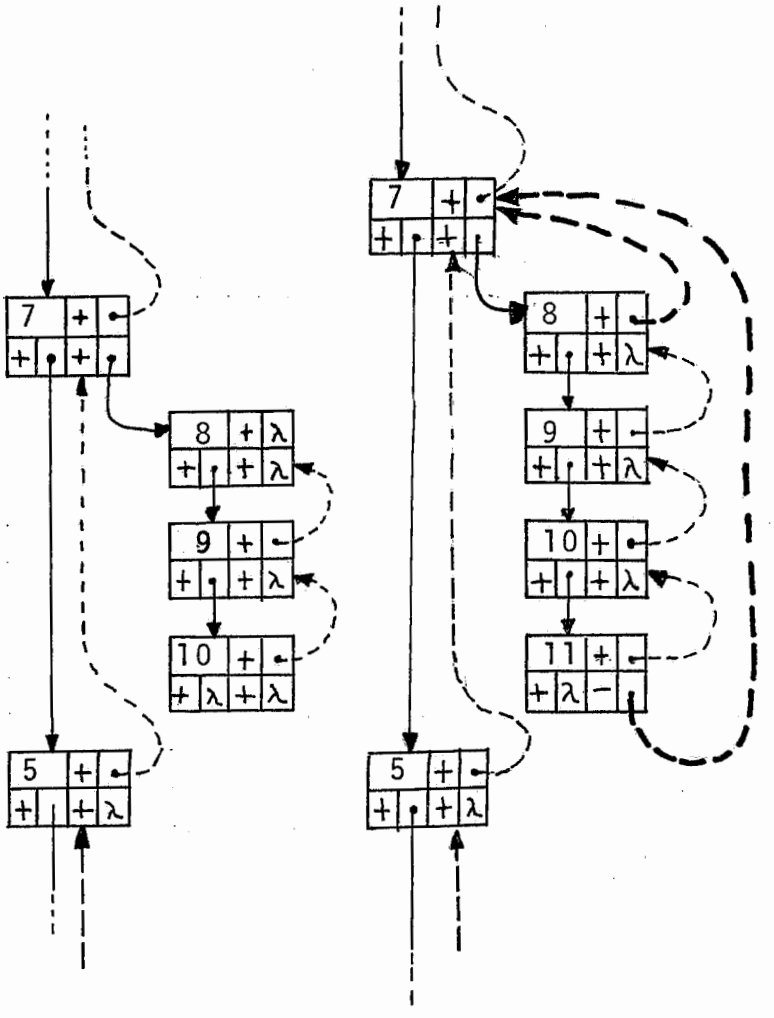
um nível na profundidade.

O ponto fundamental dos algoritmos de geração e percurso na árvore, responsável pela principal limitação do uso em modo conversacional é a maneira escolhida para gerar o nó "FILHO" ou seja quando da ocorrência do símbolo ' : '. Como pode ser visto no algoritmo número 1 o nó "FILHO" é gerado sem que seja colocada a "COSTURA" correspondente. Esta "COSTURA" será completada apenas quando da ocorrência do símbolo ' . ' correspondente. Isso significa que apenas quando o bloco entrante estiver completo será possível acessar comandos mais externos. Esta restrição é devida ao fato do compilador ignorar totalmente a entrada fragmentada do programa, não dispondo portanto de meios para preservar as informações correntes sobre o bloco entrante. Apenas quando essas informações já são desnecessárias, ou seja, quando o bloco estiver completo, é permitido ao usuário alterar outros pontos do programa. A figura 12 ilustra a situação.

Tendo sido determinados quais os pontos acessíveis de um programa em certa situação, são apresentados os algoritmos números 3, 4 e 5 utilizados para inserção, eliminação e substituição de comandos.

Para inserção duas observações devem ser feitas:

a) a inserção do comando após uma linha "FIM" tem efeito análogo a inserção após o comando gerador desse bloco. O objetivo desta facilidade é permitir ao usuá-



Para o bloco incompleto apenas os nós 8,9,10 e suas subarsão acessíveis.

FIGURA 12 Ligações para os blocos completos e incompletos

rio fazer a inserção em ponto logicamente contiguo (representado pelo "IRMAO") como fisicamente contiguo (pois a linha será imediatamente seguinte ao 'FIM', na listagem);

b) não é permitida a inserção de comandos após a linha "SE...ENTAO" (devido a existência do nó continuação representativo de 'SENAO');

O algoritmo de eliminação de comandos trabalha sob as seguintes restrições:

a) não é permitida a eliminação das linhas "INICIO" e "FIM", isoladamente. Estas linhas poderão ser eliminadas quando o bloco completo for eliminado.

b) as linhas marcadas como continuação(COSTURA|Q| < 0) não podem ser isoladamente eliminadas. Isto significa que é permitido eliminar as linhas "SE...ENTAO" e "SENAO" ao mesmo tempo porém, não é permitido eliminar uma delas mantendo a outra.

O algoritmo de substituição de comandos es

tá construído de modo a possibilitar ao usuário a máxima flexibilidade possível, sem sobrecarregar a tarefa de recompilação. Para isso os comandos foram classificados em três tipos conforme a estrutura da subarvore gerada:

a) o tipo 1 é formado pelos comandos que não geram subarvores, ou seja não contêm blocos em seu interior. Estes comandos são identificados pelo caracter ' ; ' que os encerra.

b) o tipo 2 é formado pelos comandos com um bloco em seu interior, identificados pelo símbolo ' : ' no encerramento da linha, e pelo símbolo ' ; ' após o símbolo ' . ' correspondente ao bloco.

c) o tipo 3 é formado pelo comando 'SE...ENTAO...SENAO' que admite dois blocos em seu interior. A identificação deste comando tanto poderá ser feita através de comunicação do compilador ao editor, como pela ausência do caracter ' ; ' após o símbolo '.' do primeiro bloco.

Uma matriz de decisão é usada pelo algoritmo para, em função dos tipos da linha substituída é substituinte , permitir ao usuário maior flexibilidade de escolha quando a alteração desejada. Isto implica em conhecer o tipo da linha substituin-

te, e portanto admiti-la, antes de eliminar a linha substituída. Para permitir tal efeito, a linha substituinte é provisoriamente admitida como "IRMAO IMEDIATO" da linha substituída, analisada sintaticamente e compilada, para, apenas nesse momento, o editor efetivar a substituição.

O algoritmo prevê certas restrições básicas:

a) não é permitida a substituição das linhas "FIM" "INICIO" e "SENAO", visto que estes fragmentos não podem ser substituídos por outros fragmentos que não eles mesmos, sem causar erro no restante do programa. Esta restrição, na verdade, apenas obriga o usuário, quando quiser substituir algumas dessas linhas, a referir-se diretamente aos comandos aos quais estas linhas estão logicamente associadas: ao comando ao qual o bloco pertence, no caso de 'FIM' e 'INICIO', e a linha 'SE...ENTAO' no caso de 'SENAO' ;

b) não é permitido à linha entrante ser terminada por 'FIM', pois essa alteração obrigaria um desnecessário esforço de recompilação. Esta recompilação é desnecessária porque, caso a substituição esteja sendo feita no bloco entrante, o símbolo 'FIM' pode ser admitido da maneira usual e caso esteja sendo feita em bloco completo, significa uma alteração de maior porte no programa e deverá ser tratada por um comando especial do editor e não pelo algoritmo de substituição. No capítulo seguinte algumas extensões propostas cuidam de dar ao usuário a possibilidade de tratar tais casos.

As decisões tomadas pelo algoritmo em função das linhas trocadas podem ser resumidas no seguinte:

a) em qualquer caso descrito nos itens seguintes, que envolvam a eliminação ou a manutenção de blocos internos, o editor consulta o usuário antes de efetivar a mudança;

b) a substituição de um comando por outro de mesmo tipo é feita diretamente. Caso existam blocos interiores estes são mantidos.

c) a substituição de comandos com blocos interiores por comando sem blocos interiores, causa a eliminação desses blocos.

d) a substituição de um comando sem blocos interiores por outro que os admita é feita normalmente, e a admissão de linhas prossegue com a primeira linha do novo bloco aberto.

e) a substituição de um comando com um bloco pelo comando "SE...ENTAO", coloca esse bloco como seguindo ao "ENTAO" e abre a linha "SENAO".

f) a substituição de um comando com dois blocos por outro com bloco único, força o editor a solicitar ao usuário qual o bloco mantido e qual o eliminado.

Esta configuração da matriz de decisão do algoritmo é apenas uma dentre os diversos conjuntos de atitudes

possíveis. A principal preocupação ao se formular o algoritmo foi mantê-lo extremamente simples e passível de alterações. A escolha final da providência a ser tomada em cada caso dependerá, em última análise, da frequência com que ocorram e das necessidades mais comuns aos usuários, quando de sua implementação. O algoritmo de substituição apresentado deve ser considerado, portanto, como uma primeira aproximação do problema, porém a facilidade com que pode ser modificado, bem como o grande número de alternativas possíveis para tratar cada caso vem evidenciar que a estrutura em árvore associada a fragmentação do programa é um instrumento com potencial suficiente para resolver tal problema.

O conjunto dos algoritmos apresentados pode ser agora evidenciado no que constituiu, talvez, sua fundamental diretiva de projeto: em nenhuma situação das até agora descritas houve necessidade de recompilar qualquer trecho do programa.

Evidentemente, tal foi obtido a custa de restrições feitas ao modo conversacional de admissão: eliminação de blocos, impossibilidade de acesso e proibições de alterações. Apesar disso, e não será esta uma afirmação leviana, a grande maioria das situações normais que os usuários deverão enfrentar estará satisfatoriamente resolvida. Para um programador devidamente consciente que sua tarefa deverá estar resolvida antes de apresentá-la a máquina, que se coloque frente ao terminal com seu programa escrito, pronto e estudado, que se compenetre que o terminal é uma máquina de receber programas e não resolver problemas, para este programador o sistema aqui descrito será uma ferramenta útil e poderosa.

ALGORITMO Nº 3 INSERÇÃO

```
% ESTE ALGORITMO ESTA ESCRITO NA LINGUAGEM DE REFERENCIA
% VIDE APENDICE C
%
% UTILIZA MESMAS CONVENCOES E ROTINAS DOS ALGORITMOS 1 E 2
%
% O NOVO NO E INSERIDO LOGICAMENTE APOS O NO Q
INICIO
SE FILHO [Q]<0 ENTAO SALTO(FILHO) SENAO;          % FIM
SE IRMAO [Q]<0 ENTAO ERRO SENAO;                  % CONTINUACAO
    INICIO
    R ← Q;                          %
    NOVO(Q);                        %
    IRMAO[Q] ← IRMAO[R];            % ACERTO DAS
    IRMAO[R] ← Q;                   % LIGACOES
    COSTURA[Q] ← R;                 %
    COSTURA[IRMAO]Q]] ← Q;          %
    .FIM;                           %
.FIM
```

ALGORITMO Nº 4 ELIMINAÇÃO

```
% ESTE ALGORITMO ESTA ESCRITO NA LINGUAGEM DE REFERENCIA
% VIDE APENDICE C
%
% UTILIZA MESMAS CONVENCOES E ROTINAS DOS ALGORITMOS 1 E 2
%
% PROIBIDO ELIMINAR INICIO FIM E CONTINUACAO
%
INICIO
SE FILHO[Q]<0  OU                    % FIM        .
   FILHO[COSTURA]Q]]=Q OU            % INICIO
   COSTURA[Q]<0 ENTAO ERRO SENAO:    % CONTINUACÃO
   INICIO                            %
   L ← SE IRMAO[Q]< 0 ENTAO IRMAO[- IRMAO]Q]]      %
                      SENAO IRAMO[Q];              %
   IRMAO[COSTURA]Q]] ← L;                          %
   COSTURA[L] ← COSTURA[Q];                        %
   .FIM;
.FIM;
```

ALGORITMO Nº 5 SUBSTITUIÇÃO

```
% ESTE ALGORITMO ESTA ESCRITO NA LINGUAGEM DE REFERENCIA
% VIDE APENDICE C
%
% UTILIZA MESMAS CONVENCOES E ROTINAS DOS ALGORITMOS 1 E 2
%
% TIPO 1: FRAGMENTO TERMINADO EM ';'
% TIPO 2: FRAGMENTO 'SE...ENTAO:'
% TIPO 3: OUTROS
%
% MD(1,1), MD(2,2), MD(3,3)=1
% MD(1,2), MD(1,3)=2
% MD(2,1), MD(3,1)=3
% MD(2,3), MD(3,2)=4
%
% O COMPILADOR FORNECE O TIPO
% COMO E NECESSARIO CONHECER O TIPO DA LINHA SUBSTITUINTE
% ESTA E PROVISORIAMENTE COLOCADA COMO IRMAO IMEDIATO
% DA LINHA SUBSTITUIDA
%
ROTINA SUBARVORE(Q)                    % ESTA ROTINA PRENDE A
    INICIO                             % SUBARVORE AO NOVO NO
    SE FILHO[Q]≠ LAMBDA ENTAO:         %
       INICIO                          %
       ENQUANTO IRMAO[L]≠ LAMBDA FACA L ← IRMAO[L];        %
```

ALGORITMO Nº 5 (cont.) ELIMINAÇÃO

```
      FILHO[L] ← - Q;
      COSTURA[FILHO]Q]] ← Q;
      .FIM;
   SENAO;
%
SE FILHO[Q]<0 OU                          % PROIBIDO TROCAR
   COSTURA[Q]<0 OU                        % FIM INICIO
   FILHO[COSTURA]Q]]= Q ENTAO ERRO SENAO:     % E CONTINUACAO
   INICIO
   S ← Q;                                 % S NO A SUBSTITUIR
   NOVO(Q);                               %
   IRMAO[Q] ← S;                          % ENTRADO COMO
   IRMAO[COSTURA]S]] ← Q;                 % IRMAO IMEDIATO
   COSTURA[Q] ← COSTURA[S];               %
   COSTURA[S] ← Q;                        %
% NESTE PONTO A NOVA LINHA E ENTRADA E COMPILADA
% O COMANDO SEGUINTE NAO EXISTE NA LINGUAGEM
% MAS E USADO PARA MAIOR CLAREZA
   CASO MD[TIPO]Q], TIPO[S]] DE:          %
      INICIO                              % CASO 1
      FILHO[Q] ← FILHO[S]:                % FRAGMENTO DE
      SUBARVORE(Q);                       % MESMO TIPO
      COSTURA[IRMAO]S]] ← Q;              %
      IRMAO[Q] ← IRMAO[S];                %
```

ALGORITMO Nº 5 (cont.) ELIMINAÇÃO

```
      .FIM;                          %
      :INICIO                        % CASO 2
      IRMAO[Q] ← IRMAO[S];           % ENTRA TIPO 1
      COSTURA[IRMAO]S]] ← Q;         %
      .FIM;                          %
      :INICIO                        % CASO 3
      IRMAO[Q] ← IRMAO[S];           % ENTRAM COMANDOS
      COSTURA[IRMAO]S]] ← Q;         % COM BLOCO E SAI
      P[PROF] ← Q;                   % COMANDO SEM
      NOVO(Q);                       % BLOCO
      FILHO[P[PROF]] ← Q;            %
      CRIAIRMAO(Q);                  %
      .FIM;                          % A EDIÇAO CONTINUA
.FIM
```

Para os demais deverá ser um aprendizado de disciplina.

Mesmo com tais observações em mente, serão descritas no capítulo seguinte algumas extensões do editor capazes de solucionar situações relativamente anormais.

IV.5. Rotulação de Comandos

O processo de edição adotado vem obrigar a utilização de rótulos dentro da linguagem.

Na medida em que o processo de fragmentação é responsabilidade do editor e não existe qualquer modo do programador referenciar cada fragmento, a linguagem deve admitir maneiras de atingir uma linha determinada através do comando 'DESVIE PARA...'. Na definição da linguagem feita anteriormente recorde-se que este comando foi descrito provisoriamente na forma 'DESVIE PARA <rotulo>', deixando para após o estabelecimento do processo de fragmentação a escolha do tipo de rótulo adotado.

Como foi visto na análise do LSE 15 e se encontra fartamente discutido na literatura [Wulf,71] [Dijkstra,68], a utilização indiscriminada de comandos tipo "GO TO ..." é um estilo de programação que tende a ser prejudicial ao trabalho do programador. Com o objetivo de limitar as consequências danosas de seu uso, vamos estabelecer restrições para o uso de rótulos. A justificativa para tanto pode ser encontrada na já extensa polêmica sobre programação estruturada.

Em primeiro lugar vamos estabelecer que os rótulos devem ser iniciados por um caracter especial, digamos ' " '. Esta providência destina-se a evitar o uso de uma declaração especial para rótulos.

Os rótulos quando existirem, deverão ser o primeiro elemento sintático de cada fragmento. Evidentemente, não são admitidos rótulos em declarações ou cabeçalhos de rotinas.

Supondo que o comando 'DESVIE PARA ...' pertença a linha Q, com profundidade q, e o rótulo referenciado por esse comando pertença a linha R com profundidade r, são condições obrigatórias:

a) $q \geq r$;

b) se q = r então, obriga-se
   Q = IRMAO (IRMAO...(IRMAO(R)...) ou
   R = IRMAO (IRMAO...(IRMAO(Q)...)

c) se q > r, sendo P[q-1], P[q-2]...P[r] os ascendentes diretos de Q, obriga-se
   P[r] = IRMAO (IRMAO...(IRMAO(R)...) ou
   R = IRMAO (IRMAO...(IRMAO(P[r])...)

Note-se que estas regras correspondem a limitar o desvio para os irmãos do comando 'DESVIE PARA...' ou para qualquer irmão de um dos blocos ao qual pertence. O desvio para o interior de subrotinas fica automaticamente eliminada.

Resta definir a maneira como o editor e o compilador processarão o sistema de rotulação. Um processo simples é a criação de uma tabela de indexação , onde à primeira ocorrência de um rótulo o introduz na tabela em conjunto com a profundidade do fragmento. Esta ocorrência tanto poderá ser sua associação em certo comando, como sua utilização em uma instrução de desvio. Na segunda ocorrência do rótulo será verificada sua adequação as regras. Caso haja inadequação a linha causadora da segunda ocorrência do rótulo será considerada incorreta. Na hipótese em que mais de um comando 'DESVIE PARA...' utilize o mesmo rótulo antes que este seja definido, todas as situações serão testadas, indicando o compilador qual a instrução de desvio causadora do erro.

Este processo de verificação do uso rótulos é uma das tarefas da compilação que entretanto deverá solicitar ao editor a adequação do posicionamento na árvore.

IV.6. Edição e Compilação de Fragmentos quaisquer

Após a apresentação do trabalho do pré-processador na edição dos comandos, será aqui tratada a fragmentação e a compilação dos demais elementos do programa. O problema que aqui se apresenta é análogo ao problema da compilação em modo conversacional de uma estrutura em blocos semelhante ao ALGOL 60, onde são permitidas declarações em cada bloco. A simplificação feita na linguagem, basicamente substituindo o bloco do ALGOL por comandos com

postos, permitiu que grande parte do programa, constituída pelos comandos, pudesse ser editada e compilada em modo conversacional, evitando-se qualquer tarefa de recompilação.

Poderíamos, também para as declarações e cabeçalhos de rotina, utilizar exatamente o mesmo sistema adotado para fragmentação de comandos. Nesse caso as consequências seriam as seguintes:

a) a verificação da adequação entre a definição e a utilização das variáveis e rotinas teria de ser transferida para a fase de execução. Isto implica, praticamente, no uso de interpretador;

b) na medida em que a verificação na fase de análise sintática é incompleta, seriam detetados erros sintáticos conjuntamente com erros de lógica do programa.

Acreditamos que é conveniente para um sistema destinado a ensino que os erros sintáticos sejam anotados quando da própria admissão da linha, permitindo ao programador tratar um erro de cada vez e, também, efetuar a correção do restante do programa antes mesmo de submetê-lo. O compilador atuaria, então, como um efetivo auxiliar do aprendizado. Em outras situações diversas da descrita talvez seja conveniente utilizar um interpretador simplificando a edição e a análise sintática.

Vamos, portanto, assumir o problema de efetuar uma compilação completa. Em primeiro lugar, fica mantido o esquema de fragmentação adotado, visto que os mesmos delimitadores ' . ', ' : ', ' ; ' são usados na definição sintática das declarações e rotinas. Assim sendo, dois tipos de linhas podem ser distinguidos:

a) declarações e cabeçalhos de rotina, tipo 1;

b) comandos, tipo 2.

A distinção entre eles se dá, basicamente, pela extensão dos efeitos da substituição de uma das linhas:

a) a alteração de uma declaração altera o contexto de todo o programa, se for global, ou de toda a rotina, se for local. A alteração de um cabeçalho de rotina altera o contexto de todo o programa.

b) a alteração de um comando não modifica o contexto de qualquer parte do programa, exceto o comando 'DESVIE PARA...' que já teve o processo de tratamento definido anteriormente.

A solução adotada prevê a recompilação de

parte do programa em caso de alteração de declaração ou cabeçalhos de rotina e o algoritmo número 6 descreve a extensão do trecho recompilado. Trata-se em última análise de submeter novamente ao compilador todas as subarvores descendentes de todos os irmãos do nó alterado.

A maneira como o algoritmo foi construído garante que se a alteração ocorrer em uma declaração global ou cabeçalho de rotina todo o programa será recompilado. Se a alteração ocorrer em uma declaração local apenas o corpo da rotina será re - compilado.

ALGORITMO Nº 6 RECOMPILAÇÃO

```
% ESTE ALGORITMO ESTA ESCRITO NA LINGUAGEM DE REFERENCIA
% VIDE ANEXO C
%
% NO CORRENTE = Q
%
% RECOMPILE E UMA CHAMADA AO ANALISADOR SINTATICO
%
INICIO
ROTINA DESCEARV (X)                          % ESTA ROTINA VARRE
     INICIO                                  % UMA SUBARVORE EM
     ENQUANTO IRMAO[X]≠LAMBDA FACA           % PROFUNDIDADE FIXA
          INICIO                             %
          RECOMPILE (X);                     % SE HOUVER SUBARVORE
          SE FILHO[X]≠LAMBDA ENTAO           % FILHO EFETUA CHAMADA
               DESCEARV (FILHO[X]);          % RECURSIVA
          X←IRMAO[X];                        %
          FIM;                               %
%
%
% PROGRAMA PRINCIPAL
%
ENQUANTO FILHO[COSTURA[Q]]≠Q FACA            % PROCURA O NO INICIO
          Q←COSTURA[Q];                      %
DESCEARV (Q);                                %
FIM;
```

CAPÍTULO V

EXTENSÕES AO PRÉ-PROCESSADOR. CONCLUSÕES.

V.1. Extensões ao Pré-processador

Neste parágrafo serão propostos mecanismos adicionais para a edição e compilação de programas fragmentados , conforme o descrito no capítulo anterior.

Alguns adendos, aqui sumariamente descritos, embora não façam parte do núcleo fundamental do pré-processador visam atender determinadas situações nas quais seria exigido demasiado trabalho do usuário para corrigir seu programa apenas substituindo, eliminando ou inserindo linhas. Outros adendos referem-se a facilidades de edição propriamente dita, ou seja facilidades de submissão e alterações no texto fonte, sem implicações para o processo de compilação. Finalmente é proposta e discutida uma política de incentivo ao uso de comentários.

V.1.1. Criação de Blocos de Comandos já Compilados

Uma interessante facilidade de edição, uti

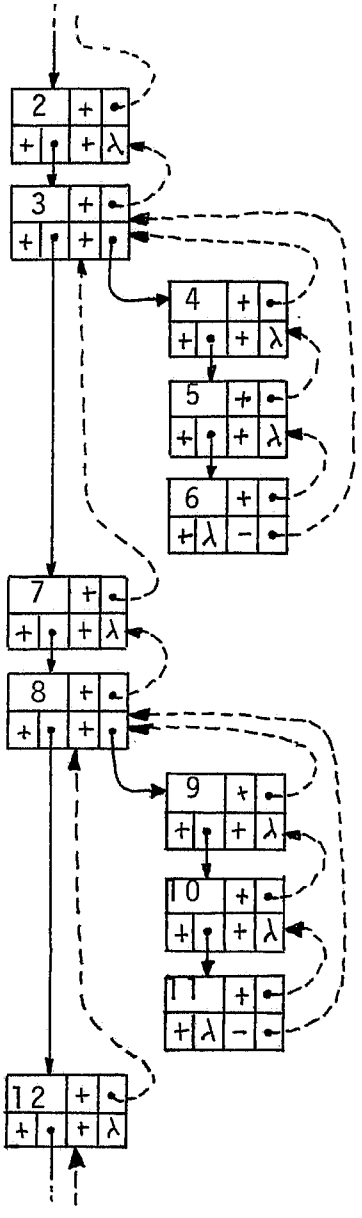
lizável através de um comando especial do editor - de "mesmo nível" que o comando de inserção, por exemplo - seria o que poderíamos denominar 'CRIEBLOCO'.

Imaginemos um programa já entrado, conforme a figura . Admitindo que o nó corrente seja o nó 2, associado à linha (2.3), nesse ponto o usuário comanda o pré-processador com 'CRIEBLOCO'. Imediatamente é aberta a nova linha (2.4) e criado o novo nó 20. A linha (2.4) deverá obrigatoriamente terminar com o símbolo 'INICIO'. Após a admissão e compilação desta linha é aberta a linha (2.4.1) e o nó 21, e o pré-processador aguarda agora o sinal 'PARABAIXO' (⊕). A cada sinal ⊕ recebido um fragmento "IRMÃO" é admitido, bem como toda sua subárvore "FILHO". O recebimento do caracter 'FIM' encerra o processo.

A figura é auto explicativa e a única restrição é a proibição de incluir no bloco um fragmento 'FIM' já existente, na mesma profundidade do bloco. Com a estrutura em árvore adotada, basta que nós com continuação ('SE...ENTAO...SENAO') sejam incluídos conjuntamente, para que não haja necessidade de recompilação. Como se pode concluir o comando 'CRIEBLOCO' nada mais é que o comando 'INSERÇÃO', de certa forma generalizado.

V.1.2. Reutilização de Comandos em Blocos Eliminados

A idéia deste comando do editor é obter a si



(2.3) S1;

(2.5) ENQUANTO B1 FACA:

(2.5.1) INICIO

(2.5.2) S2;

(2.5.3) FIM;

(2.7) S3;

(2.9) ENQUANTO B2 FACA:

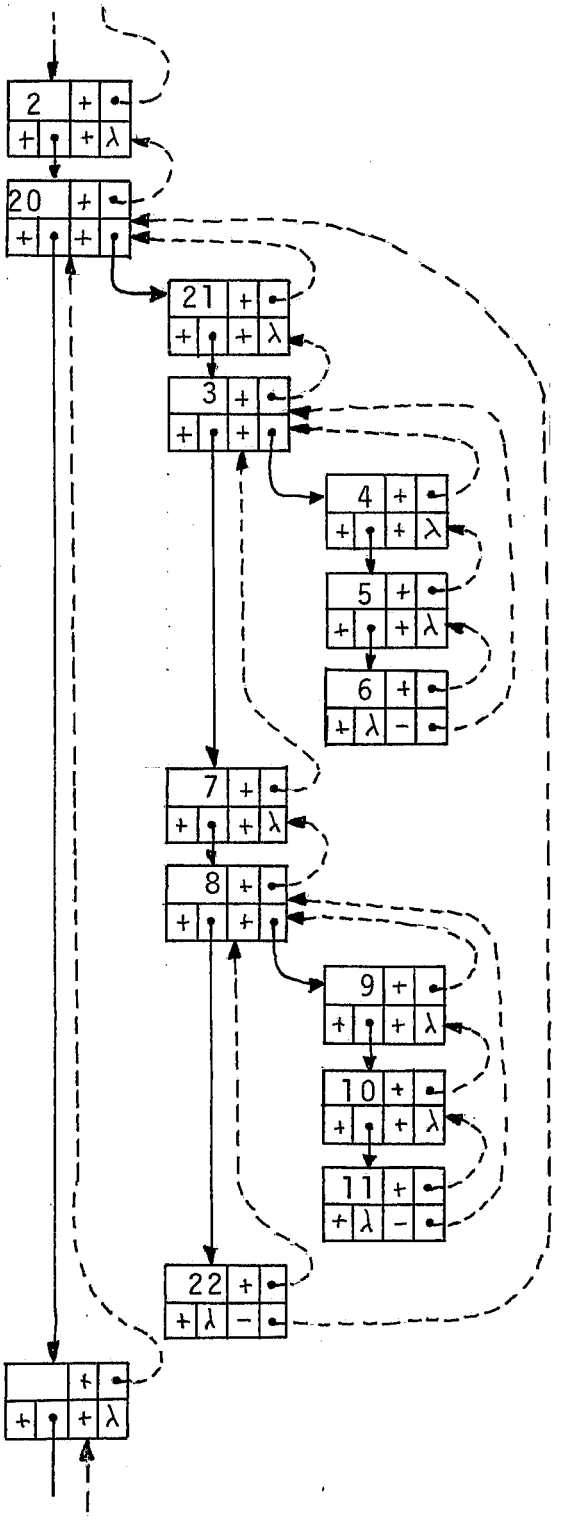
(2.9.1) INICIO

(2.9.2) S4;

(2.9.3) FIM;

(2.11) S5;

FIGURA 13 Utilização do comando CRIEBLOCO.
Situação inicial.



SE B6 ENTAO:

INICIO

FIM

FIGURA 13 (cont.) Situação após a entrada: 'CRIEBLOCO'
'SE B6 ENTAO:'
'+++.'

tuação inversa do comando 'CRIEBLOCO': trata-se de generalizar a eliminação.

Esta função do pré-processador permite eliminar uma certa instrução do programa com uma ou mais subárvores em seu interior salvando os comandos destes blocos.

Ao contrário do 'CRIEBLOCO' a totalidade da subárvore é recuperada, exceto os nós 'INICIO' e 'FIM'. A figura 14 demonstra sua aplicação.

V.1.3. Edição de Caracteres na Linha

O fato do editor controlar a entrada do texto fonte, caracter por caracter não impede que uma série de facilidades sejam adicionadas para edição de uma linha, tais como: retorno do cursor apagando ou não os caracteres já entrados, inserção de caracter no meio da linha com deslocamento dos caracteres à esquerda, salto automático para o início da linha ou para o último caracter admitindo substituição ou eliminação de caracteres ou trechos inteiros de linha, etc.

Estas facilidades, disponíveis automaticamente para a linha entrante, podem ser também aproveitadas em conjunção com o comando seguinte.

V.1.4. Reutilização do Texto Fonte

SITUAÇÃO INICIAL nó corrente (2.5)

```
(2.3)        S1;
(2.5)        ENQUANTO B1 FACA:
(2.5.1)         INICIO
(2.5.2)         S2;
(2.5.3)         FIM;
(2.7)        S3;
(2.9)        S4;
```

SITUAÇÃO APÓS 'REUTILIZE'

```
(2.3)        S1;
(2.5.2)      S2;
(2.7)        S3;
(2.9)        S4;
```

FIGURA 14 Reutilização de comandos em blocos eliminados

Este comando do editor permite que, em lugar da substituição de uma linha tal como descrito no capítulo anterior, onde a nova linha teria de ser inteiramente escrita, o usuário aproveite o próprio texto da linha substituida para alterá-lo e submete-lo novamente. Do ponto de vista da compilação o processo seria em tudo análogo a uma substituição simples.

V.1.5. Edição de mais de um Comando por Linha

Através deste comando especial do editor, o usuário pode substituir o salto automático para a linha seguinte quando o editor deteta os caracteres ' . ', ' : ', '; ' por um salto automático apenas quando forem detetados os pares de caracteres '.RC ', ' :RC ', ' ;RC ', onde RC é o caracter "RETURN". Com isto é possível editar mais de um comando por linha, embora o editor os coloque em nós distintos tal como no esquema original e a compilação dos comandos seja feita separadamente.

V.1.6. Tabulação de Comentários

Mediante comando especial do editor é possível ao usuário escolher determinada coluna do texto fonte, para onde o editor saltará automaticamente e imprimirá o caracter indica-

tivo do início de comentário, sempre que no esquema original deves se saltar para nova linha.

Este esquema é um forte incentivo à documentação do programa, praticamente linha a linha, coerente com a proposta de estilo de programação discutida nos capítulos anteriores e especialmente conveniente para uma linguagem de ensino de computação.

Outro esquema alternativo poderia prever o salto automático para uma coluna distante de um intervalo fixo do último caracter da linha. Em qualquer caso as maiores facilidades devem ser concedidas ao programador para escolher e alterar o formato de apresentação de seus comentários, devendo cada linha fonte ser preservada na exata forma escolhida pelo usuário.

V.2. Conclusões

A visão de conjunto do presente trabalho vem caracterizá-lo, principalmente, como uma primeira e inicial aproximação aos problemas de compilação em modo conversacional. Sua apresentação dentro de uma ótica voltada para o objetivo de ensino de computação, a par de lhe conferir um sentido prático com razoáveis possibilidades de utilização a médio prazo, não lhe retira a generalidade necessária a um estudo teórico sobre a compilação fragmentada de programas em linguagens tipo ALGOL.

Como pode ser observado no decorrer do trabalho, raras foram as referências bibliográficas encontradas, que sequer se aproximassem do assunto tratado. Como mais forte indicação ressalte-se os trabalhos sobre o compilador incremental "CPL/I" [*Blaizot,72*] [*Lorho,72*], e o artigo sobre "FORTRAN Conversacional" [*Barron,71*], que ao enfrentarem o mesmo problema propõem soluções distintas entre si e totalmente diversas da proposta deste estudo.

A análise da linguagem LSE 15 é uma proveitosa lição a respeito do projeto de uma linguagem de ensino, especialmente sobre as soluções que devem ser evitadas. A linguagem construída e proposta a partir de sua crítica, constante do apêndice C, e que serviu de exemplo para a utilização do pré-processador, não se propos desde o princípio, a ser mais que um projeto defensável de linguagem de programação do tipo ALGOL. Apesar de contar com características interessantes e desejáveis, apesar de oferecer razoável poder de computação - demonstrado pelo fato de todos os algoritmos apresentados serem através dela expressos - apesar de ser projetada a partir de uma filosofia, a nosso ver, bem fundamentada, apesar de todos esses fatores o projeto de uma linguagem de programação destinada a ensino de computação é um trabalho muito mais amplo e complexo, do qual esta linguagem não é senão um ensaio.

O processo de fragmentação proposto necessita principalmente ser submetido ao teste prático de implementação e utilização. Uma seria restrição a este trabalho é o fato

de se basear grandemente em premissas de certo modo teóricas sobre a maior ou menor facilidade de enfrentar o problema de comunicação homem-máquina: será preciso que os usuários efetivos do sistema acabem por decidir a melhor maneira de utilização. E a experiência demonstra que a capacidade dos programadores agirem de modo absolutamente imprevisível sempre superou a mais fértil imaginação dos projetistas de sistemas.

Por outro lado será necessário enfrentar a tarefa de implementação do pré-processador e do compilador para aferir corretamente quais as vantagens obtidas em termos do aproveitamento da máquina, e, também, para melhor definir qual a configuração do sistema de computação que torna conveniente a adoção do processo. A primeira vista o esquema proposto parece adequado a utilização em minicomputadores, dotados de sistema de interrupção e memória auxiliar, acoplados a terminais de vídeo, especialmente por permitir uma economia de memória utilizada.

Enfim, acreditamos que o trabalho atingiu seu principal e mais modesto objetivo: servir como elemento de abertura de um campo de estudos e como precursor e incentivador de trabalhos posteriores.

BIBLIOGRAFIA - AUTORES

* Ashcroft E. e Manna Z., 1971 - "The translation of "goto" programs into "while" programs" - Comp.Sci. Dep. Rep. CS 188 - Stanford.

* Backus J.W., 1959 -"The Syntax and Semantics of the proposed international algebraic language of the Zurich ACM -GAMM Conference" - Proc.Int.Conf.Proc. UNESCO

* Barron D.W., 1971 - "Approaches to Conversational FORTRAN" - Computer Journal, vol. 14, nº 2, pp.

* Bauer W.F., 1967 - "The Economics of on-line systems" - in "On-line computing", Karplus W.J. (ed) McGraw-Hill.

* Blaizot F. e Blaizot L., 1972 - "CPL/I: Interpreteur Conversationnel" - Th. Docteur-ingenieur Univ. Toulouse.

* Cheatham T.E., 1972 - "The recent evolution of programming languages" - IFIPS 71 - North-Holland.

* Dijkstra E.W., - "Go to statement considered harmful" - Letter to the editor - CACM 3, vol. 11.

* Eklind J.I., Carbonell J.R. e Nickerson R.S., 1968 - "On the psychological importance of time in a time - sharing system" - Human Factors, 10.

* Gilman L. e Rose A.J., 1970 - "APL/360: an interactive approach" - Wiley Int. Ed.

* Hoare C.A.R., 1973 - "Hints on programming languege design" -Comp. Sci. Dep. Rep. no. CS-403 - Stanford.

* Jorrand P. e Hammer M.M., 1969 - "The formal definition of BASEL" - M.C. Associates, Inc.

* Kemeny J.G. e Kurtz T.E., 1971 - "BASIC Programming" - Wiley Int. Ed.

* Knuth D.E., 1972 - "The art of computer programming" - Fundamental Algorithms, vol. 1, pp. 305 - Addison - Wesley.

* Lorho B. e Vatoux M., 1972 - "CPL/I: Analyseur - Générateur - Th. Docteur-Ingenieur - Univ. Toulouse.

* Marks S.L. e Armstrong G.W., 1967 - "The Joss primer" - Rand Memo RM-5220-PR in Infotech (039) - State of the art report 4.

* Martin J. e Norman A.R.D., 1970 - "The computerized Society" -

Prentice-Hall.

* McKeeman W.M., 1965 - "Algorithm 268: ALGOL 60 reference Language editor" - CACM 3, vol. 8

* MITRA-CII, 1972 - "Language Symbolique d'Enseignement" - Manuel d' apresentation 4390 P/FR Comp. Int. pour L'Informatique.

* MITRA-CII, 1973 - "Language Symbolique d'Enseignement" - Manuel d' utilization 4242 E/PR - Comp. Int. pour L'In - formatique.

* Naur P.(ed), 1963 - "Revised report on the algorithmic Language ALGOL 60" - Computer Journal, vol. 5.

* Peterson W.W., Kasami T. e Tokura N., 1973 - "On the capabilities of While, Repeat and Exit Statements " CACM 8, vol. 16.

* Reynolds J.C., 1970 - "GEDANKEN: a simple typeless language based on the principles of completeness and the reference concept" - CACM 5, vol. 13.

* Sammet J.E., 1969 - "Programming language: history and fundamen - tals" - Prentice-Hall.

* Sackman H., 1968 - "Time-sharing versus batch: the experimental

evidence" - Proc. AFIPS SJCC, vol. 32.

* Scowen R.S., Allin D., Hillman A.L., Shimell M., 1971 - "SOAP - a program which documents and edits ALGOL 60 programs" - Computer Journal, vol. 14, no. 2.

* Tast E.A., 1971 - "PPL User's Manual" - Aiken Comp. Laboratory, Harvard in [Cheatham,72].

* Wirth N., 1971 - "The programming language PASCAL" - Acta Informatica, vol. 1, no. 1.

* Wulf W.A., 1971 - "Programming without the "goto"" - IFIPS 71 , TA-3-84.

BIBLIOGRAFIA - LINGUAGENS

* ALGOL 60

Naur P.(ed), 1963 - "Revised report on the algorithmic Language ALGOL 60" - Computer Journal, vol. 5.

* APL

Gilman L. e Rose A.J., 1970 - "APL/360: an interactive approach" Wiley.

* BASEL

Jorrand P. e Hammer M.M., 1969 - "The formal definition of BASEL" M.C. Associates. Inc.

* BASIC

Kemeny J.G. e Kurtz T.E., 1971 - "BASIC programming" - Wiley.

* CPL/I

Lorho B. e Vatoux M., 1972 - "CPL/I: Analyseur-Gēnērateur" - Th. Docteur-Ingenieur - Univ. Toulouse.

* FORTRAN

ASA, 1964 - "ASA Standard FORTRAN" - CACM 7, vol. 10.

* GEDANKEN

Reynolds J.C., 1970 - "GEDANKEN: a simple typeless language based

on the principles of completeness and the reference concept" - CACM 5, vol. 13.

* JOSS

Marks S.L. e Armstrong G.W., 1967 - "The Joss primer" - Rand Memo RM-5220-PR in Infotech 039 - State of the art Report no.4.

* LSE 15

, 1973 - "Language Symbolique d'Enseignement" Manuel d'utilization 4242 E/FR Comp.Int. pour L' Informatique.

* PASCAL

Wirth N., 1971 - "The programming Language PASCAL" - Acta Informatica, vol. 1, no. 1.

* PPL

Tast E.A., 1971 - "PPL User's manual" - Aiken Comp. Laboratory Harvard in [Cheatham,72].

APÊNDICE A

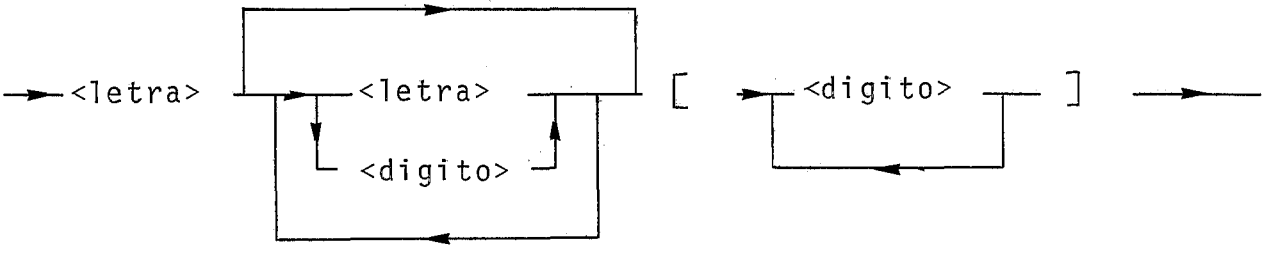
LINGUAGEM DE COMANDO DO MONITOR LSE 15

1. Formalismo para descrição sintática

A descrição da notação utilizada apresentada a seguir refer-se às árvores gramaticais dos apêndices A, B, C e das figuras 2 e 4 do texto principal. Trata-se de uma variação da forma normalizada de Backus [*Backus,59*] cuja principal característica é minimizar o número de variáveis metalinguísticas necessárias. A fórmula que figura abaixo como exemplo ilustrativo utiliza a totalidade das regras desta notação.

Exemplo:

<variável subscrita> :: =

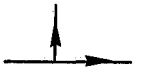


a) sequências de caracteres minúsculos limitados pelos símbolos < > representam variáveis metalinguísticas cujos valores são sequências de símbolos e cujo nome procura des

crever a natureza da variável correspondente. Ex. <letra>;

b) o sinal :: = é um conetivo metalinguísti co e deve ser lido "é definido por";

c) o sinal ⟶ é um conetivo metalinguís tico e deve ser lido "é seguido por";

d) os sinais | e  são conetivos metalinguísticos e devem ser lidos "ou seguido por";

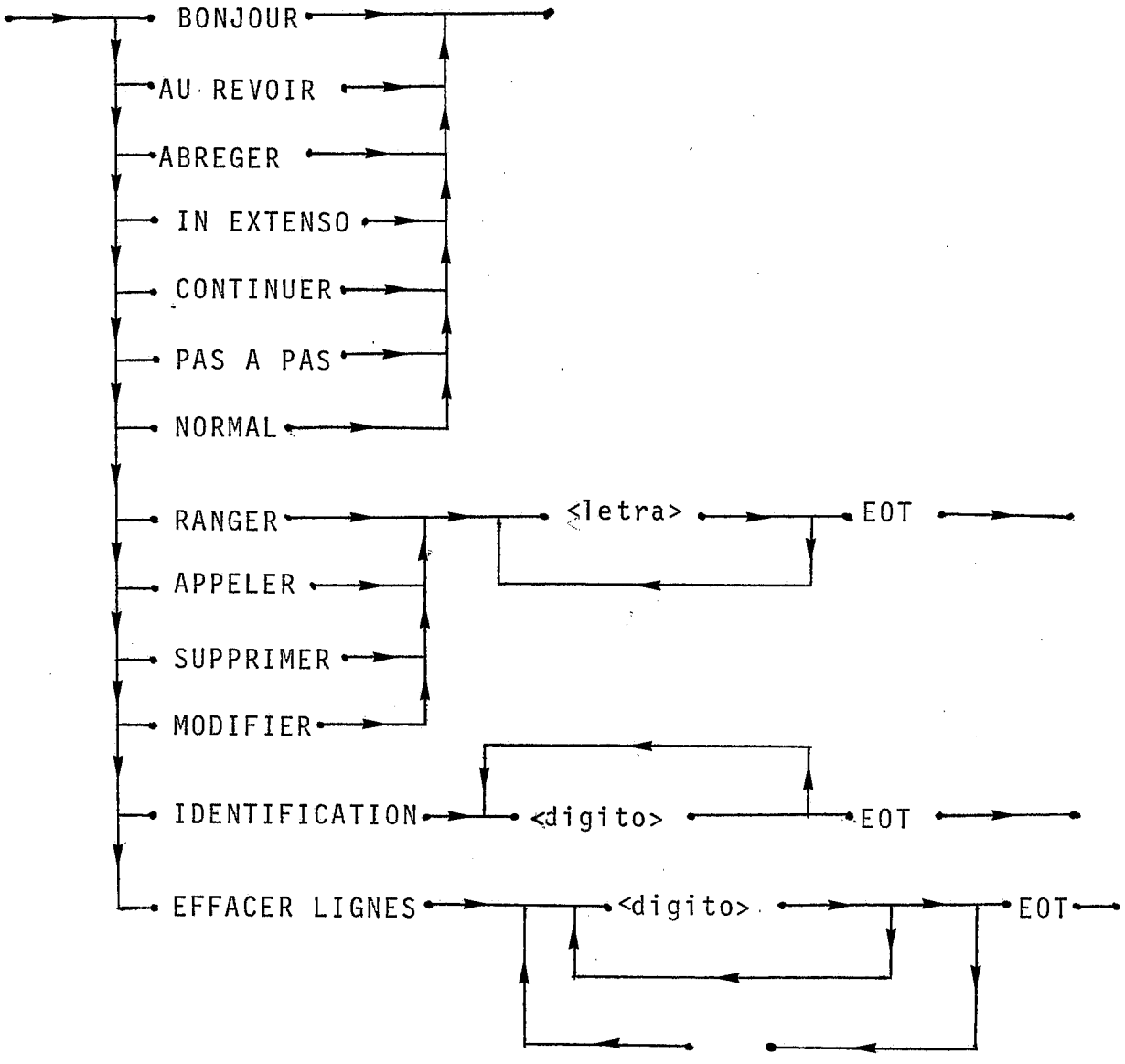
e) qualquer outro sinal de uma fórmula que não seja variável metalinguística ou conetivo denota a si mesmo;

f) as fórmulas independentes são numeradas entre barras. Ex. |1|;

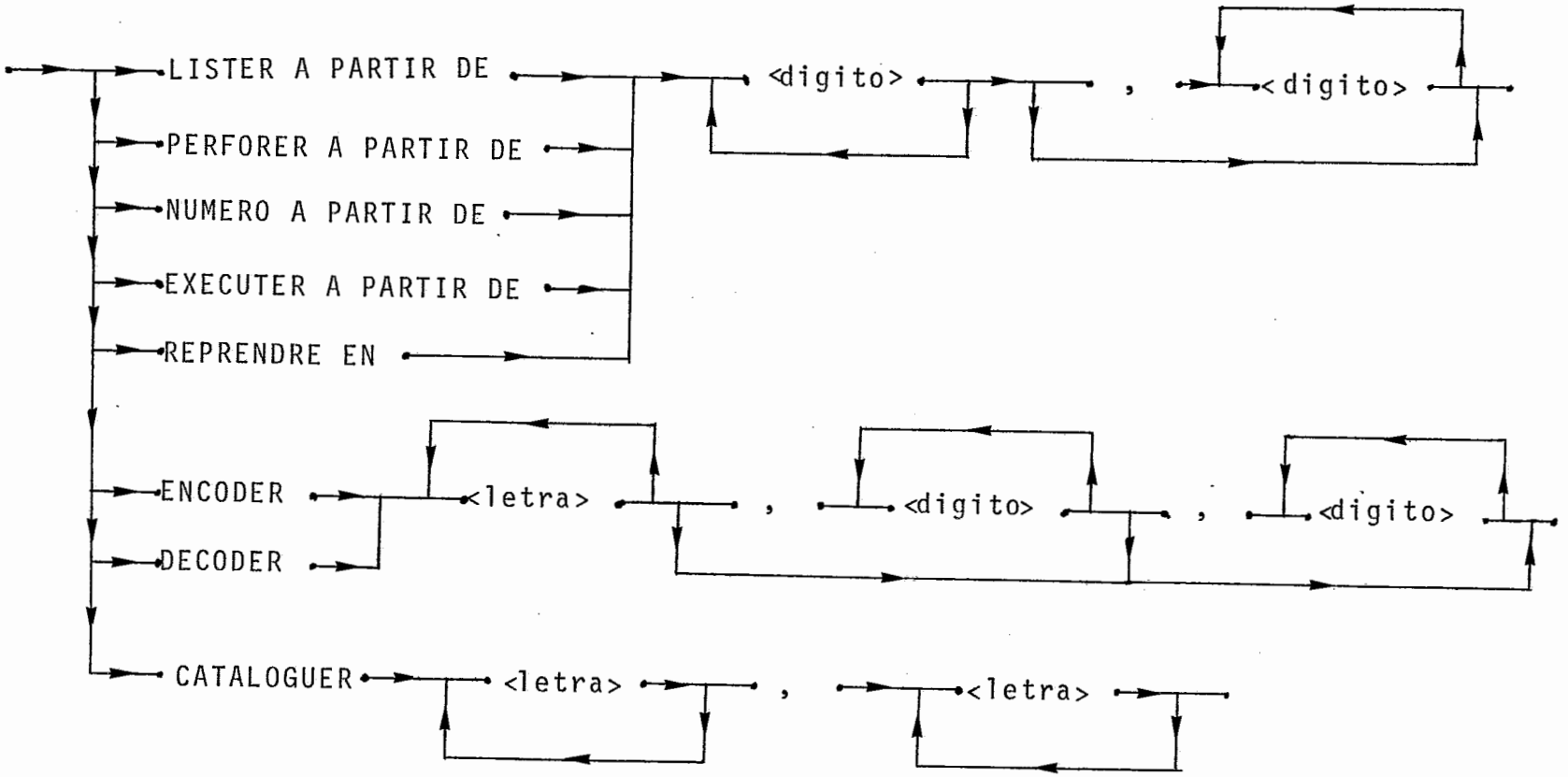
g) duas fórmulas com mesmo número e na mesma árvore são partes de uma fórmula única.

2. Arvore gramatical da linguagem de comando LSE 15

|1| <comando> ::=



|1| <comando> ::=

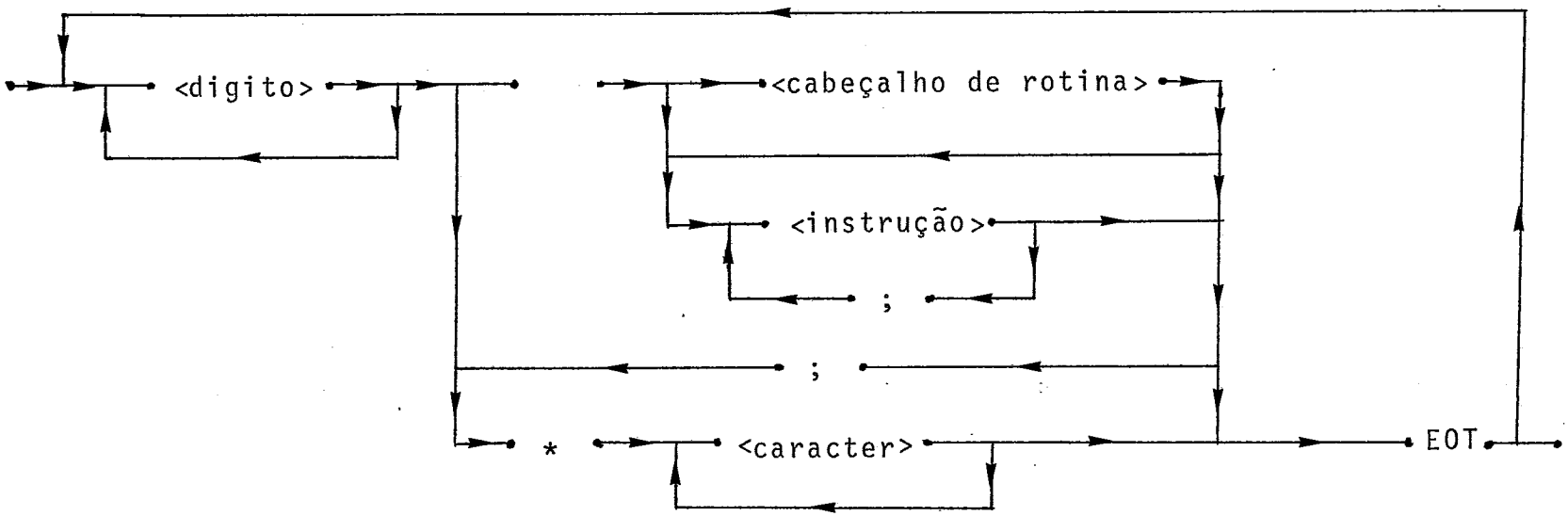


|2| <letra> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

|3| <digito> ::= 0|1|2|3|4|5|6|7|8|9

APENDICE B ARVORE GRAMATICAL DA LINGUAGEM LSE 15

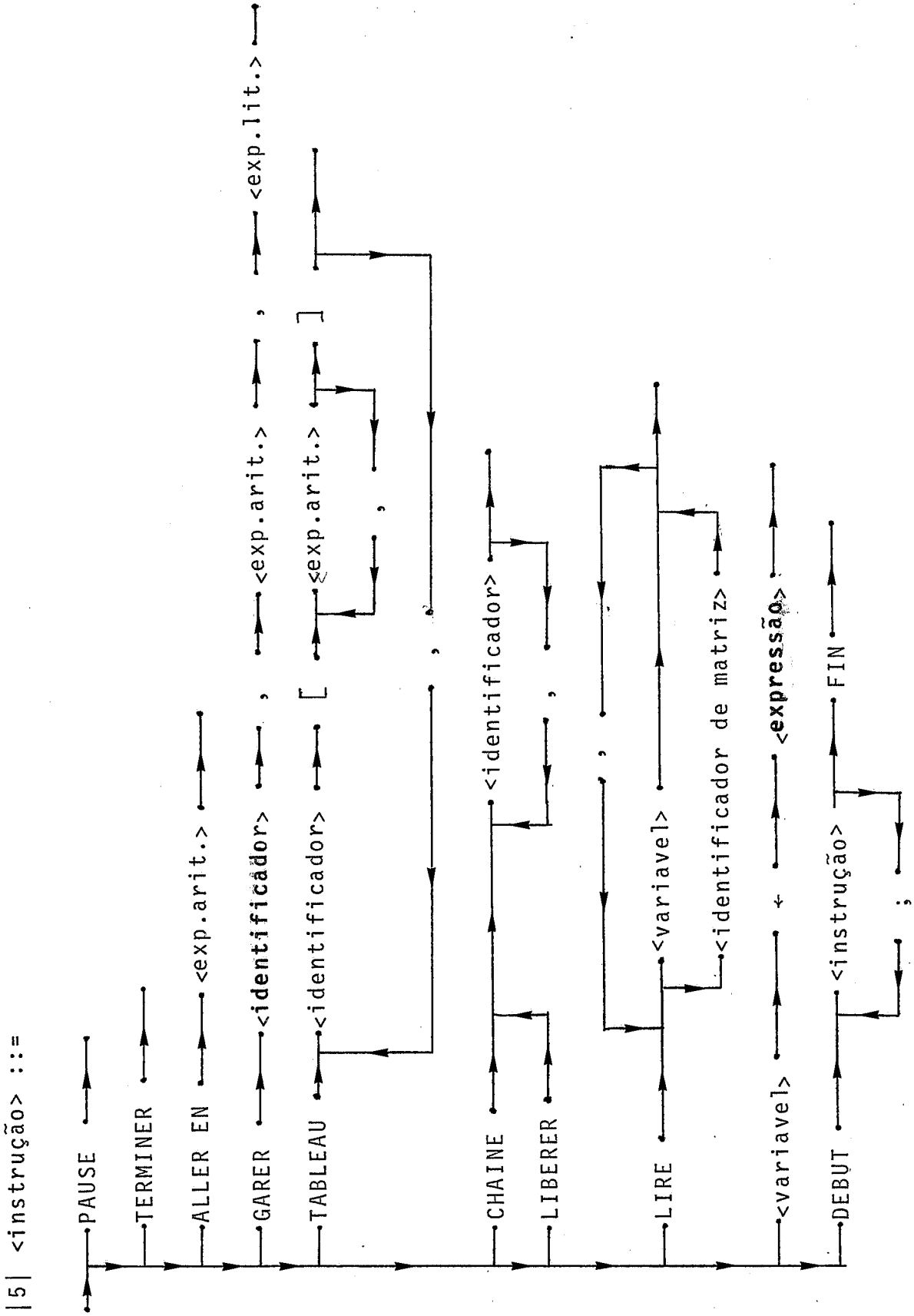
|1| <programa> ::=



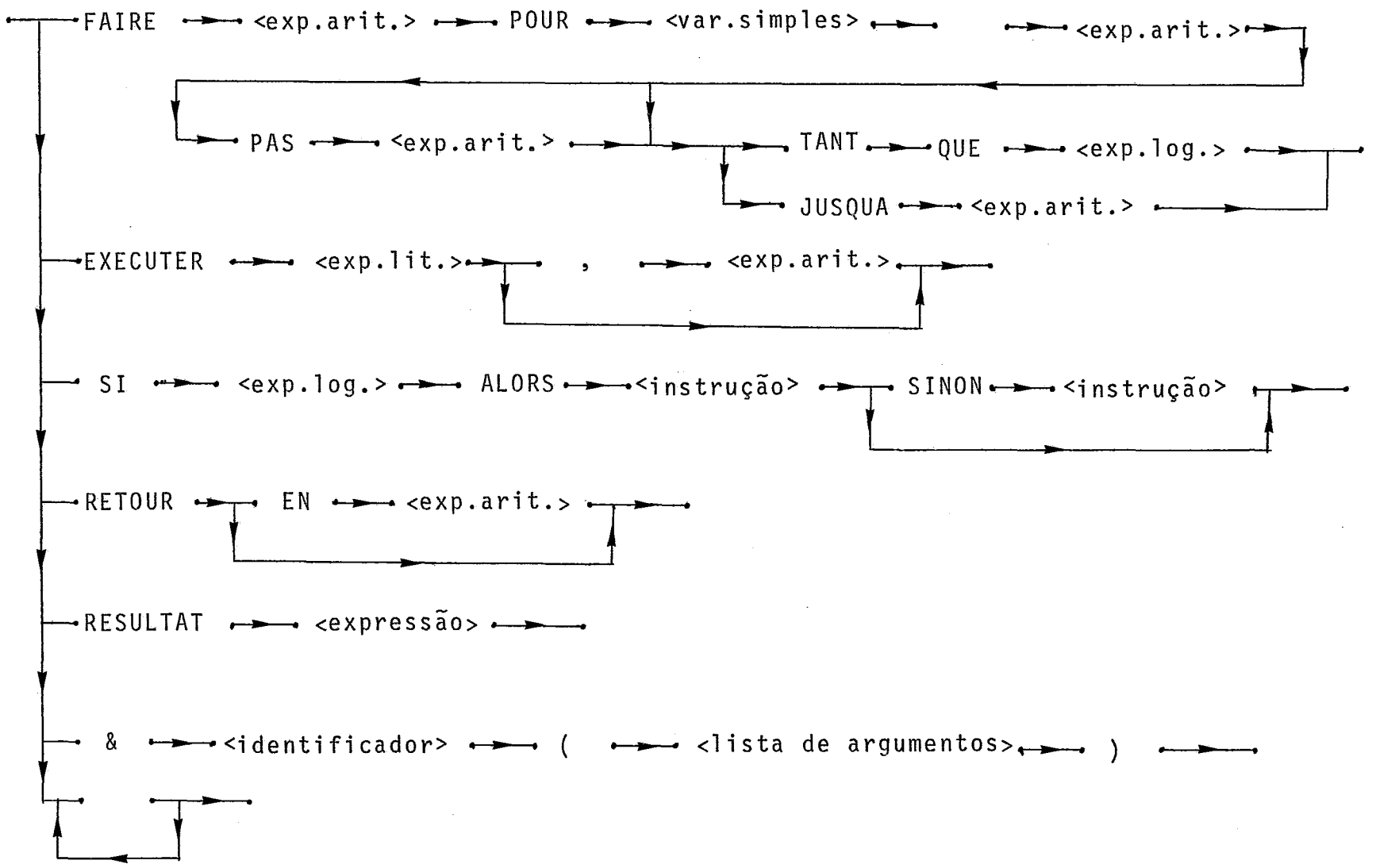
|2| <digito> ::= 0|1|2|3|4|5|6|7|8|9

|3| <caracter> ::= <letra>|<digito>|,|.|:|
;|?|!|'|"| |@|$|%|&|(|)|[|]|←|+|-|
*|/|↑|=|≠|<|>

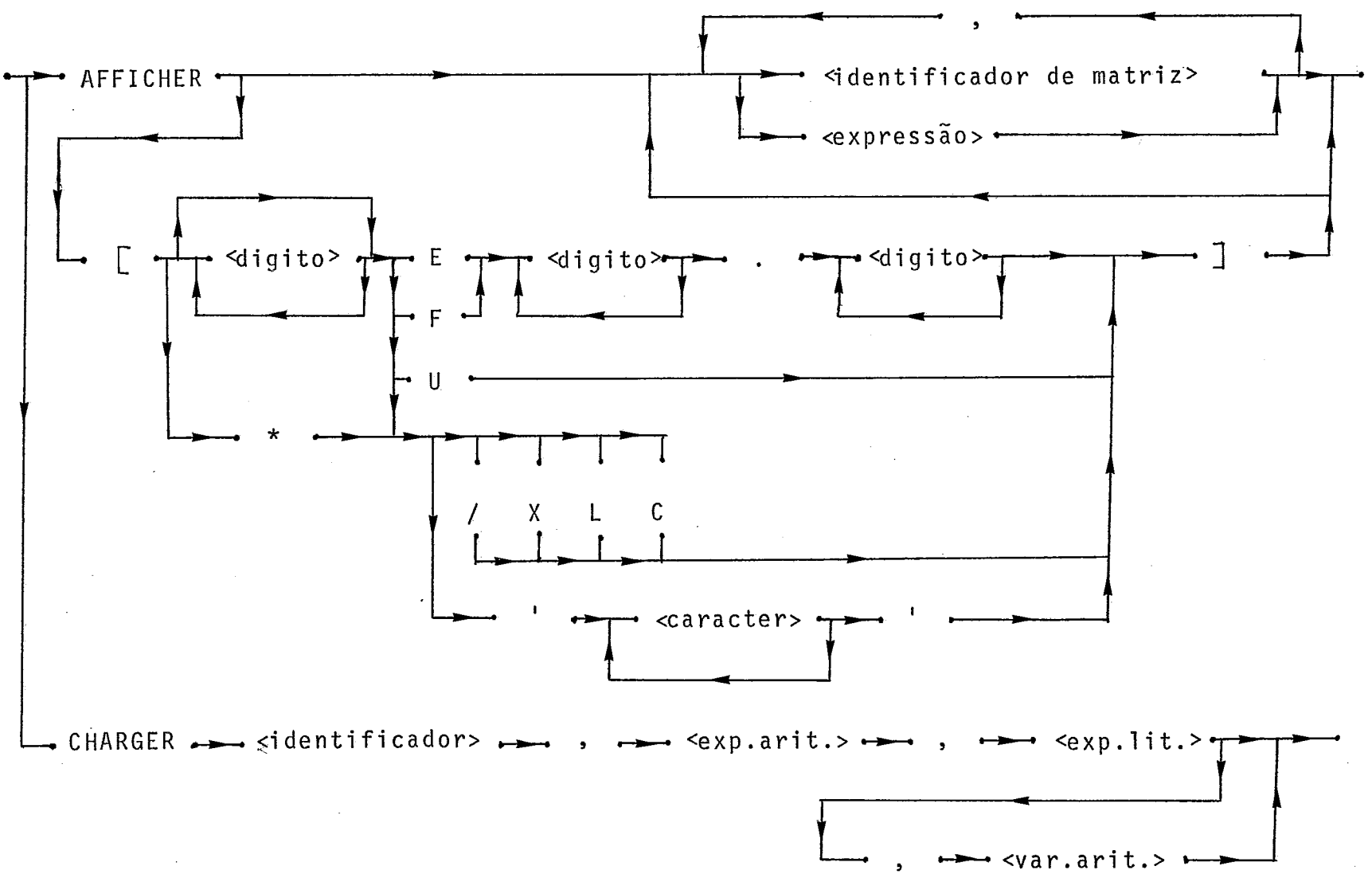
|3| <letra> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z


|5| <instrução> ::=
PAUSE
TERMINER
ALLER EN
<exp.arit.>
GARER
<identificador>
,
<exp.arit.>
,
<exp.lit.>
TABLEAU
<identificador>
[
<exp.arit.>
,
]
,
CHAINE
<identificador>
,
LIBERER
,
LIRE
<variavel>
<identificador de matriz>
<variavel>
←
<expressão>
DEBUT
<instrução>
;
FIN

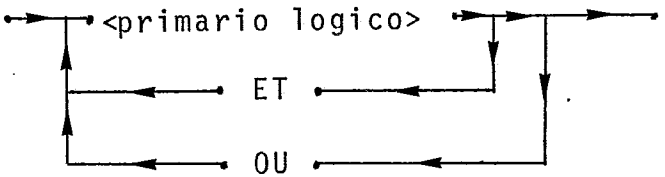
|5| <instrução> ::=


FAIRE
<exp.arit.>
POUR
<var.simples>
<exp.arit.>
PAS
<exp.arit.>
TANT
QUE
<exp.log.>
JUSQUA
<exp.arit.>
EXECUTER
<exp.lit.>
,
<exp.arit.>
SI
<exp.log.>
ALORS
<instrução>
SINON
<instrução>
RETOUR
EN
<exp.arit.>
RESULTAT
<expressão>
&
<identificador>
(
<lista de argumentos>
)

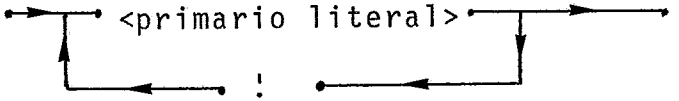
|5| <instrução> ::=


AFFICHER
,
<identificador de matriz>
<expressão>
[
<digito>
E
<digito>
.
<digito>
]
F
U
*
/
X
L
C
'
<caracter>
'
CHARGER
<identificador>
,
<exp.arit.>
,
<exp.lit.>
,
<var.arit.>

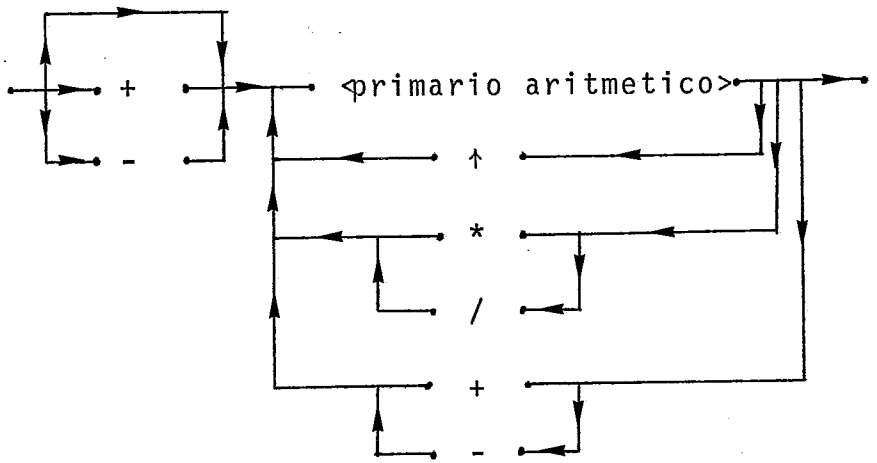
|6| <exp.log.> ::=



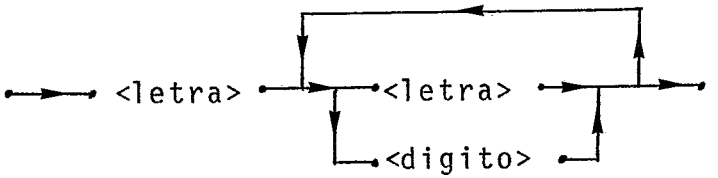
|7| <exp.lit> ::=



|8| <exp.arit.> ::=

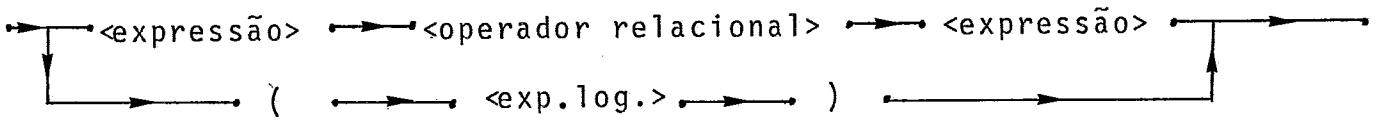


|9| <identificador> ::=

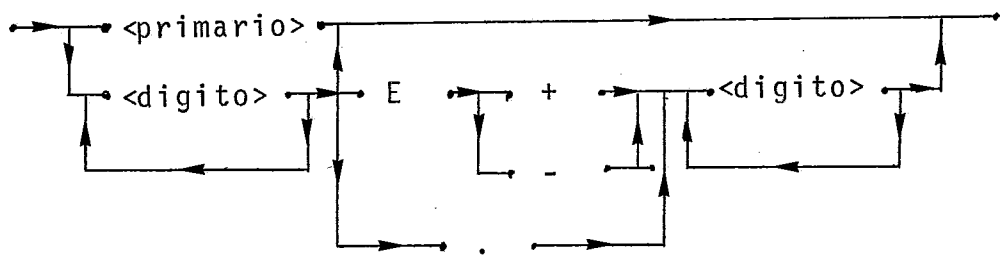


|11| <operador relacional> ::= <|>|=|≠|
<=|>=

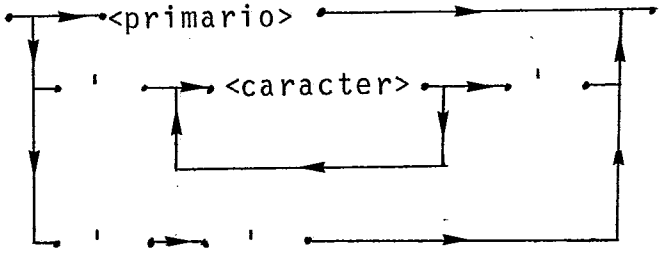
|10| <primario logico> ::=



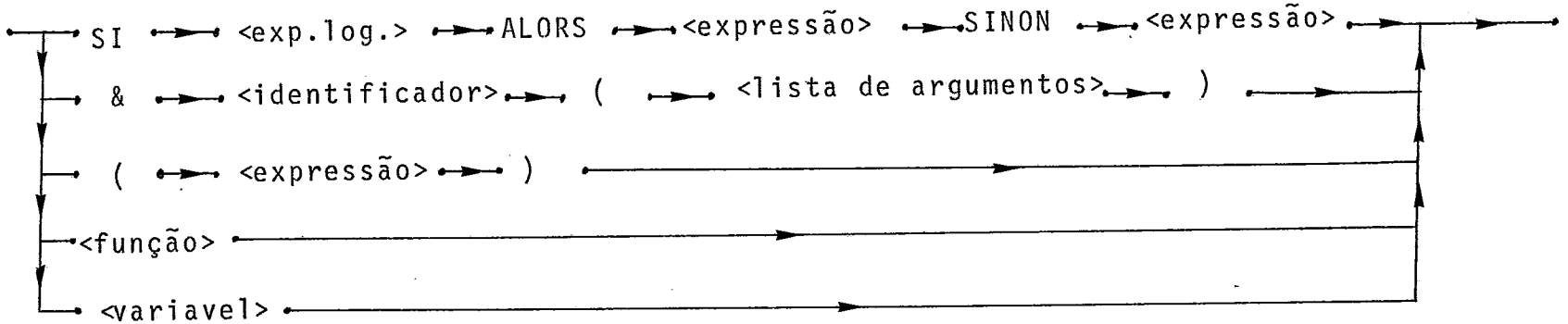
|12| <primario aritmetico> ::=



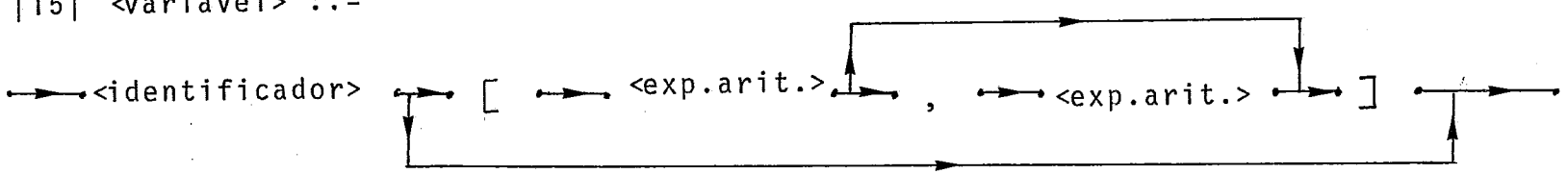
|13| <primario literal> ::=



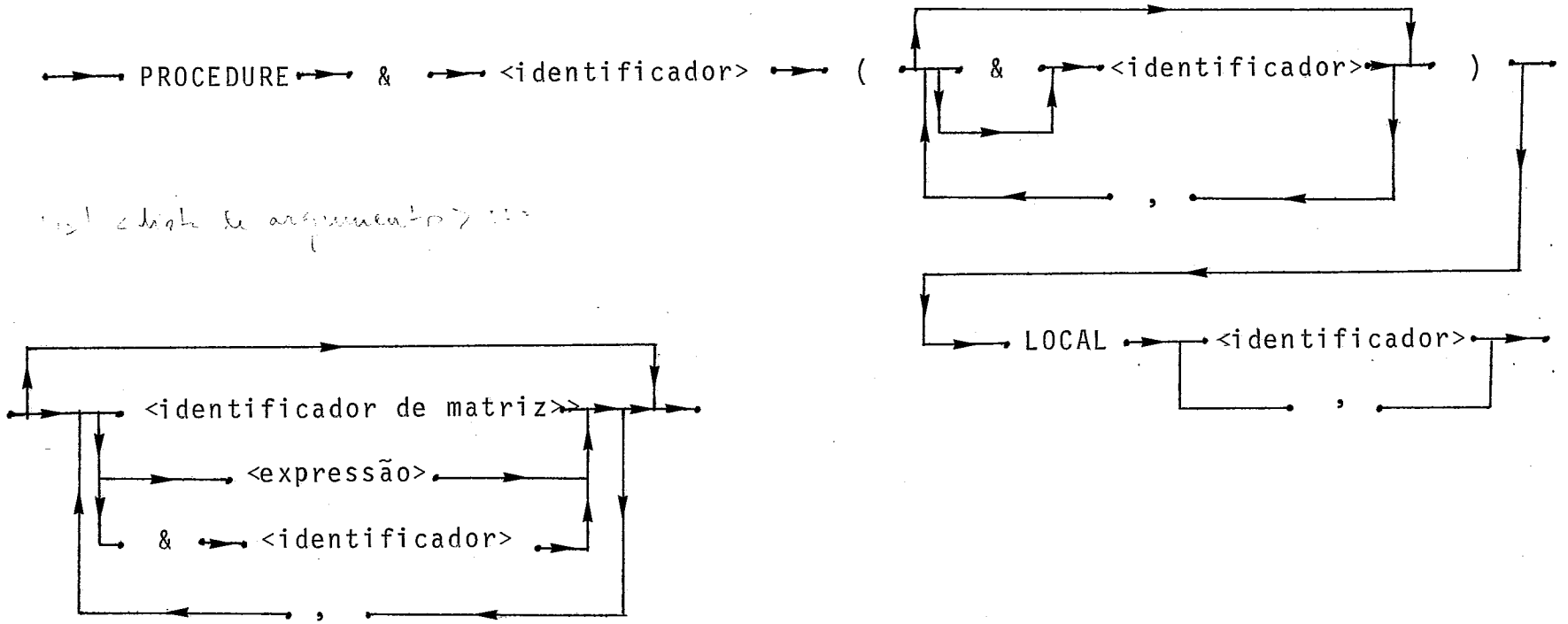
|14| <primario> ::=



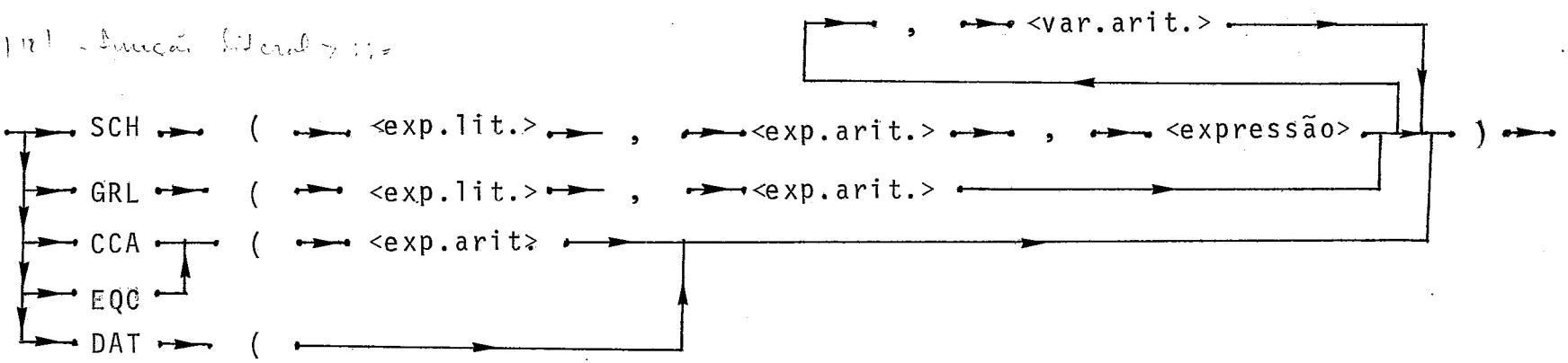
|15| <variavel> ::=



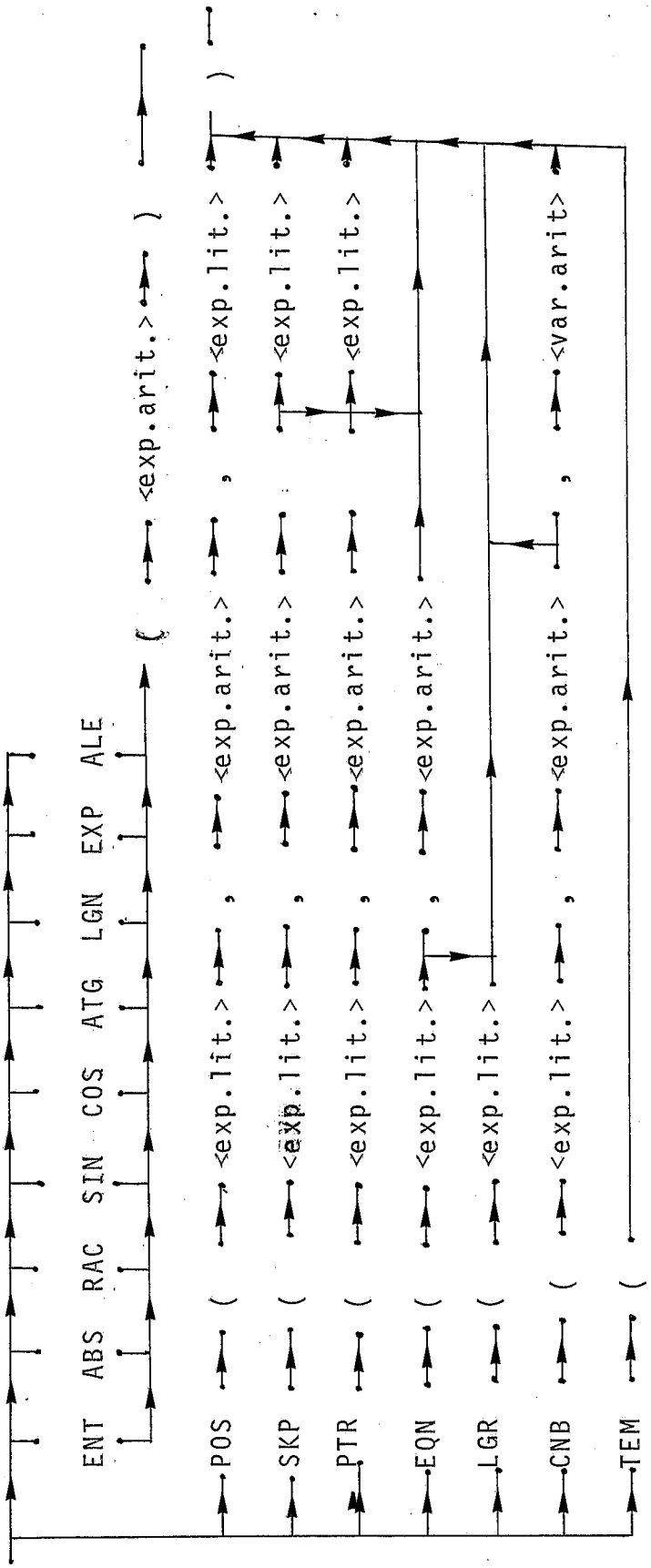
|16| <cabeçalho de rotina> ::=



|17| <função lógica> ::=



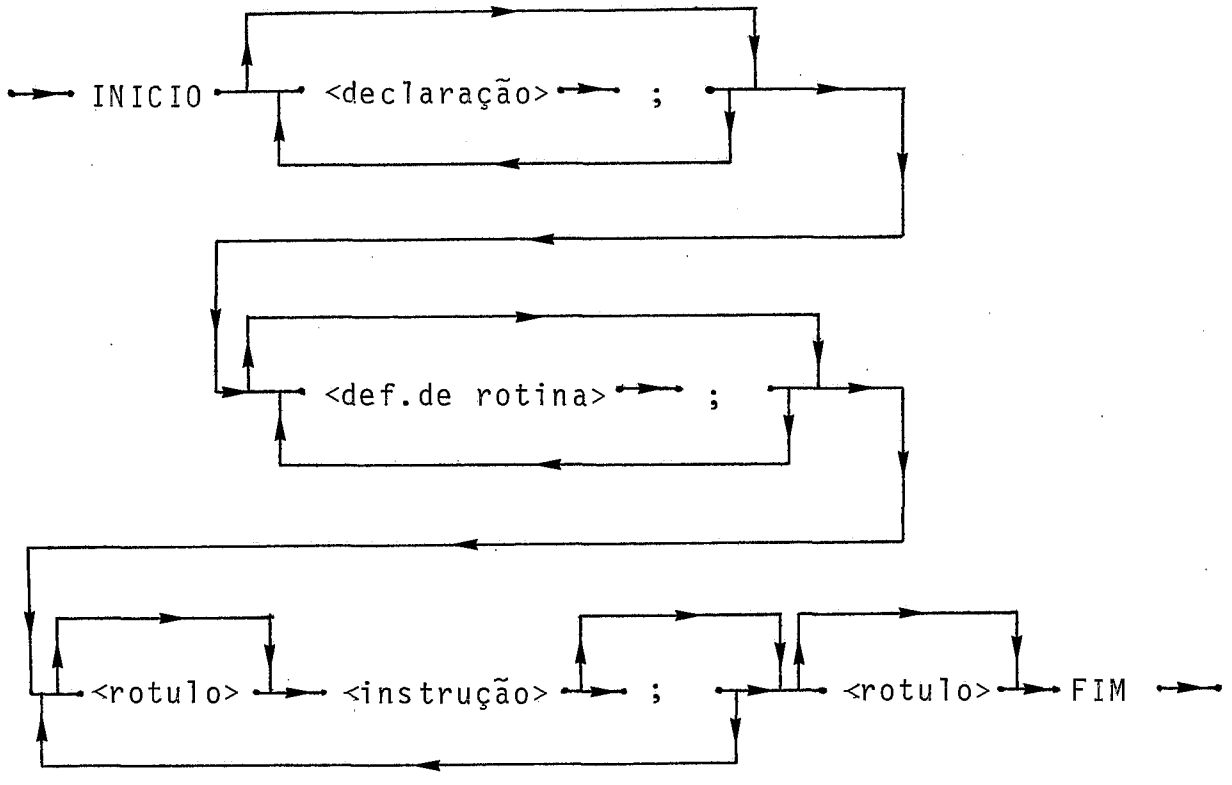
|19| <função aritmetica> ::=



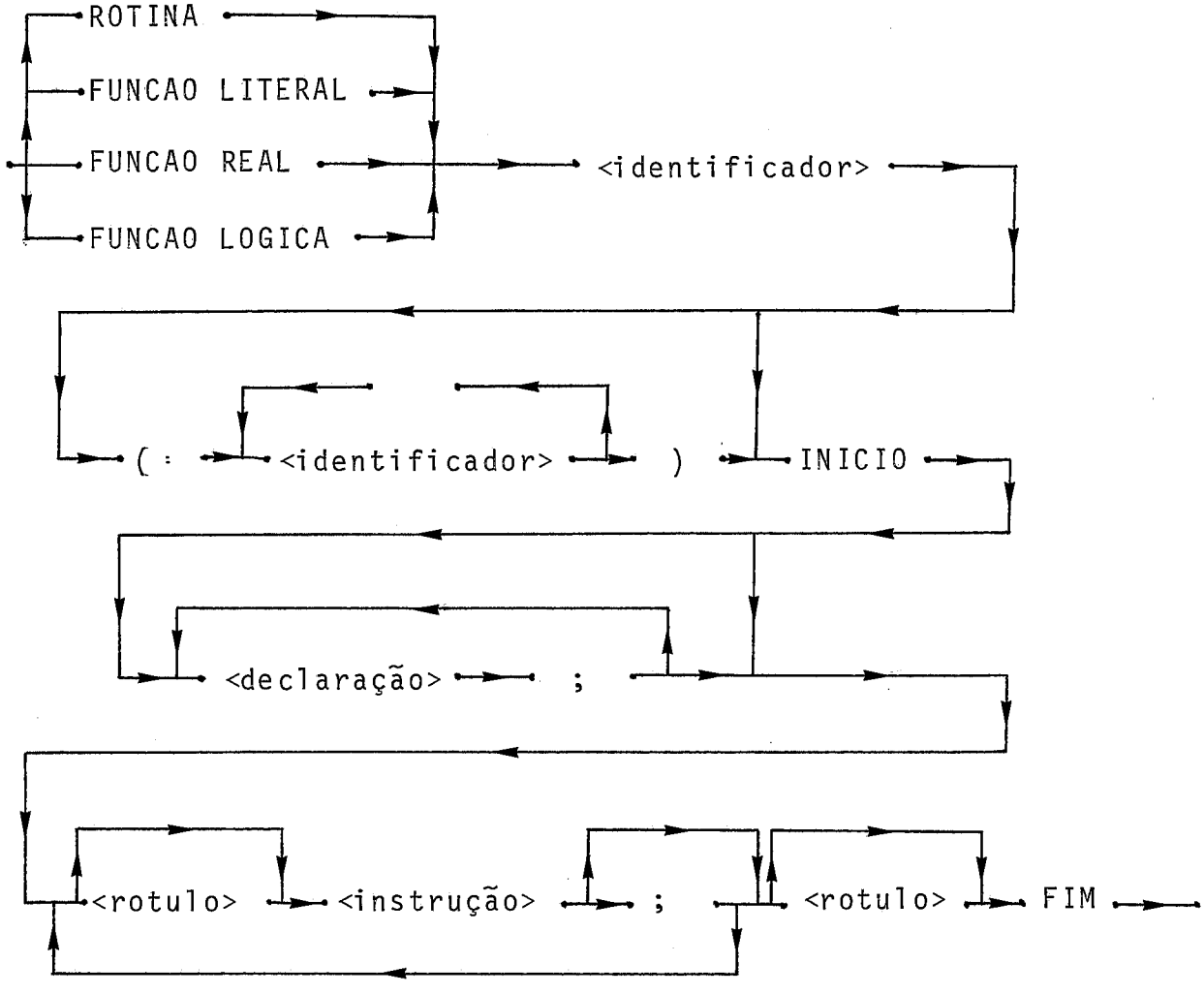
APENDICE C

ARVORE GRAMATICAL DA LINGUAGEM PROPOSTA PARA ENSINO DE COMPUTAÇÃO

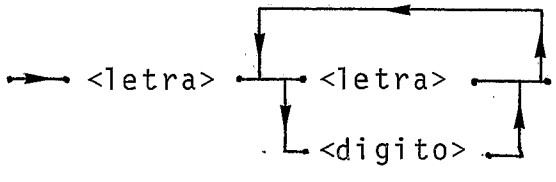
|1| <programa> ::=



|2| <def.de rotina> ::=

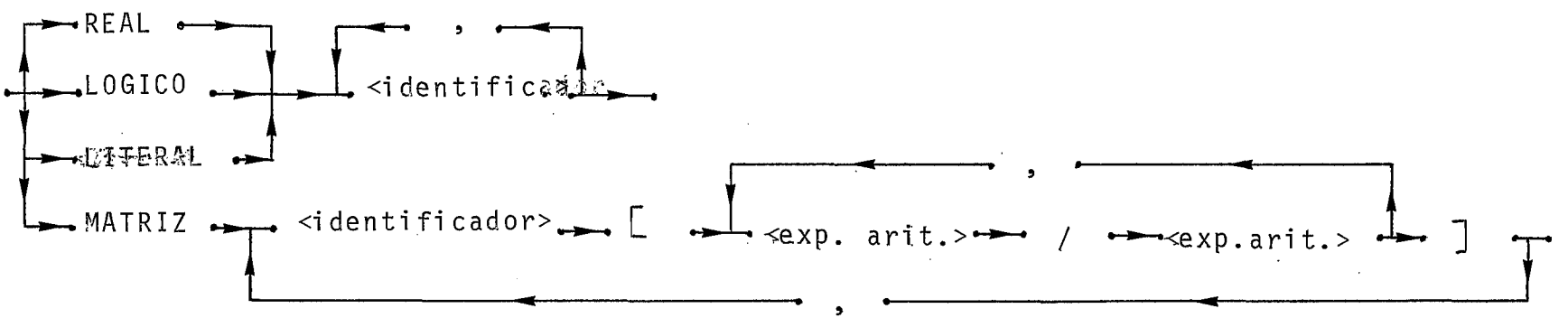


|3| <identificador> ::=

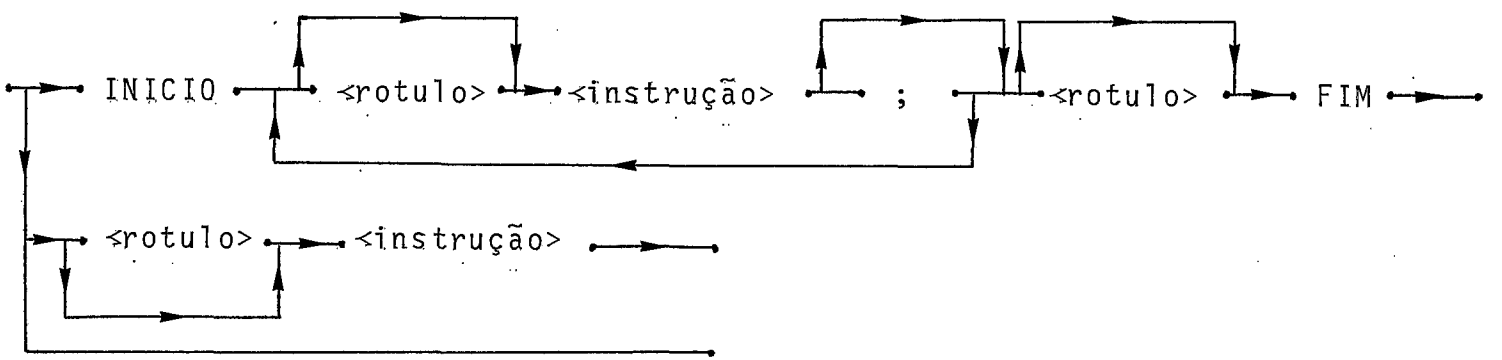


|4| <rotulo> ::=

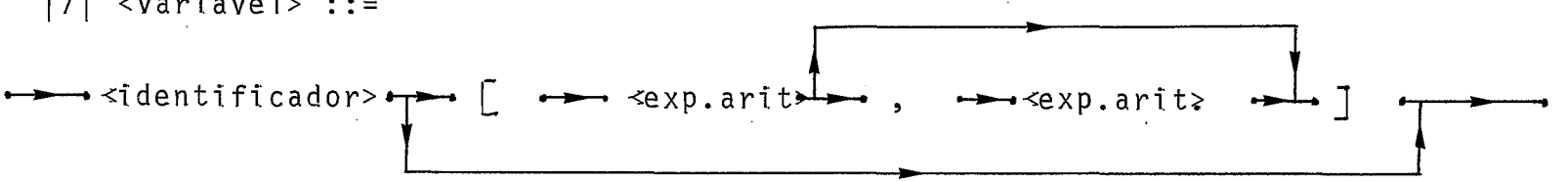
|5| <declaração> ::=



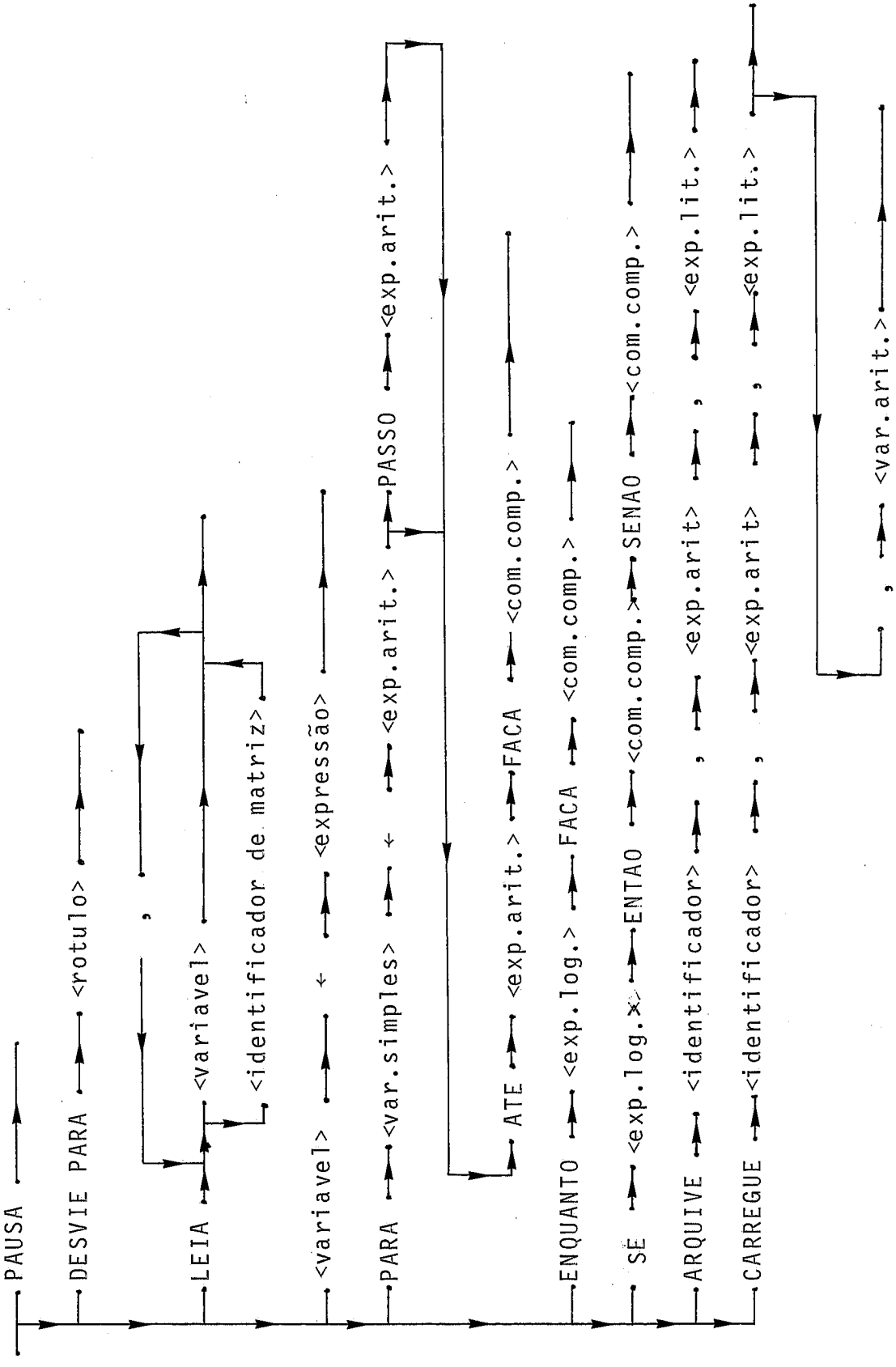
|6| <comando composto> ::=



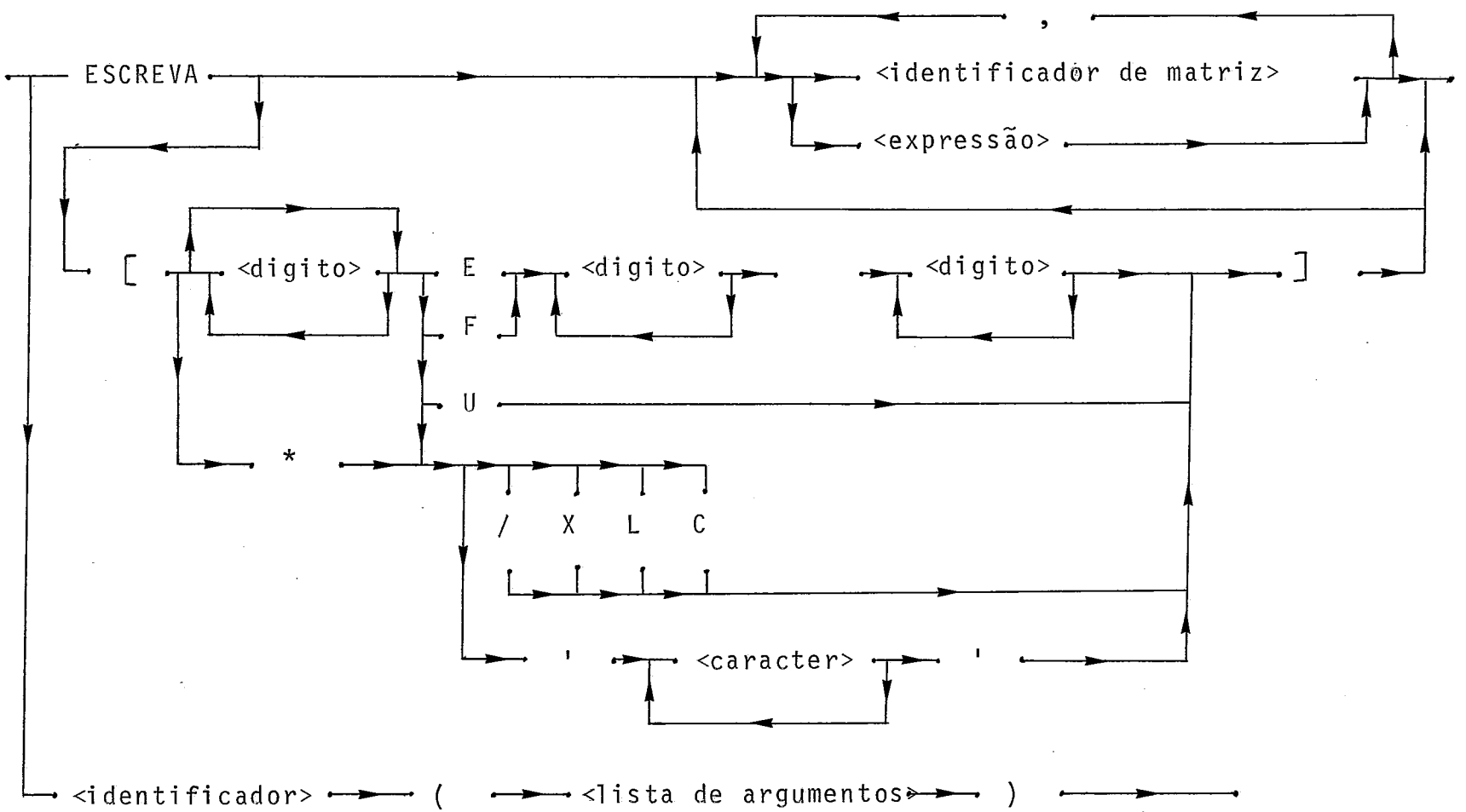
|7| <variavel> ::=



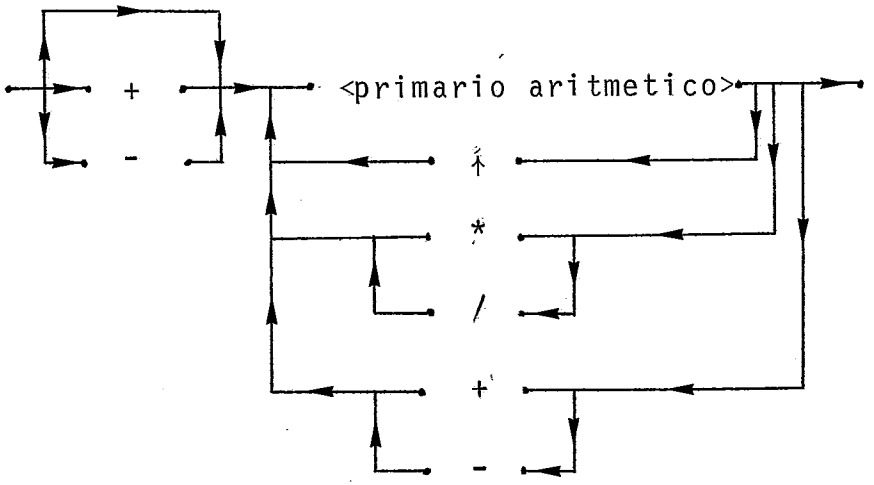
|8| <instrução> ::=


PAUSA
DESVIE PARA
<rotulo>
LEIA
<variavel>
,
<identificador de matriz>
<variavel>
←
<expressão>
PARA
<var.simples>
←
<exp.arit.>
PASSO
<exp.arit.>
ATE
<exp.arit.>
FACA
<com.comp.>
ENQUANTO
<exp.log.>
FACA
<com.comp.>
SE
<exp.log.>
ENTAO
<com.comp.>
SENAO
<com.comp.>
ARQUIVE
<identificador>
,
<exp.arit>
,
<exp.lit.>
CARREGUE
<identificador>
,
<exp.arit>
,
<exp.lit.>
,
<var.arit.>

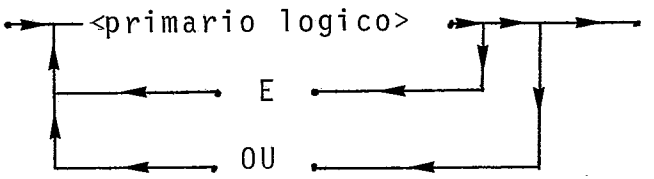
| 8| <instrução> ::=


ESCREVA
,
<identificador de matriz>
<expressão>
[
<digito>
E
F
<digito>
<digito>
]
U
*
/
X
L
C
'
<caracter>
'
<identificador>
(
<lista de argumentos>
)

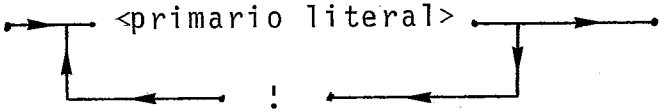
|9| <exp.arit.> ::=



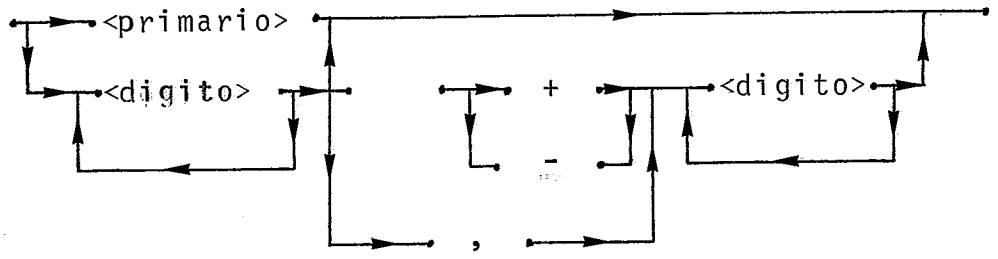
|10| <exp.logica> ::=



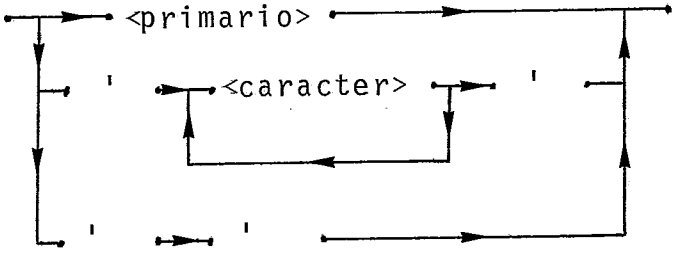
|11| <exp.lit.> ::=



|12| <primario aritmetico> ::=

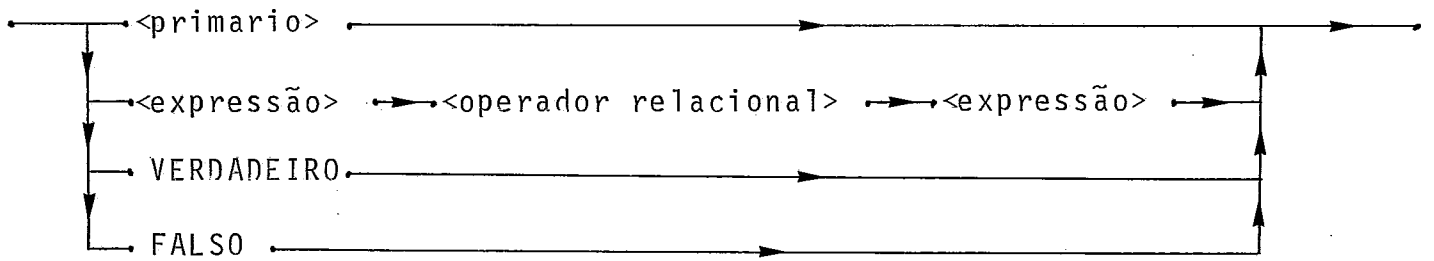


|13| <primario literal> ::=

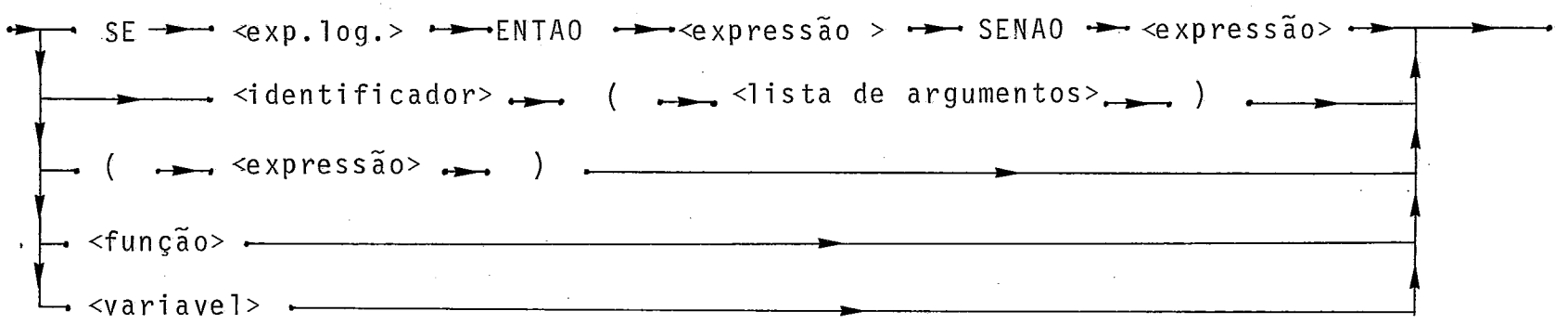


|14| <operador relacional> ::= <|>|=|≠|<=|>=

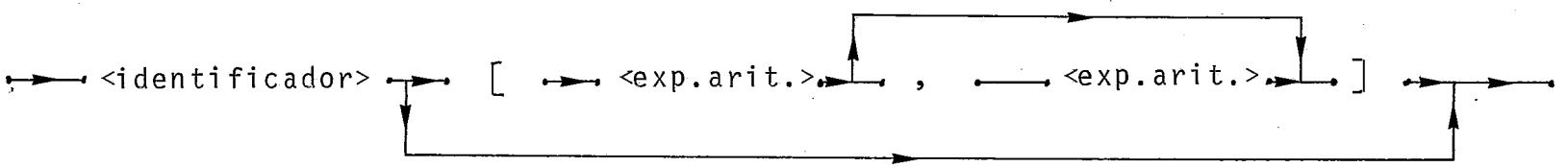
|15| <primario logico> ::=



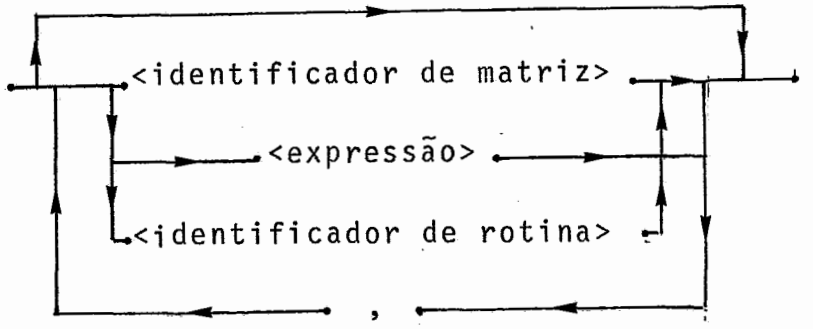
|16| <primario> ::=



|17| <variavel> ::=



|18| <lista de argumentos> ::=

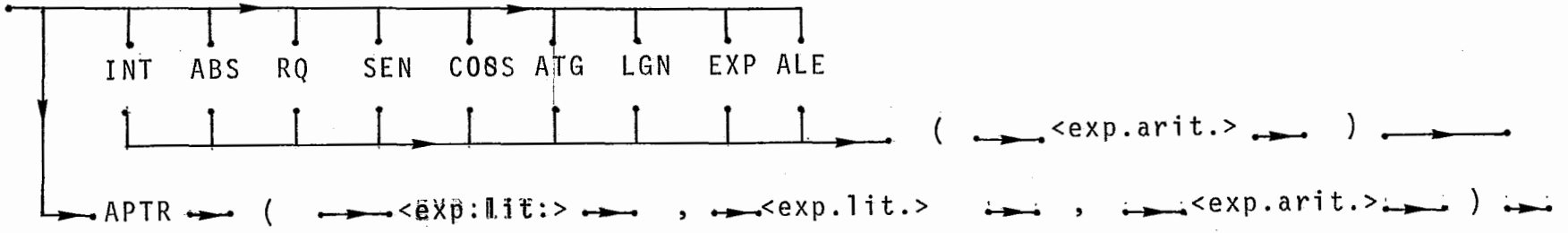
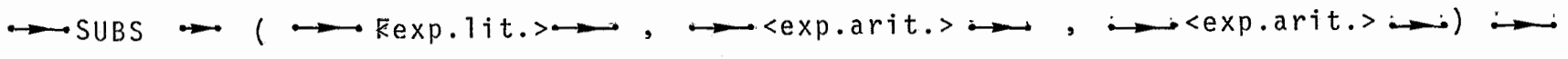


|19| <letra> ::= A|B|C|D|E|F|G|H|I|J|
K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

|20| <digito> ::= 0|1|2|3|4|5|6|7|8|9

|21| <caracter> ::= <letra>|<digito>|
<|>|←|↑|||||,|;|:|'|"| | |$|(|)|
+|-|*|/|≤|≠

|22| <função aritmetica> ::=



|23| <função literal> ::=