PORTABILIDADE DE COMPILADORES

MANOEL MARCILIO SANCHES

DISSERTAÇÃO APRESENTADA AO

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DA

UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE MESTRE EM

MATEMÁTICA APLICADA

ORIENTADOR:

Prof. Dr. Valdemar W. Setzer

São Paulo, Março de 1979

A meus pais

e irmãos

Agradecimentos

Ao Prof. Valdemar W.Setzer, além de excelente orientador, um amigo sempre preocupado com o aspecto humano das coisas.

Aos colegas do Departamento de Matemática Aplicada do IME-USP cujos incentivos muito colaboraram para a realização deste trabalho.

Ao CNPq, FINEP e IBM pelo auxílio concedido.

À Srta. Regina Helena pelo excelente trabalho de datilografia.

São Paulo, Março de 1979

Í N D I C E

CAPÍTULO I - INTRODUÇÃO 1

1. HISTÓRICO. 1
2. CONTEÚDO DESTE TRABALHO. 3
3. NOTAÇÃO UTILIZADA NESTE TRABALHO . 4
4. A TÉCNICA DE "BOOTSTRAPPING" . . . 6

CAPÍTULO II - PORTABILIDADE DE PROGRAMAS 9

1. PORTABILIDADE USANDO LINGUAGENS DE ALTO NÍVEL 10
2. PORTABILIDADE USANDO LINGUAGENS INTERMEDIÁRIAS 15
3. CONCLUSÃO. 40

CAPÍTULO III - PORTABILIDADE DE COMPILADORES. 41

1. O PROCESSO DE COMPILAÇÃO 43
2. ESTRUTURA DE UM COMPILADOR PORTÁTIL. 49
3. A IMPLEMENTAÇÃO DE UM COMPILADOR PORTÁTIL 63
4. ALGUNS COMPILADORES PORTÁTEIS. . . 68
5. CONCLUSÃO. 74

CAPÍTULO IV - A PORTABILIDADE DO COMPILADOR PARA A LINGUAGEM LEAL 75

1. A LINGUAGEM LEAL 76
2. A ESTRUTURA DO COMPILADOR. 78
3. A LINGUAGEM INTERMEDIÁRIA PARA GERAÇÃO DE CÓDIGO. 81
4. O TRANSPORTE DO COMPILADOR 103
5. CONCLUSÕES E JUSTIFICATIVAS. . . . 107

APÊNDICE I DIAGRAMAS SINTÁTICOS DA LEAL 111

APÊNDICE II 115

REFERÊNCIAS BIBLIOGRÁFICAS. 120

CAPÍTULO I

INTRODUÇÃO

1 - HISTÓRICO

A preocupação em construir-se programas que pudessem ser facilmente implementados em vários computadores surgiu com a própria diversificação das máquinas construídas. De fato, o advento das linguagens de alto nível acenava com a possibilidade de programas totalmente independentes de máquina.

A primeira publicação por nós conhecida que faz referência a este problema foi apresentada em 1958 pelo "Share Ad-Hoc Committee on Universal Languages" /S58/. O seguinte problema foi apresentado:

"Implementar n linguagens $L_1, L_2, \ldots, L_n$ em m computadores $M_1, M_2, \ldots, M_m$ sem construir $m \times n$ tradutores".

A solução apresentada foi a seguinte:

"Projetar-se uma Linguagem Intermediária Universal(*UNCOL*-Universal Computer Oriented Language) e construir-se n tradutores $T_1, T_2, \ldots, T_n$ de cada uma das linguagens L_i para a *UNCOL* escritos em *UNCOL*. Construir-se m tradutores $T'_1, T'_2, \ldots, T'_m$ da *UNCOL* para a linguagem de máquina LM_j de cada um dos computadores M_j".

Indicando por (A, B) um programa A escrito na linguagem B, um programa P escrito na linguagem L_i seria executado no computador M_j por:

$$(P,\ L_i) \xrightarrow{T_i} (P,\ UNCOL) \xrightarrow{T'_j} (P,\ LM_j) \longrightarrow \textit{execução em } M_j$$

Assim, o problema foi teoricamente resolvido com $m+n$ tradutores.

Já em 1958 uma primeira proposta para uma *UNCOL* foi dada por Conway /C58/. Em 1961 Steel /S61b/ descreve a viabilidade de uma *UNCOL*.

Devido a diversificação das linguagens e dos computadores ficou claro que as propriedades que tal linguagem universal deveria ter, isto é, adaptar-se a qualquer linguagem e qualquer computador, eram genéricas demais.

No entanto, um programa que possa ser facilmente transportável de uma máquina para outra é bastante desejável economicamente. Assim, outras técnicas foram desenvolvidas.

Essa propriedade de um programa de ser facilmente transportável é chamada *portabilidade* ("portability") e um programa com esta propriedade é chamado um *programa portátil* ("portable program").

Estamos interessados no caso de programas compiladores. Não conhecemos o primeiro compilador portátil construído. O primeiro bastante conhecido parece-nos ser o compilador da linguagem BCPL construído por Richards em 1969 que hoje se encontra implementado em várias máquinas e que pode ser implementado segundo o autor em cerca de 2 homens-meses.

2 - CONTEÚDO DESTE TRABALHO

Um compilador C para u'a máquina M1, é em geral um programa gerando linguagem de máquina de M1. Assim transferir este compilador de M1 para outra máquina M2 não significa simplesmente fazer com que possa ser executado em M2. É necessário modificar também a geração de linguagem de máquina de M1 para linguagem de máquina de M2.

Portanto, para que um compilador seja portátil, é necessário que haja tanto a portabilidade do programa compilador, quanto facilidades para a modificação do código gerado.

Na verdade, o código gerado também é um programa e algumas técnicas usadas na portabilidade de programas podem ser usadas com o mesmo objetivo para o código gerado, como veremos no decorrer do trabalho.

Assim, no capítulo II preocupamo-nos apenas com a portabilidade de programas com o objetivo de usar os elementos ali estudados no capítulo seguinte. No capítulo III estudamos especificamente a portabilidade de compiladores o que constitui o objetivo final deste trabalho.

Finalmente, no capítulo IV apresentamos uma aplicação do estudo que fizemos sobre portabilidade, com o projeto e construção de um compilador portátil para a linguagem LEAL.

Neste texto, as construções feitas em alguma linguagem de programas estão em itálico. As palavras e frases em inglês estão entre aspas e as palavras reservadas da linguagem LEAL estão sublinhadas. As referências à itens de outros capítulos são feitas mencionando explicitamente o capítulo. Assim 1.2.3 refere-se ao item 1.2.3 deste capítulo e II.1.2.3 refere-se ao item 1.2.3 do Capítulo II.

3 - NOTAÇÃO UTILIZADA NESTE TRABALHO

A representação da interação de um programa com tradutores, interpretadores e sua execução, motivou o aparecimento de várias notações para representar o processo /S58/, /B61/, /ES70/ e /R77b/.

Usaremos neste trabalho a notação denominada T-diagramas apresentada por Bratman /B61/ e generalizada por Earley-Sturgis /ES70/.

Rosin /R77b/apresenta uma notação mais completa no sentido de conseguir representar de maneira mais clara processos mais complexos. No nosso caso particular os processos são relativamente simples e a notação adotada, que descrevemos a seguir parece-nos a mais indicada.

Nos símbolos descritos em a, b e c abaixo, a letra i indica um número que identifica o particular símbolo onde se encontra. Essa identificação foi por nós introduzida para melhor podermos fazer referência a determinados elementos que ocorrem nos diagramas do texto e será usada opcionalmente.

a) P / i / L — representa um programa P escrito na linguagem L.

b) L1 $\overset{i}{\rightarrow}$ L2 / L3 — representa um tradutor da linguagem L1 para a linguagem L2 escrito na linguagem L3.

c) L1 / i / L2 — representa um interpretador da linguagem L1 escrito na linguagem L2.

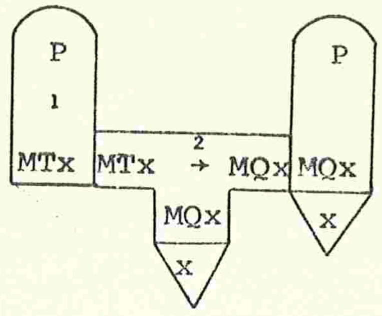
d) x — representa o computador x.

e) MTx — representa a linguagem de montagem do computador x.

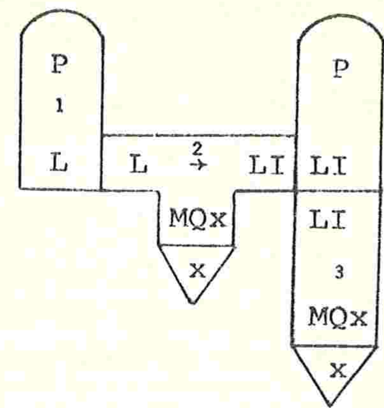
f) MQx representar a linguagem de máquina do computador x.

Os elementos descritos em a, b, c e d são combinados de modo a representar um processo como veremos nos exemplos que seguem.

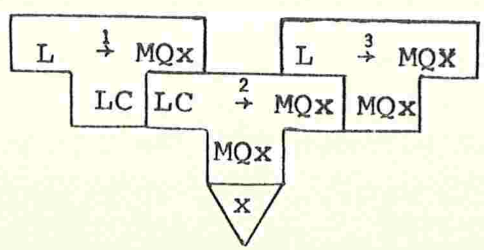
1º) Tradução do programa P escrito em linguagem de montagem (1) pelo montador do computador x (2) e sua execução em x.



2º) Um programa P na linguagem L (1) é traduzido para LI pelo compilador de L para LI em x (2) e executado pelo interpretador de LI em x (3).



3º) Um compilador de L escrito em LC (1) traduzido pelo compilador de LC em x (2), obtendo-se um novo compilador (3).



4 - A TÉCNICA DE "BOOTSTRAPPING"

É uma técnica utilizada na implementação de compiladores. Supondo que desejamos obter um compilador da linguagem L na máquina x

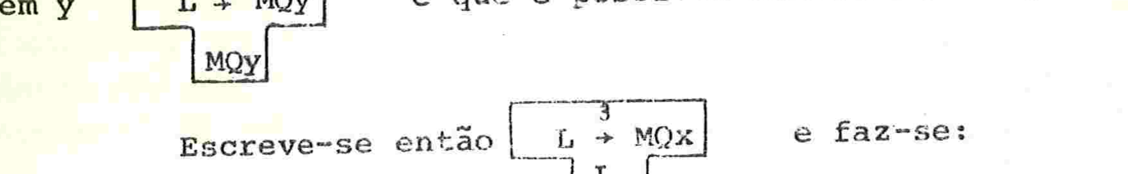
L $\overset{1}{\rightarrow}$ MQx
MQx

,dois tipos básicos de "bootstrapping" são possíveis:

1º) "Half-bootstrapping" ou "Pushing"

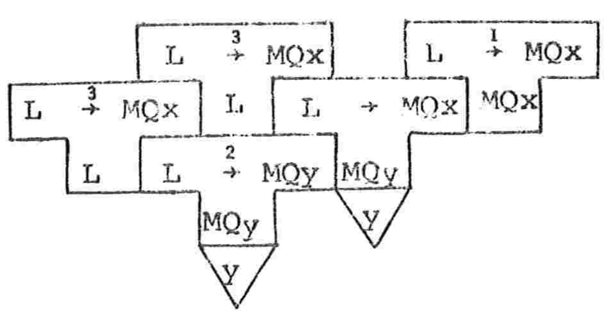
É caracterizado pela utilização de uma máquina auxiliar y.

Para ilustrar, suponhamos que temos um compilador de L em y



e que é possível escrever o compilador em L.

Escreve-se então e faz-se:



Uma das primeiras utilizações desta técnica foi feita por Halstead /H62/ para construir compiladores da linguagem NELIAC para os computadores CDC-1604, Burroughs 220 e IBM-704 e 709 usando como computador auxiliar o UNIVAC M-460.

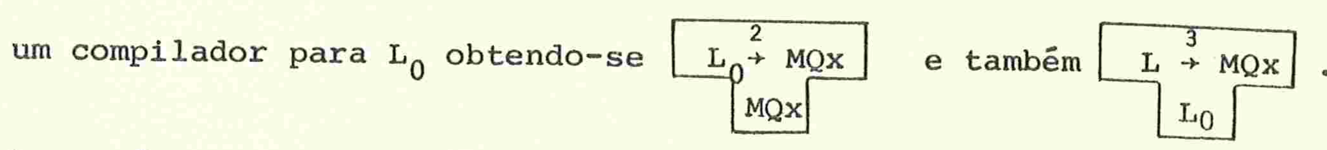
2º) "Full-bootstrapping" ou "Pulling"

Apenas a máquina x é utilizada no processo.

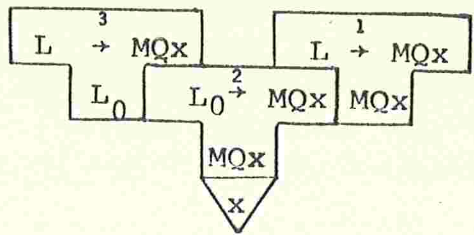
Para ilustrar, suponhamos que o compilador para L possa ser escrito num subconjunto L_0 de L.

Escreve-se então em alguma linguagem já existente em x

um compilador para L_0 obtendo-se e também .

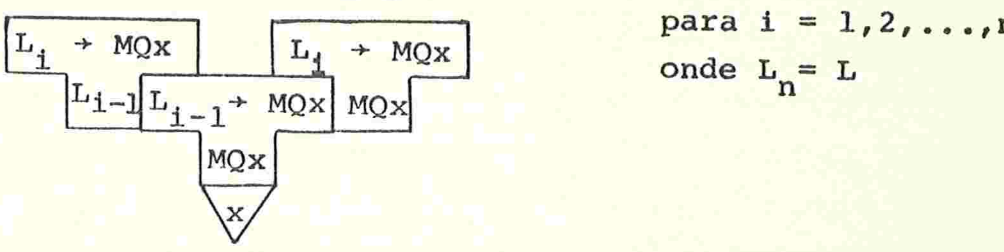


Agora faz-se



Uma das primeiras utilizações desta técnica foi feita também por Halstead /H62/ para o primeiro compilador da linguagem NELIAC no computador UNIVAC M-460, onde o compilador para L_0 foi escrito diretamente na linguagem de máquina deste computador.

Uma generalização desta técnica pode ser utilizada se houver necessidade extendendo aos poucos o subconjunto da linguagem utilizada. Supondo que o compilador para o subconjunto L_i possa ser escrito no subconjunto L_{i-1} faz-se



para $i = 1,2,\ldots,n$
onde $L_n = L$

Esta técnica foi utilizada para a implementação do primeiro compilador para a linguagem PASCAL /W71c/ e será melhor descrita em III.4.2.

CAPÍTULO II

PORTABILIDADE DE PROGRAMAS

Segundo Poole /P74/ a portabilidade de um programa é uma medida da facilidade com que este programa pode ser transferido de uma máquina M1 para outra máquina M2.

Não cremos que exista, até o momento, uma definição ou mesmo uma medida de portabilidade de programas que seja satisfatória em todos os sentidos. Por exemplo, pode ser que seja relativamente fácil transferir um programa de M1 para M2, mas difícil de M1 para M3. Este fato pode ocorrer quando M1 e M2 possuem aproximadamente as mesmas características, enquanto que M3 é bastante diferente de M1.

É possível no entanto, após algumas experiências avaliar a portabilidade de um programa como aponta Waite /W76/: sendo T a "quantidade de esforço" necessária para transferir um programa portátil de M1 para M2, e E a "quantidade de esforço" para construir-se todo o programa em M2, então E/T denota o fator de portabilidade. A "quantidade de esforço" pode ser medida em termos de homens-anos ou homens-meses, embora não seja esta medida representativa senão após algumas experiências pois por exemplo, se 3 homens levam 1 mês para completar uma implementação, não se pode concluir que 1 homem levará 3 meses.

Na seção 1 veremos o uso de Linguagens de Alto Nível para a portabilidade de programas, na seção 2 o uso de Linguagens Intermediárias e na seção 3 algumas conclusões.

1 - PORTABILIDADE USANDO LINGUAGENS DE ALTO NÍVEL

A existência de certas linguagens de alto nível(FORTRAN, COBOL, ALGOL, etc...) na maioria dos sistemas, pode fazer com que a programação nestas linguagens produza imediatamente a portabilidade dos programas. Na verdade, a independência de máquina é um dos objetivos das linguagens de Alto Nível.

No entanto, não podemos dizer hoje que este objetivo foi alcançado. Dois problemas ainda permanecem quanto ao uso generalizado de Linguagens de Alto Nível:

1º) Existem aplicações para os quais as linguagens mais populares existentes não são muito adequadas, como por exemplo o processamento de listas.

2º) Uma implementação de uma destas linguagens numa determinada máquina nem sempre é equivalente à implementação numa outra máquina.

O primeiro problema não tem solução através da programação em Linguagens de Alto Nível. Se desejamos programar numa linguagem mais orientada para um problema, outras técnicas precisam ser utilizadas para tornar o nosso programa portátil, e que serão vistas na seção seguinte.

Uma solução para o segundo problema também é difícil. Este problema vai contra um dos principais objetivos das Linguagens de Alto Nível que é a independência de máquina. Dentre os fatores que levam duas implementações de mesma linguagem diferirem em alguns pontos podemos citar:

a) Embora as Linguagens de Alto Nível possuam definições oficiais (FORTRAN-/F66/, COBOL-/C68/, etc...), esta definição nem sempre é suficientemente completa para não permitir diferentes interpretações.

b) Nada impede que um fabricante adicione novas constru-

ções à sua implementação. Como a maioria dos usuários desconhece a definição oficial da linguagem, é sempre provável que seu programa não seja independente de máquina pois pode estar usando característ ticas particulares desta implementação.

c) As características particulares de cada máquina, como o tamanho da palavra podem ocasionar por exemplo diferenças na precisão numérica ou no número de caracteres por palavra.

Uma padronização mais efetiva, é proposta por Poole/P74/. Tal padronização não seria apenas da linguagem, mas também dos compiladores, de forma a poder testar se um dado compilador fornecido por um fabricante possui as especificações da definição da linguagem. É claro que isto não impede que sejam adicionadas novas construções às linguagens, porém tem-se a garantia que os atributos oficiais mínimos da implementação foram cumpridos.

1.1 - Três Linguagens de Alto Nível

Como exemplo, descrevemos a seguir alguns problemas com três das Linguagens de Alto Nível mais comuns.

1.1.1 - FORTRAN

Juntamente com COBOL é a mais usada das Linguagens de Alto Nível. Existe na maioria dos sistemas, e embora possua uma definição oficial /F66/, isto não é suficiente para que programas nela escritos possam ser considerados portáteis.

Vamos citar alguns destes problemas:

1º) A variável de controle de um comando *DO*, de acôrdo com /F66/ torna-se indefinida quando termina a execução deste comando, embora permaneça definida se a saída do comando *DO* se processar por meio de um desvio *(GO TO)*. Devido à dificuldade de implementar-se um valor indefinido, em geral esta variável sai do comando com o último valor assumido. Em /PD78/ são dados exemplos de

compiladores nos quais este valor é "limite superior+incremento", e outros nos quais é apenas "limite superior".

2º) A mistura de modos em expressões aritméticas é limitada pelo FORTRAN para *REAL* com *DOUBLE PRECISION* ou *REAL* com *COMPLEX* através dos operadores +,-,* e /. Porém, a maioria dos compiladores permitem a mistura de *REAL* com *INTEGER*, nas quais são feitas as conversões convenientes. O problema aqui, é que não existe um padrão pelo qual as conversões são feitas. Suponhamos a expressão $I/J+A$ com $I=1, J=2$ e $A=2.3$. Existem 2 formas para calcular o seu valor

a) calculando I/J no modo inteiro

$1/2+2.3=0+2.3=2.3$

como é feito no FORTRAN do Burroughs B6700 /F72/.

b) calculando I/J no modo real

$1/2+2.3=0.5+2.3=2.8$

como é feito por um dos compiladores FORTRAN do CDC-6400 /F69/.

3º) Quando, no comando *DO*, o valor inicial da variável de controle é maior que o limite superior, não está definido em /F66/ o que deve ser feito, embora na maioria das implementações a repetição se processa pelo menos uma vez. Em /F76/ é estabelecido que a repetição não se efetuará neste caso.

Em /PD78/ é feito um estudo comparativo de alguns compiladores onde são apresentados graves resultados quanto a portabilidade do FORTRAN. Em /S76a/ é sugerida uma programação disciplinada em FORTRAN com vistas à portabilidade.

Têm sido feitas tentativas para tornar portáteis "pacotes" de programas de aplicação ("packages") escritos em FORTRAN com o uso de pré-processadores especiais. Estes pré-processadores tem como objetivo fazer correções nos programas fonte, quando se deseja implementá-los num outro computador. Isto pode ser uma ten-

tativa válida, uma vez que se o programa é realmente portátil, apenas algumas alterações sintáticas são necessárias, como por exemplo nos comandos de entrada e saída (código do periférico, formatos, etc...). Em /A76/ e /K76/ esta tentativa é analisada com detalhes, e dois destes pré-processadores são descritos.

1.1.2 - ALGOL 60

A definição desta linguagem, é dada em /N63/. O transporte entre computadores diferentes de programas nesta linguagem, apresenta certamente mais problemas que em FORTRAN. Isto ocorre devido ao fato de certas construções necessárias nos programas, não estarem definidas nesta linguagem.

Vamos citar alguns destes problemas

1º) A entrada e saída nesta linguagem não está definida oficialmente. Embora tenham sido feitas tentativas neste sentido por Knuth/K64/ e Wirth-Hoare/WH66/ várias implementações apresentam diferenças consideráveis. Assim, a parte de entrada e saída de um programa nesta linguagem e quase sempre dependente de máquina.

2º) A aritmética em precisão dupla, embora necessária em muitos algoritmos numéricos, também não está definida.

Wichmann /W77/ descreve alguns problemas com portabilidade nessa linguagem. Hemker /H76/ apresenta algumas condições necessárias para que um transporte de programas nesta linguagem seja bem sucedido.

1.1.3 - COBOL

A versão oficial mais comum desta linguagem é apresentada em /C68/.

Certas declarações em COBOL são claramente dependentesde máquina. Por exemplo, campos podem ser declarados com atributo *SYNCHRONIZED* alinhados à esquerda ou à direita. Este campo irá

ocupar um número inteiro de palavras,e caracteres notacionais são adicionados na parte não ocupada. Assim, um registro que possui um campo com este atributo terá comprimento diferente em computadores com tamanhos diferentes de palavras. Portanto, se um arquivo com registros desse tipo é usado por um programa num computador é possível que não possa ser usado por este mesmo programa num outro computador, a menos que sua declaração seja modificada.

Situação análoga ocorre com campos de *USAGE COMPUTATIONAL* que além de dependerem do comprimento da palavra, dependem também da representação interna de cada computador.

Inglis-King/IK77/ analisam a portabilidade de dados com o uso de COBOL.

2 - PORTABILIDADE USANDO LINGUAGENS INTERMEDIÁRIAS

Uma maneira de construir-se um programa portátil é codificá-lo numa Linguagem Intermediária cuja implementação seja relativamente fácil em grande parte das máquinas reais. Estas linguagens tem a denominação de "Intermediárias" devido ao fato de estarem num nível intermediário entre Linguagens de Máquina e Linguagens de Alto Nível.

Assim, parte-se do princípio de que é possível projetar-se uma Linguagem Intermediária que contenha informações suficientes para permitir a implementação de um programa escrito nesta linguagem com relativa facilidade e com aceitável eficiência de execução /W76/.

Uma Linguagem Intermediária pode ser projetada para um particular tipo de aplicação, isto é, de modo que os algoritmos deste tipo de aplicação possam ser facilmente expressos em termos desta linguagem. Isto pode ser feito colocando-se na linguagem instruções que realizam as operações básicas daqueles algoritmos. Quando se projeta uma Linguagem Intermediária com estas características,o que está sendo feito na verdade é o projeto de uma "máquina" orientada para um determinado tipo de aplicação, isto é, uma " Máquina Abstrata" especializada /NPW72/.

As expressões "programa para uma Máquina Abstrata"e "programa em uma Linguagem Intermediária" são usados com o mesmo sentido na literatura e também o serão neste trabalho.

Nesta seção vamos discutir os princípios que devem orientar o projeto de Máquinas Abstratas, a sua possível implementação numa máquina real e finalmente o transporte de programas usando este método, entre máquinas reais.

2.1 - Projeto de uma Máquina Abstrata

Embora uma Máquina Abstrata possa ser projetada para uma grande variedade de aplicações /CPW74/, este projeto é feito em geral para um particular tipo de aplicação que pode ser algo do tipo:

um processador de listas, um editor de textos, um conjunto de rotinas para cálculos estatísticos, etc...

De acordo com isto, dado um particular tipo de aplicação, um ou mais programas devem ser escritos para implementar os algoritmos deste tipo de aplicação. Um "computador ideal" para executar estes programas deve ser então projetado. O significado de "computador ideal" não pode ser bem definido, mas pode-se exigir certas características desejáveis como por exemplo o fato desses programas poderem ser escritos com facilidade em termos das instruções deste computador ideal.

A premissa da qual se parte quando se projeta uma Máquina Abstrata, é que deve ser possível identificar as componentes básicas dos algoritmos do particular tipo de aplicação desejado: as operações e os dados /W70a/.

Embora uma Máquina Abstrata esteja definida, digamos assim, pelo seu conjunto de instruções, não possuindo portanto uma estrutura física, o objetivo final é a sua "implementação" em máquinas reais. Dizemos que houve uma implementação quando programas escritos para a Máquina Abstrata, podem ser executados de alguma forma em uma máquina real.

Existem vários compromissos a serem satisfeitos no projeto de uma Máquina Abstrata. Newey-Poole-Waite /NPW72/ indicam três princípios gerais que devem ser seguidos por um projeto deste tipo:

1. A facilidade da Máquina Abstrata para expressar os algoritmos para os quais foi projetada.

2. A relação entre a linguagem da Máquina Abstrata e a estrutura dos computadores existentes.

3. A limitação imposta pelos métodos usados para converter a linguagem da Máquina Abstrata para a linguagem de máquina de uma máquina real.

Devido às grandes diferenças existentes nas operações de

entrada e saída nos computadores,estas operações devem ser bem planejadas numa Máquina Abstrata de modo a permitir compatibilidade em grande parte dos computadores. Em /PW70/ e /W71a/ Poole e Waite estudam este problema e apresentam um modelo abstrato de operação de entrada e saída que procura englobar os conceitos comuns à maioria dos computadores e dispositivos. Este modelo foi usado para uma implementação real. Há ainda o problema da eficiência destas operações frente a um determinado sistema operacional. É essencial que seja fornecida documentação suficiente desta parte da Máquina Abstrata para que numa implementação sejam usados todos os recursos existentes. No caso da implementação real apresentada pelos autores acima, são fornecidas como documentação as rotinas que realizam as operações de entrada e saída escritas em USASI FORTRAN/F66/. Essas rotinas podem ser implementadas diretamente caso se tenha acesso ao FORTRAN, ou servirem como documentação para uma implementação em linguagem de montagem.

Várias Máquinas Abstratas já foram utilizadas para fins específicos e pelo menos uma para fins gerais. Vamos descrever a seguir algumas delas de forma suscinta.

2.1.1 - FLUB

FLUB ("First Language Under Bootstrapping") /NPW72/ foi projetada para a implementação do processador de macros STAGE2 , com o objetivo de tornar este programa portátil. Assim, existe uma versão simbólica deste programa na linguagem FLUB.

O programa STAGE2 opera sobre 3 tipos de dados: cadeias de caracteres, árvores e inteiros. Assim,FLUB deve ter uma estrutura tal que facilite a manipulação desses elementos.

A palavra desta Máquina Abstrata possui 3 campos cujos conteúdos são os seguintes (Fig. 1).

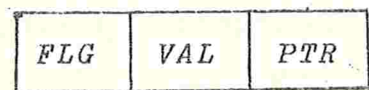


Figura 1 - Palavra da FLUB

FLG ("flag") - Indicador usado em algumas instruções. Pode ser: 0,1,2 ou 3.

VAL ("value") - Inteiro não negativo representando um caractere, ou
Inteiro não negativo representando o comprimento de uma cadeia de caracteres, ou -1 representando o final de um registro que foi lido ou a ser impresso.

PTR ("pointer")- Endereço de uma palavra da memória na FLUB, ou
Endereço de programa na FLUB, ou
Valor inteiro resultado de operações aritméticas.

Uma palavra da FLUB pode ser implementada de duas maneiras num computador real: reservando uma palavra deste computador para cada um dos campos ou compactando os 3 campos em uma ou mais palavras. A 1ª maneira facilita o adesso mas consome memória e a segunda dificulta o acesso porém economiza memória. Note-se também que a maneira escolhida para a implementação deve levar em conta os possíveis valores dos campos. Por exemplo, o comprimento do campo *PTR* determina o intervalo de valores inteiros possível bem como os possíveis valores dos endereços.

Existem 36 registradores na FLUB representados por uma letra (de A a Z) ou por um dígito (de 0 a 9). Um programa pode conter rótulos representados por 2 dígitos (de 0 a 9). As referências à memória são feitas indiretamente, isto é, o endereço referenciado deve estar sempre em um dos registradores.

Uma lista de todos os comandos da FLUB é dada na Tabela 1. Nesses comandos os parâmetros são representados por apóstrofes. Um apóstrofe representa um dos registradores e pode ser substituído por uma letra ou um dígito. Dois apóstrofes consecutivos representam um rótulo e podem ser substituídos por 2 dígitos. Os quatro

apóstrofes consecutivos no comando *MESSAGE* representam o nome de uma mensagem e devem ser substituídos por *CONV,EXPR, FULL ou IOCH.*

Como não se pode usar constantes como operandos alguns registradores são inicializados com as constantes usadas no programa STAGE2.

As operações de entrada e saída realizadas pelo programa STAGE2 são apenas leitura e impressão de uma linha de caracteres. Existem na FLUB comandos que movem um caractere da linha de entrada para o campo *VAL* de uma palavra, ou deste para a linha de saída *(VAL'= CHAR e CHAR=VAL')* e comandos que transmitem uma linha de ou para os periféricos *(READ NEXT' e WRITE NEXT')*. Até 9 periféricos são permitidos no STAGE2.

Uma crítica à FLUB (reconhecida pelos projetistas) é que devido à representação interna de cadeias de caracteres e a natureza dos comandos, a entrada e saída não é muito eficaz.

<table>
<tr><td rowspan="2">Movimento de Dados</td><td colspan="2">Registrador para Registrador</td><td>FLG'='
VAL'='
PTR'='</td></tr>
<tr><td colspan="2">Registrador para Memória</td><td>GET'='
STO'='</td></tr>
<tr><td rowspan="2">Aritméticas</td><td colspan="2">Campo VAL</td><td>VAL'='+'
VAL'='-'</td></tr>
<tr><td colspan="2">Campo PTR</td><td>PTR'='+'
PTR'='-'
PTR'='*'
PTR'='/'</td></tr>
<tr><td rowspan="8">Controle</td><td colspan="2">Incondicional</td><td>STOP
TO"
TO"BY'
RETURN BY'</td></tr>
<tr><td rowspan="3">Condicional</td><td>Campo FLG</td><td>TO" IF FLG'='
TO" IF FLG'NE'</td></tr>
<tr><td>Campo VAL</td><td>TO" IF VAL'='
TO" IF VAL'NE'</td></tr>
<tr><td>Campo PTR</td><td>TO" IF PTR' '
TO" IF PTR'NE'
TO" IF PTR'GE'</td></tr>
<tr><td rowspan="2">Entrada e Saída</td><td>Caráter</td><td>VAL'=CHAR
CHAR=VAL'</td></tr>
<tr><td>Registro</td><td>READ NEXT'
WRITE NEXT'
REWIND'
MESSAGE"" TO'</td></tr>
<tr><td rowspan="2">Pseudos</td><td>Definição de Rótulo</td><td>LOC"</td></tr>
<tr><td>Fim do Programa</td><td>END PROGRAM</td></tr>
</table>

Tabela 1 - Comandos da FLUB

2.1.2 - JANUS

JANUS /CPW74/ é uma família de Máquinas Abstratas projetada com o objetivo de produzir "software" portátil, especialmente compiladores. O seu código simbólico pode ser traduzido para a linguagem de montagem de um computador real usando um processador de macros (STAGE2).

O sentido de "família de Máquinas Abstratas" é que JANUS é projetada levando em conta uma grande variedade de máquinas reais. Assim o código simbólico possui muitas informações e o seu conjunto de instruções é bastante extenso. Assim, para uma particular situação não é necessário que todas as instruções sejam usadas, isto é, podemos limitar uma determinada implementação de JANUS. Além disso, existe a possibilidade de adicionar novas construções ao código simbólico para uma determinada aplicação, isto é, JANUS pode ser extendida.

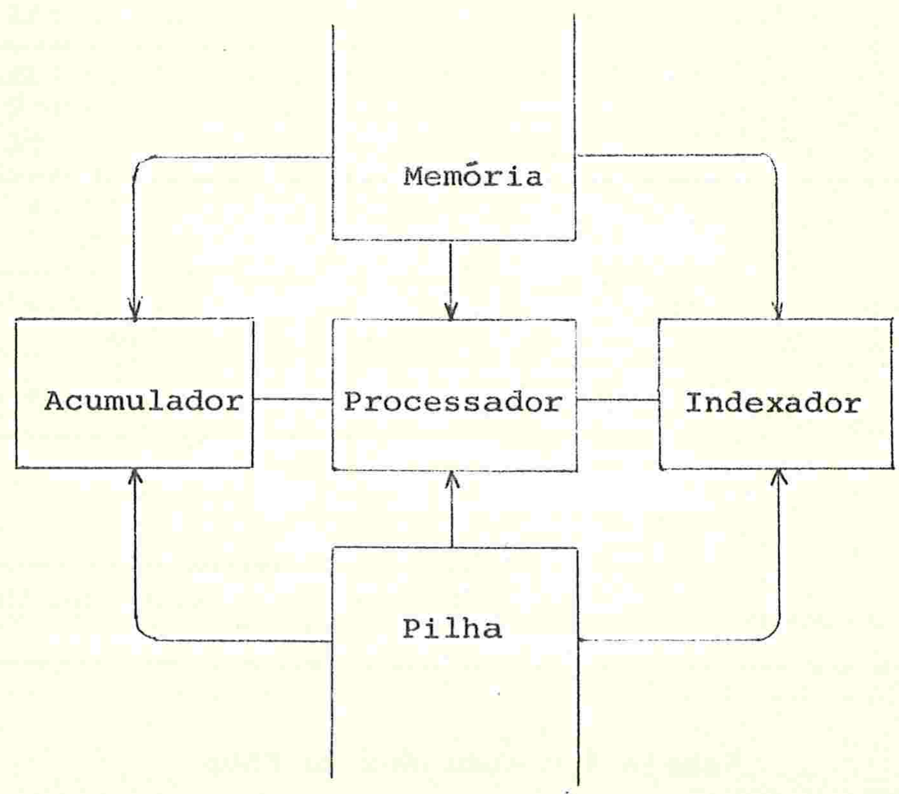


Figura 2 - Arquitetura da Família de Máquinas Abstratas JANUS

Como ilustração vamos dar o correspondente em JANUS de alguns comandos e declarações na linguagem LEAL (ver IV.1):

1º) *inteiro* X; *real* Y; *inteiro* Z[0..10]

```
SPACE   INT    GLOBAL   G1( ).
SPACE   REAL   GLOBAL   G37( ).
SPACE   INT    GLOBAL   G2(11).
```

2º) X:=27

```
LOAD  INT  CONST  X31( ) A 27.
STORE INT  GLOBAL G1( ).
```

3º) *se* X>0

então X:=inteiro (Y)
senão Y:=real (X)

```
LOAD    INT    GLOBAL   G1( ).
CMPN    INT    CONST    X31( ) A 0.
JLE     INSTR  CODE     G78 ( ).
LOAD    REAL   GLOBAL   G37( ).
CONV    REAL   INT.
STORE   INT    GLOBAL   G1( ).
JMP     INSTR  CODE     G79( ).
LOC     VOID   VOID     G78( ).
LOAD    INT    GLOBAL   G1( ).
CONV    INT    REAL.
STORE   REAL   GLOBAL   G37( ).
LOC     VOID   VOID     G79( ).
```

4º) Z[4]:=X+1;

```
LOAD    INT    GLOBAL   G1( ).
ADD     INT    CONST    G45( ) A 1.
STORE   INT    GLOBAL   G2(3*INT)
```

2.1.2.5 - Observações Gerais

Das Linguagens Intermediárias conhecidas, JANUS parece ser a que mais se aproxima de uma Linguagem Universal proposta pelos primeiros idealizadores da portabilidade /S58/. Sua característica principal é colocar em cada instrução o máximo de informações para facilitar a tradução para uma grande variedade de máquinas existentes.

2.1.3 - SIL

A implementação original da linguagem SNOBOL4 foi feita de modo a permitir portabilidade. O sistema consistindo de um compilador-interpretador foi escrito numa linguagem especialmente projetada para este fim (SIL-Snobol Implementation Language) que possue facilidades específicas para esta tarefa. A unidade básica de dados é um descritor que funciona como palavra da Máquina Abstrata. Não possui registradores, todas as instruções são de memória para memória. O número de instruções é 130 e a tradução destas para uma linguagem de montagem pode ser feita através de um macro-montador como o do IBM/360. Em /G72/ Griswold descreve a implementação desse sistema.

2.1.4 - O sistema SLANG

O objetivo inicial deste sistema era construir compiladores automaticamente recebendo como entrada:

1º) Um conjunto A de comandos descrevendo a compilação de uma linguagem.

2º) Um conjunto B de comandos descrevendo o computador para o qual a compilação seria feita.

A saída seria:

1º) Um programa em linguagem de máquina deste computador equivalente a A.

2º) Documentação apropriada

Na primeira fase seria gerado código para uma Máquina Abstrata (Máquina E). As instruções desta máquina são bastante gerais com o objetivo de poderem ser traduzidas para uma variedade de computadores, por exemplo:

MOVE A, B (A←B), *ADD A, B, C* (A←B+C). Em /S61a/Sibley descreve um conjunto de 55 instruções. Esse conjunto não é fixo, o sistema prevê a inclusão ou exclusão de instruções, o que poderia

ser útil numa determinada aplicação.

Na segunda fase as instruções da máquina E seriam traduzidas para a máquina real, usando-se as descrições dadas em B.

Segundo Coleman-Poole-Waite /CPW74/, esta foi uma tentativa mal sucedida devido à dificuldade de construir-se processadores tão poderosos.

2.1.5 - Máquinas Abstratas para Geração de Código em Compiladores

Com o objetivo de tornar o código gerado portátil muitos compiladores tem sido construídos gerando código para uma máquina Abstrata. Uma vez que a portabilidade de compiladores é um dos objetivos deste trabalho,vamos tratar este assunto com mais detalhes no capítulo seguinte, onde várias destas máquinas serão descritas.

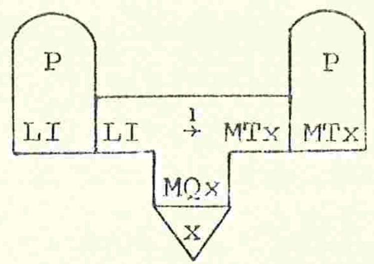
2.2 - Implementação da Máquina Abstrata em Máquinas Reais

Existem algumas maneiras para implementar-se uma Máquina Abstrata numa máquina real. A melhor maneira seria aquela que permitisse a maior eficiência possível nesta máquina real.No entanto há outras considerações a serem feitas, como o tempo que será gasto para produzir-se tal implementação. Vamos examinar os vários métodos existentes.

2.2.1 - Uso de Processadores de Macros

Esta técnica consiste em ter-se um programa que seja capaz de traduzir o código simbólico da Máquina Abstrata para uma forma executável em um computador real. Lembramos que um dos fatores a serem analisados quando se projeta uma Máquina Abstrata é a técnica que será usada na sua implementação. Assim o código pode ser projetado tendo em vista que será usado um tradutor deste tipo.

Sendo P um programa, LI a Linguagem Intermediária e x a máquina de implementação, a execução abaixo mostra este processo, supondo a tradução para a linguagem de montagem de x.



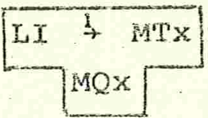
Com o objetivo de construir "software" portátil, foram projetados Processadores de Macros de uso geral, bastante flexíveis e poderosos cujo objetivo é traduzir uma variedade de Linguagens Intermediárias para uma variedade de máquinas. (STAGE2/W70b/, ML/I/B67/, GPM/S65/). Linguagens Intermediárias podem ser projetadas tendo em vista estes processadores, por exemplo JANUS e STAGE2. Cada comando é considerado uma macro e um programa é uma sequência de chamadas de macros.

É claro que devido a generalidade destes processadores para se fazer uma tradução de uma determinada Linguagem Intermediária para uma particular linguagem de montagem, é necessário fornecer ao processador a definição das macros a serem traduzidas. Assim é necessário que estas macros sejam escritas manualmente antes do processo de tradução.

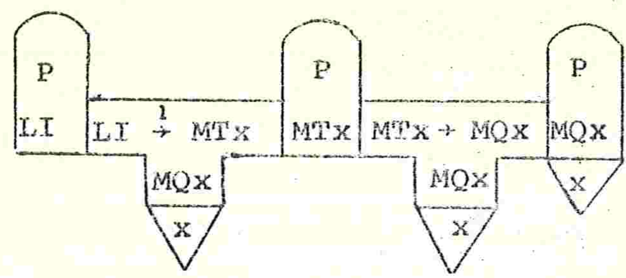
Em linhas gerais o procedimento para uso de um destes processadores é:

1) Escrever as definições de cada macro desta Linguagem Intermediária em termos da linguagem de montagem da máquina real.

2) Fornecer estas definições ao Processador de Macros obtendo então o programa:



3) Um programa P em LI seria processado por:



P
MTx

Algumas modificações manuais podem ser necessárias em visando principalmente um melhor desempenho de P no sistema operacional de x. Estas modificações podem ser: inclusão de rotinas de entrada e saída, segmentação etc....

Agora, para um programa P' em LI, apenas o item 3 deve ser repetido.

Embora cada um dos passos pareça simples, segundo os usuários desta técnica problemas podem surgir, principalmente na interface do programa com o sistema operacional.

Dependendo do tamanho e complexidade de P o tempo gasto para realizar o item 1 pode ser bem menor que para realizar o item 3.

Daremos abaixo algumas características de 2 destes processadores de macros.

2.2.1.1 - STAGE2

STAGE2 /W70b/ é um processador que recebe como entrada cadeias de caracteres transformando-as de acordo com definições fornecidas pelo usuário. Usa algoritmos de reconhecimento de padrões

("pattern matching") para isolar de uma cadeia de entrada os vários parâmetros. Parâmetros são cadeias de caracteres de qualquer comprimento.

Cada definição de macro é dada por um padrão ("template") e uma sequência de linhas que especifica a tradução desta macro. Na Figura 3 são dados exemplos de padrões, onde cada parâmetro é especificado por um apóstrofe.

*ALPHA =(')*DELTA*

ALPHA =''*

ALPHA ='

'='

'=(')'*

Figura 3 - Exemplos de Padrões para o STAGE2

A seguinte linha de entrada:

*ALPHA=(BETA+GAMA)*DELTA*

é reconhecida como pertencendo a qualquer um dos padrões da Figura 3. Quando há ambiguidade, isto é, uma linha de entrada pertence a dois ou mais padrões. Um conjunto de regras é usado para decidir qual será o padrão considerado.

As macros neste sistema são bastante versáteis. Existem vários tipos de parâmetros que especificam o tipo de substituição a ser feita e determinam instruções ao processador que possui várias facilidades durante o processamento, como operações com variáveis internas e controle de repetições. Devido à quantidade de especificações, daremos apenas alguns exemplos de definições e usos destas macros.

1º) Substituição simples.

A especificação **i0* é substituida pelo parâmetro i.

Macro: '='+'.

```
LDA   * 20$
ADD   * 30$
STA   * 10$
$
```

Chamada: *A=B+C*

Saída:

```
LDA  B
ADD  C
STA  A
```

2º) Sem substituição

A especificação **F1m* faz com que uma cópia exata da linha que constitui o corpo da macro seja impressa no canal m.

Macro: *COMENTARIO.*

```
/* ISTO E' UM COMENTARIO */*F14$
$
```

Chamada: *COMENTARIO.*

Saída: */* ISTO E' UM COMENTARIO */*

(impresso no canal 4)

3º) Atribuição à variáveis internas

A especificação **F3* usa o primeiro parâmetro como uma variável interna, atribuindo a este o segundo parâmetro.

Macro: *'FICA'*

```
*F3$
$
```

Chamada: *X FICA 5.*

Saída: nenhuma

Ação: atribui à variável interna X o valor 5.

4º) Macro condicional

A especificação **F6k* faz os parâmetros 1 e 2 serem

comparados numericamente com os operadores <,=, ≠ ou > conforme k seja -, 0, 1 ou + respectivamente. Se a comparação for verdadeira o parâmetro 3 indica o número de comandos a partir deste que devem ser pulados.

Macro: *SE '<' PULE '.*
**F6-$*
$

Chamada: *SE 14<25 PULE 3.*
Saída: nenhuma
Ação: serão pulados 3 comandos

5º) Iteração

A especificação **F7* atribui a um contador interno o valor da expressão aritmética anterior a ela. A especificação **F8* decrementa o contador e repete a iteração se o contador não for zero.

```
Macro:    DEFINA' CONSTANTES COM VALOR'.
          *10*F7$
          DC   *20$
          *F8$
          $

Chamada:  DEFINA 3 CONSTANTES COM VALOR 0.
Saída:    DC  0
          DC  0
          DC  0
```

2.2.1.2 - ML/I

ML/I /B67/ é um processador com os mesmos objetivos que STAGE2, diferindo deste quanto ao processamento das macros. Ao contrário do anterior que reconhece uma chamada de macros comparando com os diversos padrões definidos, cada macro neste sistema é identificado por um nome.

Na definição de macros neste sistema é especificado um

conjunto de delimitadores para os argumentos que permitem que estes sejam reconhecidos durante uma chamada. Um delimitador pode ter um ou mais sucessores como por exemplo na macro *IF* que será usada numa das formas abaixo.

(a) *IF argumento1 =argumento2 THEN argumento3 END*

(b) *IF argumento1 =argumento2*
THEN argumento3
ELSE argumento4 END

O delimitador *THEN* possue 2 sucessores *(ELSE e END)* enquanto *ELSE* possui apenas 1 *(END)*. Esta estrutura de delimitadores pode ser extendida indefinidamente definindo-se um delimitador como sucessor de si próprio. Isto pode ser útil no caso de expressões aritméticas definindo-se por exemplo o delimitador + como sucessor de +. Assim as expressões *A+B* e *A+B+C* podem ser tratadas pela mesma macro.

Uma outra facilidade de ML/I é a possibilidade de macros chamarem outras macros. Suponhamos os macros *INT(argumento)* e *REAL (argumento)* que transformam um valor real em inteiro e inteiro em real respectivamente. (Note-se que os nomes das macros são *"INT("* e *"REAL("* respectivamente) e a macro *SET argumento1 = argumento2* que atribui a *argumento1* o valor de *argumento2* .Uma chamada permitida seria

SET I = INT(REAL(J))

Outras facilidades internas desse sistema:

(1) Variáveis Internas Globais P1, P2,... alocadas no início do processamento.

(2) Variáveis Internas Locais T1, T2,... alocadas no início de cada chamada de macros.

(3) Macros de desvio interno condicional

MCGO argumento1 UNLESS argumento2 = argumento3 ;

que efetua um desvio interno caso *argumento2* ≠ *argumento3* ,e incondicional

MCGO argumento;

A definição é feita pelo comando *MCDEF* no qual são especificados os delimitadores. Quando existem delimitadores opcionais estes são especificados por *OPT* d_1 *OR* d_2 *OR*... *OR* d_k *ALL*.

Alguns exemplos gerando linguagem de montagem do IBM/360:

1º) Substituição simples

```
definição:MCDEF  MOVA  PARA;  AS
          <L     1,:A1
          ST     1,:A2
          >;

chamada:  MOVA  X  PARA Y
resultado: L    1,X
           ST   1,Y
```

2º) Uso de macros condicionais e variáveis internas. O valor da variável interna T2 é o número de chamadas de macros até o momento.

```
definição:MCDEF IF = THEN OPT ELSE END OR END ALL; AS
          <          L        1,:A1.
                     C        1,:A2.
                    BNE       X:T2.
                    :A3.
          MCGO L1 UNLESS      :D3.=ELSE;
                     B        Y:T2.
          X:T2.     :A4.
          Y:T2. MCGO L2.
          :L1.
          X:T2.     :L2>;
chamadas: IF A=B THEN  B  EQUAL     END
          IF I=J THEN  L  1,=F'0'
                       ST 1,I
                 ELSE  L  1,=F'1'
                       ST 1,I       END
```

```
Saída:                  1,A         PRIMEIRA CHAMADA
                        1,B
                BNE     X1          SUPONDO T2=1
                B       EQUAL
                        1,I         SEGUNDA CHAMADA
                        1,J
                NE      X2
                        1,=F'0'
                ST      1,I
                B       Y2
        X?              1,=F'1'
                ST      1,I
        Y?
```

2.2.2 - Interpretação

Este método é muito simples e consiste em escrever-se um programa interpretador que simula as operações da Máquina Abstrata numa máquina real.

Em termos gerais o algoritmo de um interpretador tem a forma:

a) Obter uma instrução na memória

b) Incrementar o contador de instruções

c) Separar os operandos (em geral na memória)

d) Desviar para a rotina que simula a execução desta instrução

e) Execução dessa rotina

f) Retorno para executar a próxima instrução

Construir-se tal programa é em geral uma tarefa simples. Portanto, poderia ser esta a a melhor maneira se não fosse a inevitável ineficiência dos interpretadores. Isso ocorre principalmente porque a execução das operações acima é muito demorada quando comparada com o equivalente feito por "hardware".

Este método pode ser útil quando a eficiência não é essencial e tem sido bastante utilizado. Pasko /P73/ cita vários trabalhos nesse sentido.

2.2.3 - Micro-Interpretação

Alguns computadores são micro-programáveis, isto é, suas instruções não são executadas diretamente por "hardware" e sim interpretadas por um microprograma que por sua vez é executado por "hardware". Este microprograma pode ser em casos especiais modificado de forma a alterar o conjunto de instruções do computador.

Assim, as instruções da Máquina Abstrata podem ser microprogramadas numa máquina real deste tipo possibilitando assim uma implementação bastante eficiente.

Algumas experiências deste tipo já foram feitas. Weber /W67a/ descreve uma implementação de um subconjunto da linguagem EULER no IBM/360 modelo 30, cujo compilador gera uma linguagem simbólica. Um interpretador, escrito em micro-código e portanto executável diretamente por hardware, foi programado para interpretar esta linguagem simbólica. Hassit-Lageschulte-Lyon /HLL73/ descrevem uma implementação análoga da linguagem APL completa no computador IBM/360 modelo 25.

Alguns computadores como Burroughs B1700/B1800,dispõe de muitas facilidades para micro-programação possibilitando uma aplicação bastante ampla para esta técnica.

2.3.4 - Outras Técnicas

Uma Linguagem Intermediária pode ser traduzida por meio de um Processador de Macros para uma Linguagem de Alto Nível ao invés de Linguagem de Montagem. Brown /B72/ apresenta uma tentativa deste tipo onde o processador ML/I é traduzido para PL/I no computador IBM/360. Isto pode ser útil se o implementador não conhece a linguagem de montagem da máquina em questão.

Pode-se usar no lugar de um Processador de Macros conven

cional um programa gerador de código, especial para cada implementação, que recebe o programa em Linguagem Intermediária traduzindo-o para a linguagem de montagem. Esta técnica é sugerida por Richards /R71/ para a implementação do OCODE e por Pasko /P73/ para a implementação de uma Máquina Abstrata para o PASCAL. As vantagens de um gerador de código sobre o Macro Processador é a sua eficiência e a possibilidade de otimizações no código gerado. A desvantagem é que não sendo um programa portátil ele deverá ser reescrito para cada implementação.

Certamente existem outras técnicas de implementação de Máquinas Abstratas. Parece-nos no entanto, que as descritas neste trabalho são as mais comuns na atualidade.

2.3 - O Transporte dos Instrumentos de Portabilidade

Na seção anterior vimos que para implementar uma Máquina Abstrata numa máquina real os instrumentos necessários em termos de "software" são Interpretadores e Processadores de Macros, dependendo da técnica a ser utilizada. Portanto se a realização será feita usando um destes instrumentos é necessário que este esteja implementado na máquina real. Será que é necessário construí-los completamente para cada implementação, ou podemos usar parte do trabalho de uma implementação para auxiliar uma outra? A segunda hipótese evidentemente é a mais desejável e de fato ela ocorre.

Note-se que se optamos pela transferência completa, tanto Interpretadores quanto Processadores de Macros são programas e portanto o nosso problema se resume novamente em construir um programa portátil. Vejamos cada um dos casos.

2.3.1 - Interpretadores

Um Interpretador não é um programa muito complexo e pode em geral ser escrito numa Linguagem de Alto Nível, por exemplo FORTRAN, e a menos dos problemas descritos na seção 1, teríamos um programa portátil.

Pode ocorrer que a eficiência de um Interpretador escri

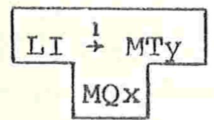
to em Linguagem de Alto Nível não seja satisfatória. Uma possibilidade então é ter-se o interpretador numa Linguagem Intermediária e traduzí-lo via um Processador de Macros para a Linguagem de Montagem do computador onde será feita a implementação. Com isto consegue-se um programa um pouco mais eficiente.

2.3.2 - Processadores de Macros

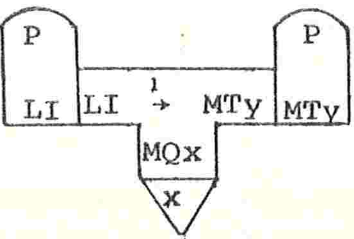
Os sistemas em geral possuem Macro-Montadores orientados para a Linguagem de Montagem deste sistema, que poderia ser usado para implementar uma Máquina Abstrata. Infelizmente aqueles possuem formatos rígidos de operação e são insuficientes para analisar textos mais gerais /S65/ como é o caso das Linguagens Intermediárias.

Portanto, é necessário um processador mais poderoso, o que significa que para implementar-se uma Máquina Abstrata é necessário implementar-se antes o Processador de Macros que em geral é um programa bastante extenso e complexo.

Essa transferência pode ser evitada se o processador já existe em uma determinada máquina, efetuando-se todo o processo de tradução nesta máquina. Mais especificamente, deseja-se mover um programa P escrito numa Linguagem Intermediária LI para uma máquina y e tem-se um Processador de Macros na máquina x. Escrevem-se então as macros que traduzem LI para MTy obtendo-se



Agora faz-se



Esta técnica é na verdade um "half-bootstrapping" daquele programa P. . Seu problema principal é que erros podem ocorrer durante a definição das macros, e deve-se então repetir novamente a geração acima na máquina x o que pode não ser muito rápido se as máquinas estão fisicamente distantes.

Para evitar este problema, vamos analisar a transferência do Processador de Macros. Se este fosse escrito numa Linguagem de Alto Nível bastante utilizada provavelmente não teríamos muitos problemas na transferência. Porém, estas Linguagens de Alto Nível mais comuns não são apropriadas para escrever algoritmos deste tipo.

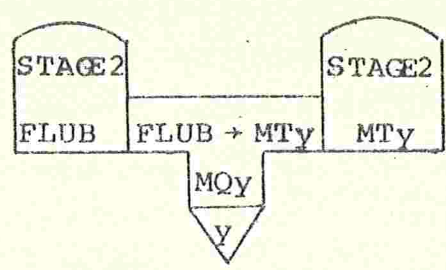
Se o processador puder ser escrito na própria linguagem que ele processa teremos uma alternativa muito interessante. Neste caso o processador em x pode transferir a si mesmo para y. É o que ocorreria se o programa P acima fosse o próprio Processador de Macros. O processador ML/I já citado, foi construido exatamente desta forma tendo sido transferido para várias máquinas /B69/.

Suponhamos agora que não temos acesso à máquina x. Neste caso todo o trabalho deve ser feito em y. Um "full-bootstrapping" do Processador de Macros é efetuado. O exemplo mais conhecido é o transporte do processador STAGE2 /PW69/. Uma versão de STAGE2 está codificada na Linguagem Intermediária FLUB que pode ser traduzida por um processador bastante simples SIMCMP /OW69/. SIMCMP é um programa em FORTRAN ASA com apenas 110 comandos, podendo ser traduzido facilmente para uma linguagem de montagem (8 homens-hora segundo Waite). O processo completo de implementação de STAGE2 é:

a) Implementar SIMCMP em FORTRAN ou Linguagem de Montagem.

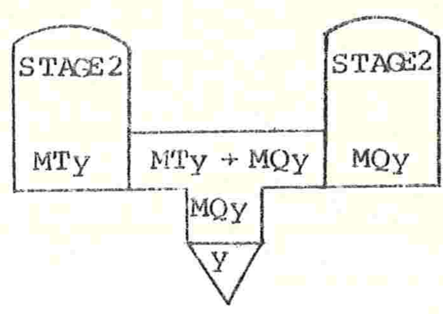
b) Escrever um conjunto de 28 macros que traduzem FLUB para a Linguagem de Montagem. Se for necessário uma implementação mais rápida já existe tradução de cada instrução de FLUB para FORTRAN.

c) Gerar STAGE2



d) Adicionar as rotinas de entrada e saída que podem ser codificadas diretamente em linguagem de montagem, visando maior eficiência e compatibilidade com o sistema operacional.

e) Gerar STAGE2 em linguagem de máquina usando o montador desta máquina.



Segundo Poole-Waite /PW69/ o esforço necessário para implementar STAGE2 numa nova máquina por este processo é de 1 homem-semana. Cremos que o maior número de experiências em portabilidade usando Linguagens Intermediárias foi sem dúvida feito com este sistema. Está implementado em vários computadores e como exemplos de processadores que foram implementados usando este sistema estão editores de texto, linguagens para processamento de listas e uma versão do SNOBOL. STAGE2 pode ser usado também para implementar outras Máquinas Abstratas como JANUS que por sua vez pode ser usado para implementar programas mais complexos como compiladores, de linguagens de Alto Nível /WH78/.

3 - CONCLUSÃO

O maior argumento dos críticos de portabilidade de programas é a ineficiência na execução de um programa portátil que ocorre em grande parte dos casos. No entanto este argumento não é irrefutável. Do ponto de vista de economia não interessa apenas a eficiência final mas também o que foi gasto para obtê-la. Assim, pode ser preferível ter-se um programa implementado em um mês, que o mesmo programa 2 ou 3 vezes mais eficiente implementado em 6 meses.

Quanto aos métodos para se obter portabilidade, parece-nos que os descritos nas seções 1 e 2 são os únicos atuais que permitem uma análise. No entanto, é difícil compará-los de uma maneira geral, mas é possível fazê-lo para um dado caso.

Finalmente, a portabilidade de programas está sendo usada e estudada em várias áreas de "software". As mais conhecidas são:

Aplicações Numéricas /C76/
Aplicações Não Numéricas /W73/
Compiladores /P74/

Cada uma das áreas possui problemas específicos quanto à portabilidade e portanto as soluções e os métodos variam. No entanto o aumento do interesse em construir-se sistemas portáteis é uma realidade.

CAPÍTULO III

PORTABILIDADE DE COMPILADORES

A transferência de um compilador de uma máquina para outra é uma tarefa um pouco mais complexa que a transferência de outros tipos de programa.

A rigor, um compilador para uma máquina M1 deve gerar código para M1. Se desejamos agora transferí-lo para uma máquina M2 além da transferência do programa compilador, será necessário modificar-se a geração de código que agora seria feita para M2. Esta modificação que pode significar uma alteração manual no programa é justamente o que desejamos evitar do ponto de vista da portabilidade.

Por outro lado, podemos considerar que um compilador para a máquina M não gere necessariamente código para M e sim um código que possa ser executado de alguma forma em M, por exemplo através de um conversor que recebe a saída do compilador e que gera código de M. Para transferir-se um compilador deste tipo, é necessário implementar-se o conversor, o que também desejamos evitar do ponto de vista de portabilidade.

Devido a estes problemas, a definição de compiladores portáteis apresenta ainda mais dificuldade que a definição de programas portáteis como foi visto no capítulo anterior. Podemos no entanto aceitar como portátil um compilador para o qual a "quantidade de esforço" na transferência de M1 para M2 seja relativamente

pequena quando comparada ao tempo necessário para construí-lo completamente em M2.

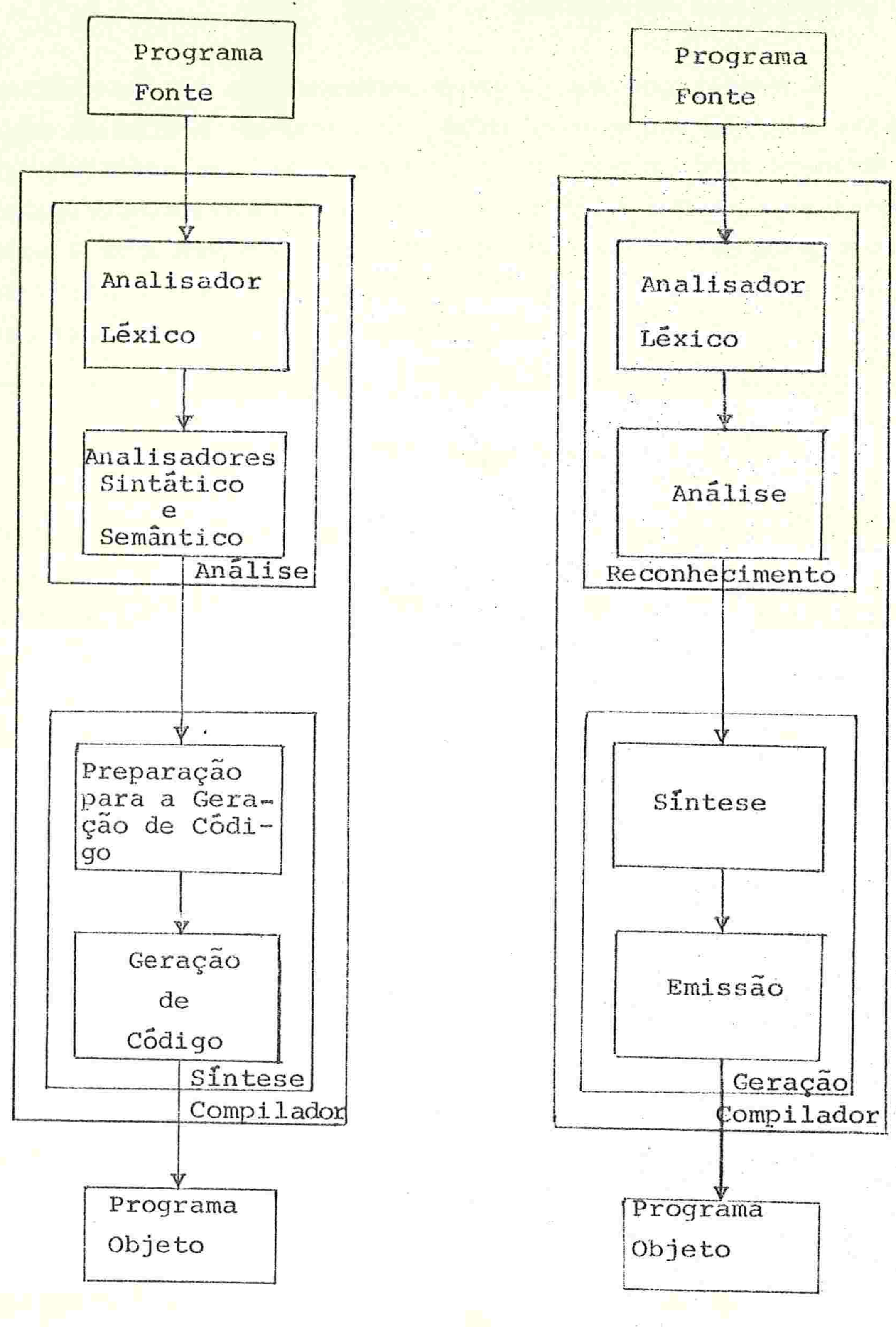
O objetivo deste capítulo é estudar as características dos compiladores portáteis. Na seção 1 descrevemos suscintamente o processo de compilação, na 2 a estrutura de um compilador portátil na 3 a transferência destes compiladores entre máquinas, na 4 alguns exemplos a na 5 algumas conclusões.

1 - O PROCESSO DE COMPILAÇÃO

A compilação de um programa consiste em sua análise e na geração de um programa equivalente, em geral em código de máquina. Embora existam diferenças significativas entre os compiladores, existe uma estrutura padrão básica que é seguida pela maioria deles. A Fig. 1 mostra esta estrutura básica como é apresentada por 2 autores. Os nomes das diferentes partes do compilador variam de autor para autor embora exista um certo consenso sobre o que cada uma das partes realiza. Existem basicamente 2 partes:

1. Análise do Programa Fonte
2. Geração de Código

Vamos descrever suscintamente estas partes com o objetivo de na seção 2 analisarmos a interferência de cada uma delas na portabilidade do compilador.



a) Gries /G71/

b) McKeeman /MHw70/

Figura 1 - O processo de Compilação

1.1 - Análise do Programa Fonte

O objetivo desta parte é determinar o significado da cadeia de caracteres que constitui o programa fonte. Dois aspectos do programa devem ser determinados: os dados e os seus atributos que são manipulados pelo programa e a sequência de operações a ser realizada sobre estes dados. A tarefa realizada nesta parte é feita em três fases: Análise Léxica, Sintática e de Contexto*

1.1.1 - Análise Léxica

O Programa Fonte é fornecido ao compilador como uma cadeia de caracteres que é varrida nesta fase. O compilador separa os vários itens léxicos deste programa como constantes, identificadores, palavras reservadas, etc... Comentários e cadeias de brancos são ignorados quando conveniente. Identificadores podem ser colocados na tabela de símbolos. Constantes podem ser codificadas na notação interna. A informação coletada nesta fase é passada para a Análise Sintática de forma codificada; assim, identificadores e palavras reservadas podem ser substituidos por índices em tabelas internas.

1.1.2 - Análise Sintática

A verificação de que o programa obedece às regras sintáticas da linguagem é feita nesta fase, que rearranja e identifica cadeias de elementos fornecidos pela Análise Léxica numa estrutura que representa a ordem das operações no programa, a qual possibilita a geração de código na ordem necessária. Se a construção sintática não está correta, erros devem ser fornecidos e possivelmente contornados para que todo o programa seja analisado.

(*) Em compilação usa-se o termo "Análise Semântica". Resolvemos utilizar "Análise de Contexto" pois "Análise Semântica" é usado hoje em dia com um significado diferente daquele empregado em compilação.

1.1.3 - Análise de Contexto

Nas Linguagens de Alto Nível, as regras sintáticas não são em geral suficientes para descrever a linguagem. Assim, torna-se necessária uma análise de contexto efetuada após a Análise Sintática. Nesta fase é completada a tabela de símbolos, pois já se tem acesso a todos os atributos de um identificador. Como na fase anterior, erros podem ser fornecidos e contornados, se for o caso. Se a construção analisada estiver correta, devem ser fornecidos à próxima parte os parâmetros necessários para que seja gerado código correspondente a esta construção.

1.2 - Geração de Código

Esta parte deve transformar a estrutura criada pela parte anterior em uma sequência de código equivalente ao programa. Pode estar dividida em várias fases, no entanto segundo Wilcox /W71b/ a maioria dos compiladores possui algumas das seguintes: Alocação de Espaço, Tradução, Otimização e Codificação.

A <u>Alocação de Espaço</u> deve associar a cada novo dado do programa um endereço que será usado pelas operações que manipulam este dado. No caso de identificadores, isto pode ser feito apenas pela inserção na tabela de símbolos dos endereços finais associados aos identificadores em questão.

Cada construção do programa fonte analisada, produzirá várias instruções no código final. A fase de <u>Tradução</u> deve ordenar os parâmetros fornecidos pela primeira parte da compilação, gerando uma "instrução intermediária" que será traduzida pela última fase.

A <u>Otimização</u> é a fase menos comum nos compiladores. Seu objetivo é detectar algumas simplificações no código gerado e efetuá-las de modo a tornar a execução do código final mais eficiente. Esta fase, quando muito elaborada é em geral efetuada sobre um código intermediário pois este apresenta maiores facilidades para o encontro de simplificações.

A <u>Codificação</u> é a tradução do código intermediário para o

código final. É necessário portanto que haja uma correspondência direta entre cada instrução do código intermediário e uma sequência de instruções no código final.

1.3 - Estrutura de Alguns Compiladores

Uma construção básica do programa, digamos um comando, deve passar por todas as fases da compilação sequencialmente. Porém, um grupo de fases pode ser aplicado a todos os comandos do programa, armazenando-o numa forma interna para em seguida ser submetido às próximas fases. Um grupo de fases com esta característica é chamado de "passo". Assim, num compilador de "um passo" todas as fases são executadas ciclicamente enquanto que num compilador de "dois passos" o primeiro pode realizar a Análise gerando um código intermediário e em seguida, o segundo passo a partir deste código intermediário gera o código final.

Em /W71b/ são dados as divisões de fases e passos de vários compiladores. Reproduzimos alguns deles na Tabela 1. Nesta tabela os passos são representados por retângulos. Se uma fase não está incluída num determinado passo o retângulo está truncado por linhas pontilhadas. Por exemplo, o compilador PL/C possui 3 passos; o primeiro é a Análise Léxica e Sintática, o segundo a Análise de Contexto e o terceiro as fases Alocação de Espaço, Tradução e Codificação. As fases Otimização e Alocação de Espaço após esta não se aplicam a este compilador. A Alocação de Espaço ocorre duas vezes na tabela, pois sua posição na sequência de fases varia nos compiladores.

FASES	PL/C	PL/I(F)	ALGOL W	BCPL	WATFOR
Análise Léxica					
Análise Sintática					
Análise de Contexto					
Alocação de Espaço					
Tradução					
Otimização					
Alocação de Espaço					
Codificação					

Tabela 1 - Fases e Passos de alguns Compiladores

2 - ESTRUTURA DE UM COMPILADOR PORTÁTIL

Existem fases da compilação que operam com as propriedades da linguagem (Analisadores Léxico, Sintático e de Contexto), fases que operam com propriedades da máquina (Alocação de Espaço e Codificação) e uma fase interna ao compilador (Tradução). Deixaremos de mencionar a fase de Otimização nas próximas descrições devido a sua pouca incidência nos compiladores.

Considerando que cada fase é dependente da anterior e que a geração de código deverá ser modificada de alguma forma a cada implementação do compilador portátil, é razoável pensar-se na separação das partes independente e dependente de máquina (abreviadas daqui por diante por PIM e PDM respectivamente). A ligação entre estas duas partes é o que deve ser projetado com grande cuidado de forma a permitir uma interação entre ambas completa e suficiente para qualquer implementação.

2.1 - A Parte Independente de Máquina

Das fases descritas anteriormente, as que com certeza compõe esta parte do compilador são: as Análises Léxica, Sintática e de Contexto e a Tradução. A Alocação de Espaço também deve ser incluída, pois ela se constitui em informações a serem passadas à PDM através da Tradução. Vejamos cada uma delas do ponto de vista de portabilidade.

2.1.1 - Análise Léxica

Um analisador léxico tradicional realiza operações que são claramente dependentes de máquinas /G71/, como por exemplo, conversão de constantes para a representação interna da máquina para a qual a compilação se realiza. No caso de um compilador portátil isto pode ser contornado por uma codificação interna do compilador.

Por outro lado, devido a simplicidade do analisador léxico, este é facilmente adaptável de uma máquina para outra. Assim, mesmo

dependente de máquina ele não seria problema num compilador portátil.

2.1.2 Análise Sintática e de Contexto

Estas duas fases são totalmente independentes de máquina . Nenhuma alteração será feita numa nova implementação.

2.1.3 Tradução

Esta é a fase que realizará efetivamente a interface entre a PIM e a PDM. Como já foi visto,esta fase deve reagrupar os parâmetros fornecidos pela Análise e passá-los para as fases que realizarão a codificação final. Existem algumas formas de passar esses parâmetros, mas o que está sendo feito na verdade é a geração de uma instrução intermediária, ou uma instrução de uma Máquina Abstrata.

Vamos a seguir fazer uma análise das características das Máquinas Abstratas para geração de código.

2.2 Máquinas Abstratas para Geração de Código

Devido a grande diferença entre as máquinas existentes, a única possibilidade para um compilador gerar um código portátil é que este código apresente facilidades de tradução para o código de várias máquinas reais. Vejamos algumas características e alguns exemplos de Máquinas Abstratas deste tipo.

2.2.1 Características Gerais

Os princípios gerais dados em /NPW72/ para o projeto de Máquinas Abstratas e apresentados no capítulo anterior (II.2.1) podem ser adaptados a este caso particular.

1. "A facilidade para expressar os algoritmos para os quais foi projetada".

Neste caso ela deve expressar as construções elementares que ocorrem na linguagem sendo compilada. É necessário identifi

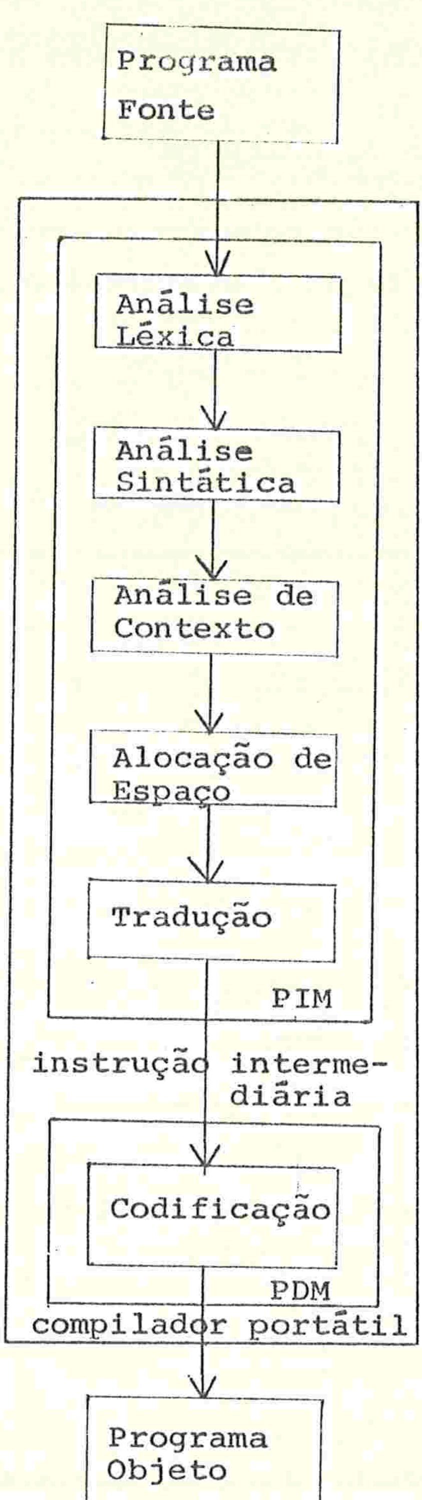


Figura 2 - Estrutura de um Compilador Portátil

car-se as operações e operandos elementares da linguagem para com e- les constituir as instruções da Máquina Abstrata. Assim, num compilador de uma linguagem de Alto Nível como FORTRAN ou ALGOL poderiam ser geradas instruções do tipo:

<u>instrução</u>		<u>comentário</u>
SOMA	p_1 p_2 p_3	$p_1 \leftarrow p_2 + p_3$
MOVA	p_1 p_2	$p_1 \leftarrow p_2$
MAIOR	p_1 p_2 p_3	desvia para p_3 se $p_1 > p_2$

2. "A relação entre a linguagem da Máquina Abstrata e a estrutura dos computadores existentes".

As instruções podem ser orientadas para determinados computadores como por exemplo, computadores com vários registradores, com um registrador ou computadores com estrutura de pilha. As Máquinas Abstratas que tem sido projetadas para compiladores de linguagens do tipo ALGOL são em geral orientadas para computadores a pilha, devido a própria estrutura da linguagem /RR64/, /P73/ e /R71/. Os exemplos anteriores numa máquina deste tipo supondo p a pilha de execução e t o seu topo ficariam:

<u>instrução</u>	<u>comentário</u>
SOMA	$p[t-1] \leftarrow p[t] + p[t-1]$; $t \leftarrow t-1$
MOVA	$p[p[t-1]] \leftarrow p[t]$; $t \leftarrow t-2$
MAIOR	desvie para $p[t]$ se $p[t-2] > [t-1]$; $t \leftarrow t-3$

3. "A limitação imposta pelos métodos usados para converter a linguagem da Máquina Abstrata para uma linguagem de máquina real".

A tradução é o objetivo final do compilador, daí a sua im

portância. Ela depende basicamente da maneira com que a Máquina Abstrata foi implementada no compilador, o que será visto em 2.3.

2.2.2 Alguns Exemplos

Como já foi dito, (II.2.1.5) vários compiladores portáteis geram código para Máquinas Abstratas especialmente definidas. Vamos descrever algumas delas.

2.2.2.1 BCPL

OCODE foi projetada por Richards /R71/ com o objetivo de tornar portátil o compilador da linguagem BCPL /R69/. Richards não usa o termo "Máquina Abstrata" e sim "Linguagem Intermediária". Existe uma versão do compilador BCPL escrita em OCODE.

Uma característica da BCPL é que esta linguagem não possui tipos de dados (inteiros, reais, etc..). Variáveis e resultados de expressões são todos armazenados em células do mesmo tamanho. Isto significa que a representação interna desses dados não é especificada e dependerá da particular implementação.

A memória para o OCODE é organizada como uma lista linear de células de mesmo tamanho e as instruções operam sobre estas células.

OCODE possui 56 comandos diferentes. Cada comando começa por uma palavra chave e é seguida por zero ou mais argumentos.

As funções e rotinas em BCPL podem ser recursivas. Assim, em OCODE os argumentos e informações de ligação são armazenados numa pilha de execução e o endereçamento é feito relativamente a um apontador P (Fig. 4) que indica o início do "registro de ativação" desta função ou rotina /M78/, que Richards chama de "stack frame". Variáveis locais também são endereçadas em relação a P.

O deslocamento S em relação a P é sempre conhecido em tempo de compilação. Quando há uma saída de bloco ou declaração de um vetor, S precisa ser alterado. Isto é feito pelo comando *STACK k* que

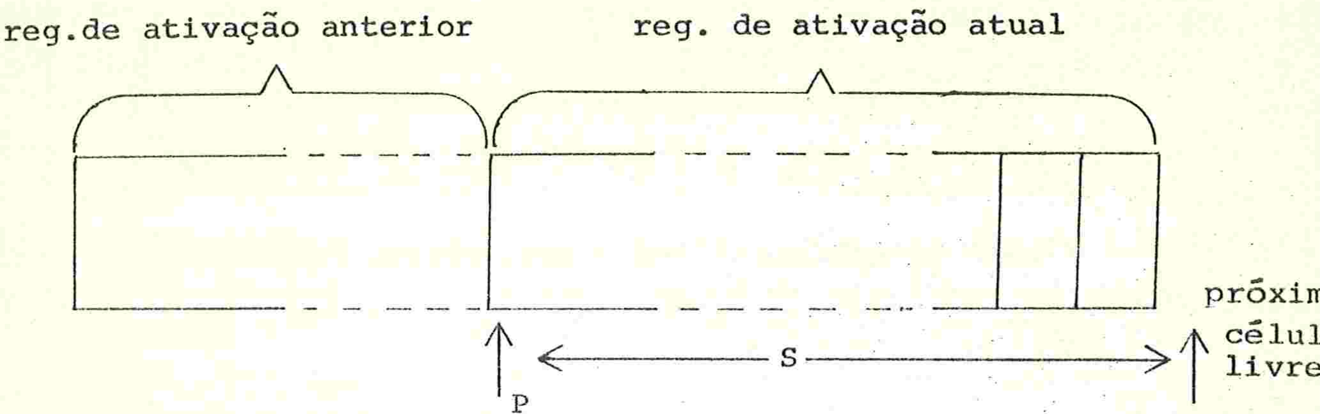


Figura 4 - A pilha de Execução

atribui a S o valor k. Note-se que S não precisa existir em tempo de execução. Assim, este comando pode ser um pseudo-comando que só tem significado durante a geração de código.

Uma chamada de função ou rotina em BCPL tem a forma *E0(E1, E2,...,En)*, onde *E0,...,En* são expressões, o valor de *E0* determina a função a ser chamada e *E1,...,En* são os parâmetros atuais. O estado da pilha de execução no momento de uma chamada deste tipo seria:

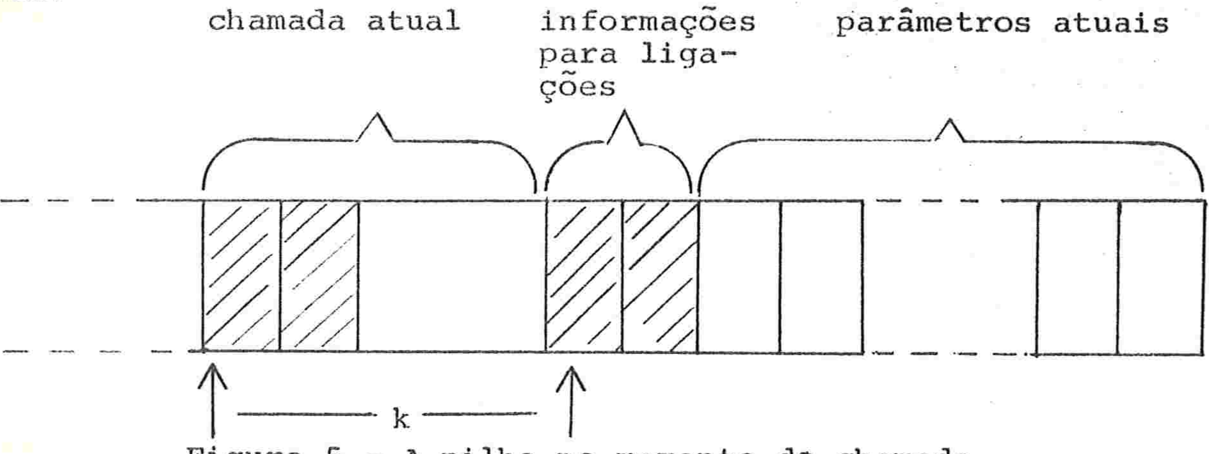


Figura 5 - A pilha no momento da chamada

O valor k, distância entre as bases da chamada atual e da chamada que está começando, é conhecido em tempo de compilação, tornando eficiente o retorno da nova chamada.

Existem 6 instruções para ligação com funções ou rotinas. Segundo Richards, a eficiência na ligação com rotinas é essencial em BCPL e portanto as instruções em OCODE correspondentes foram projetadas com muito cuidado.

2.2.2.2 ALGOL68

O ALGOL68C é uma variante do ALGOL68 desenvolvido na Universidade de Cambridge. O compilador gera código para uma Máquina Abstrata (ZCODE) orientada para as construções do ALGOL68C. É uma máquina de registradores gerais (6 ou mais), memória linear e registradores de indexação. Além de gerar ZCODE, o próprio compilador pode ser escrito neste código e existe de fato uma versão do compilador em ZCODE, com o objetivo de torná-lo portátil. Em /G77b/ é descrito o transporte deste compilador para um computador PDP-10.

2.2.2.3 ALGOL60

Existem vários projetos de Máquinas Abstratas para o ALGOL60. Uma das primeiras foi apresentada por Grau /G62/,descrita por meio de transformações simbólicas e que usa uma pilha para execução. Randel e Russel /RR64/ usam uma Máquina Abstrata para descrever a implementação do ALGOL60. Morris /M70/ descreve a máquina AOC ("ALGOL Oriented Computer") com uma estrutura bastante simples para um subconjunto do ALGOL60.

2.2.2.4 PASCAL

Uma Máquina Abstrata para a linguagem PASCAL é descrita por Pasko /P73/. Esta máquina usa várias pilhas para a execução de um programa:

(a) pilha para os marcadores do registro de ativação dos procedimentos e funções ("Run Stack").

(b) pilha para as variáveis locais ("Local Variable Stack")

(c) pilha para as variáveis alocadas pelo procedimento intrínseco *NEW* do PASCAL ("New Variable Stack").

(d) pilha para o cálculo de expressões " Expression Stack")

As pilhas a, b e d possuem também os respectivos indicadores para a

última ativação num determinado nível (pilha de bases /M78/ ou "displays"). Nesse mesmo trabalho é descrita uma implementação no IBM - 370, onde as pilhas são implementadas da seguinte forma:

(1) a e b tornam-se uma só pilha cuja pilha de bases é constituída pelos registradores gerais 12, 11,... nesta ordem.

(2) d é representada pelos registradores gerais 0,1,2,... e pelos registradores de ponto flutuante 0,1,... nesta ordem. A pilha de base de d é eliminada,pois o conteúdo dos registradores é salvo a cada entrada de procedimento ou função.

(3) c ocupa na memória a mesma área de (1), começando no final e crescendo no sentido inverso, sendo endereçada pelo registrador geral 13.

(4) os registradores gerais 14 e 15 são usados para chamada de procedimentos ou funções, já que são usados com a mesma finalidade pelas instruções do IBM-370.

2.2.2.5 Outras

Em alguns compiladores o programa fonte é traduzido para uma forma interna especial antes da codificação final. Embora não tenham objetivo de portabilidade do código gerado, estas formas internas especiais do programa fonte podem ser consideradas como uma linguagem intermediária pois apresentam facilidades para a codificação final.

Em /G71/ Gries apresenta algumas destas formas que citamos abaixo:

a) Notação Polonesa

Os operandos e operadores do programa fonte são colocados na sequência exata de execução, cada operador seguindo os respectivos operandos.

b) Quádruplas

Uma operação binária é representada por:

(operador, operando1, operando2, resultado).

Os operandos e o resultado podem ser vazios quando não e xistirem numa determinada operação.

c) Triplas

Uma operação binária é representada por:

(operador, operando1, operando2).

Como não é especificado o resultado, um dos operandos pode ser o resultado de uma tripla anterior.

Como exemplo, a Tabela 2 mostra o comando do ALGOL

if A>B *then* C:=A+B *else* C:=A-B

em cada uma destas formas (representando por *di* o desvio incondicional e por *df* o desvio se falso).

a) Notação Polonesa	b) Quádruplas	c) Triplas
(1) A B > 13 df	(1) >, A, B, T1	(1) >, A, B
(6) C A B + := 18 di	(2) df, T1, 6,	(2) df, (1),6
(13) C A B - :=	(3) +, A, B, T2	(3) + ,A, B
(18)	(4) :=, T2, , C	(4) :=, (3),C
	(5) di, 8, ,	(5) di, 8,
	(6) - , A, B, T3	(6) - , A, B
	(7) :=, T3, , C	(7) :=, (6), C
	(8)	(8)

Tabela 2

2.3 - A Parte Dependente de Máquina

A forma de receber e usar a instrução intermediária gerada

pela PIM depende da maneira com que a Máquina Abstrata para Geração de Código esta implementada no compilador. Vejamos algumas formas:

2.3.1 Uma Rotina por Instrução

A cada instrução da Máquina Abstrata corresponde uma rotina do compilador. Quando uma determinada instrução precisa ser gerada, uma chamada é feita à rotina correspondente. As rotinas constituem a PDM (Fig. 6) e são chamadas pela PIM.

Para modificar-se a geração de código para a implementação do compilador numa determinada máquina, deve-se reescrever manualmente todas estas rotinas /P74/.

Por exemplo, supondo que uma dessas rotinas é *SOMA(x,y,z)* que deve fazer $x \leftarrow y+z$. Quando o compilador precisa gerar uma soma, uma chamada do tipo *SOMA* (p_1,p_2,p_3) é efetuada . Eventualmente p_1, p_2 ou p_3 podem ser temporários e a rotina *SOMA* pode testar isto gerando código otimizado.

Se a PIM for executada num só passo, o compilador inteiro terá um só passo caso seja adotado este método, pois a cada chamada de uma rotina da PDM o código final será gerado. Isto implica que a fase de Alocação de Espaço deve ser feita concorrentemente com a codificação. Devido à independência entre as duas partes é conveniente que a própria PDM possua uma tabela de associação de endereços dos dados usados no programa, ao invés de usar a própria tabela de

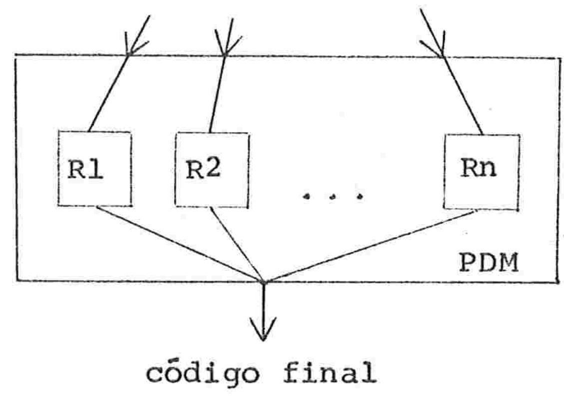


Figura 6 - A PDM de 2.3.1

símbolos da PIM. Esta tabela pode ser criada por pseudo-instruções da Máquina Abstrata, equivalentes às instruções de declaração de dados. Uma possibilidade muito usada para evitar a manipulação direta com endereços, é gerar-se Linguagem de Montagem do Computador em questão ao invés de Linguagem de Máquina, embora isto implique num segundo passo que é a montagem do programa.

2.3.2 - Gerando uma Instrução Intermediária

Com este método, uma cadeia de caracteres é gerada pela PIM contendo a instrução intermediária e seus parâmetros e passada à PDM. Esta parte identificaria qual a instrução e quais os parâmetros e selecionaria a codificação correspondente. No caso de uma soma poderia ser gerada a cadeia:

$$\text{"}SOMA \quad p_1 \quad p_2 \quad p_3\text{"}$$

ou uma outra codificada de forma mais compacta.

O compilador poderia ter 1 ou 2 passos. De 1 passo, se a cada instrução gerada fosse feita uma chamada à PDM (Fig. 7a). De 2 passos caso fosse gerado todo o programa na linguagem intermediária, armazenado e depois feita a chamada a PDM que poderia até ser um programa independente do compilador (Fig. 7b).

O 1º caso tem as mesmas características do método anterior (2.3.1). O 2º caso embora menos eficiente (2 passos) e precisando de

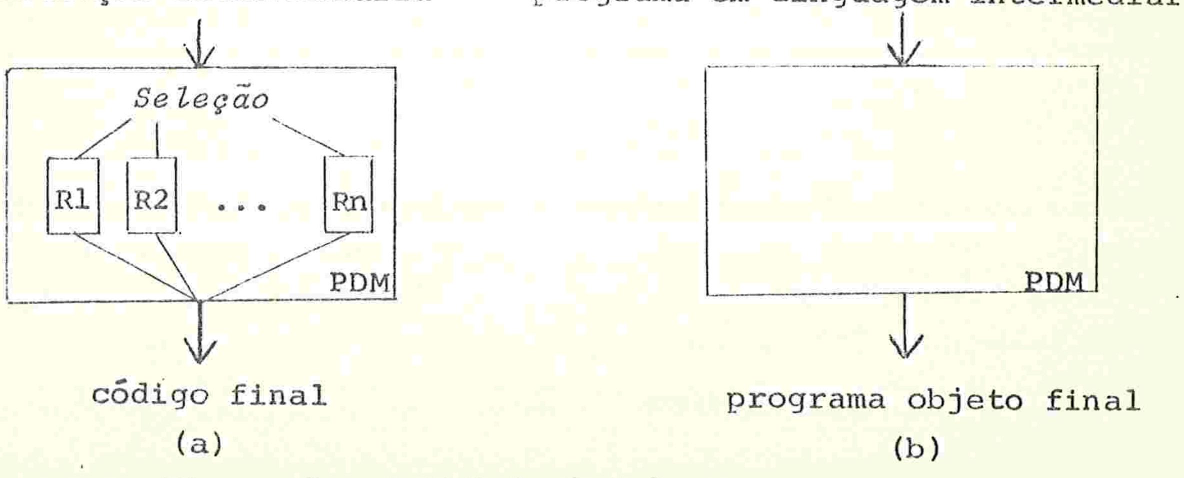


Figura 7 - A PDM de 2.3.2

área auxiliar, pode permitir otimização no programa objeto final,uma vez que a PDM tem acesso a todo o programa.

Devido a semelhança entre o método 2.3.1 e o 1º caso de 2.3.2, caberia uma análise comparativa. Vejamos alguns aspectos:

a) Eficiência de Compilação

Em 2.3.2 a instrução intermediária é passada para a PDM que deve selecionar os parâmetros e a codificação correspondente e efetuá-la. Em 2.3.1 a rotina é chamada diretamente da PIM, sendo portando um pouco mais eficiente.

b) Eficiência no Código Gerado

Em ambos os casos o código gerado pode ser o mesmo.

c) Confiabilidade

Em 2.3.1 a modificação pode ser feita manualmente no próprio compilador. Dependendo de como o compilador está escrito (Linguagem de Alto Nível, Linguagem Intermediária, etc..) isto pode ser difícil e pouco confiável. Este problema pode ser evitado se as rotinas puderem ser escritas independentemente numa linguagem conveniente, testadas e depois ligadas ao compilador.

Em 2.3.2 a rotina que constitui a PDM é independente e certamente é possível escrevê-la, testá-la e depois ligá-la ao compilador.

2.3.3 - Uso de Processadores de Macros

Linguagens Intermediárias podem ser traduzidas por processadores de macros. De fato, ZCODE (ver 2.2.2.2) pode ser traduzido pelo processador ML/I (ver II.2.2.1.2) /G77b/ e JANUS (ver II.2.1.2) que já foi usado para geração de código em compiladores /WH78/ pode ser traduzido por STAGE2 (ver II.2.2.1.1).

Como um Processador de Macros é um programa externo ao compilador, este método implica numa compilação de no mínimo 2 passos.

Neste caso portanto, a PDM é constituida pelo Processador de Macros com as definições para tradução de cada uma das instruções da Linguagem Intermediária (Fig. 8), supondo que é gerado diretamente o código final, pois se o Processador de Macros gerar Linguagem de Montagem, o programa Montador também fará parte da PDM.

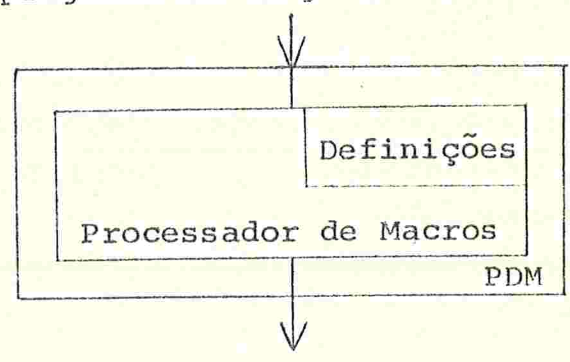


Figura 8 - A PDM de 2.3.3

2.3.4 - Processador de Macros no Compilador

A PDM pode ser um Processador de Macros interno ao compilador. Com isso, não seria necessário que o processamento das macros fosse um passo especial, pois a PDM seria executada concorrentemente com a PIM.

Embora diminuisse a eficiência em relação a 2.3.1, facilitaria a implementação, pois ao invés de escrever todas as rotinas, apenas as definições das macros seriam fornecidas. Assim, este método usa as facilidades oferecidas pelos processadores de macros descritos no capítulo anterior de modo a aumentar a portabilidade do compi-

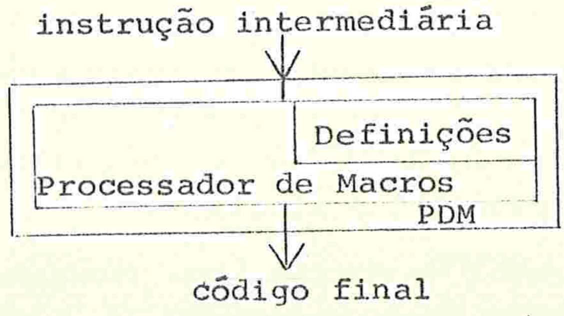


Figura 9 - A PDM sugerida em 2.3.4

lador.

Enquanto existem exemplos práticos de uso dos métodos anteriores, não constatamos este método em nenhum dos casos estudados . No entanto, embora o seu uso torne o compilador mais complexo pois um Processador de Macros é um programa de tamanho e complexidade considerável, nós o achamos perfeitamente viável.

O Processador de Macros poderia por facilidade gerar Linguagem de Montagem que seria diretamente traduzida pela própria PDM para Linguagem de Máquina.

3 - A IMPLEMENTAÇÃO DE UM COMPILADOR PORTÁTIL

Embora cada implementação possa apresentar certas particularidades inerentes ao caso, a maioria delas usa alguma forma de "half-bootstrapping" ou "full-bootstrapping" (ver I.4).

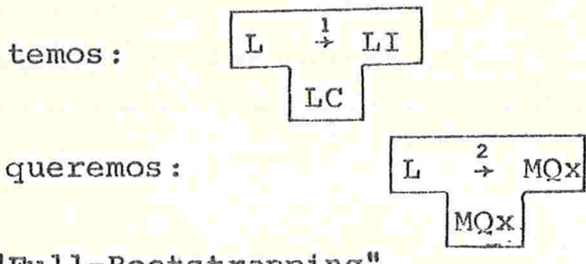
A cada nova implementação será necessário modificar-se de alguma forma a geração de código. Indicaremos esta operação por =>. Por exemplo:

LI → L2 / L => L1 → L3 / L

indica que o tradutor de L1 para L2 foi modificado para traduzir L1 para L3. Esta operação pode ser feita de várias formas (ver 2.3).

3.1 - Compiladores em Linguagem de Alto Nível

Consideremos um compilador portátil da linguagem L, gerando a linguagem intermediária LI e escrito na linguagem de alto nível LC. Deseja-se implementá-lo na máquina x.



3.1.1 - "Full-Bootstrapping"

Neste caso existe compilador para LC em x, LC 3 → MQx / MQx

a1) L 1 → LI / LC => L 4 → MQx / LC

a2) L 4 → MQx / LC ; LC 3 → MQx / MQx ; L 2 → MQx / MQx ; x

3.1.2 - "Half-Bootstrapping"

Não existe compilador para LC em x. Neste caso deve haver um compilador de LC para uma máquina y que possa ser modificado de alguma forma para gerar código para x obtendo-se um "cross-compiler" /H74/, [LC $\xrightarrow{3}$ MQx | MQy]

b1) [L $\xrightarrow{1}$ LI | LC] => [L $\xrightarrow{4}$ MQx | LC]

b2) [L $\xrightarrow{4}$ MQx | LC] [LC $\xrightarrow{3}$ MQx | MQy] [L $\xrightarrow{2}$ MQx | MQx]

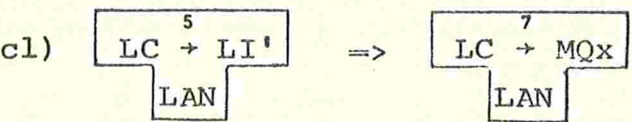
MQy

y

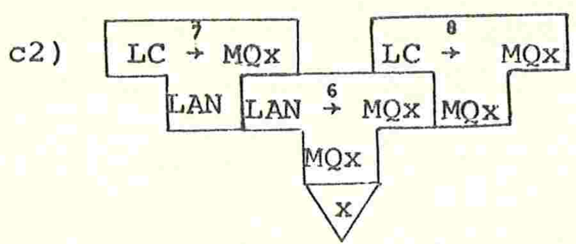
Este caso sugere que se tenha um "compilador de trabalho" para LC, [LC → LI' | MQy] cuja geração de código seja facilmente modificável (LI' indica a linguagem intermediária deste compilador com facilidades de modificação).

Uma outra possibilidade seria ter-se o "compilador de trabalho" escrito numa linguagem de alto nível LAN bastante comum (FORTRAN, COBOL, etc...), [LC $\xrightarrow{5}$ LI' | LAN] . Não importa sua eficiência,pois será usado uma só vez. Assim, se existir compilador para LAN em x, isto é [LAN $\xrightarrow{6}$ MQx | MQx] , o seguinte "full-bootstrapping" pode ser usado.

c1)



c2)

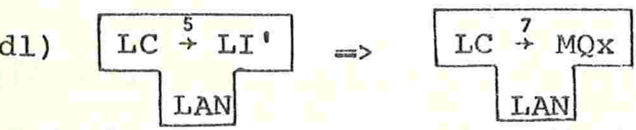


c3) idem a a1

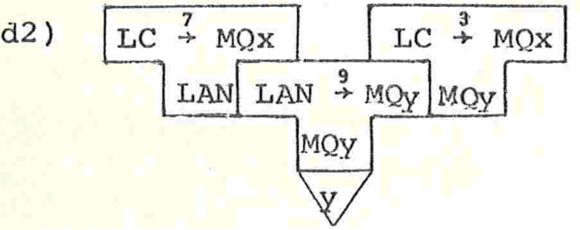
c4) idem a a2 (usando 8 ao invés de 3)

Caso não haja compilador para LAN em x, um "half-bootstrapping" usando y onde há o compilador [9: LAN → MQy / MQy] pode ser feito.

d1)



d2)



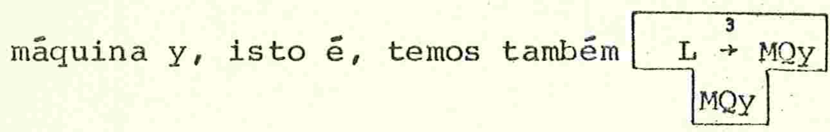
d3) idem a b1

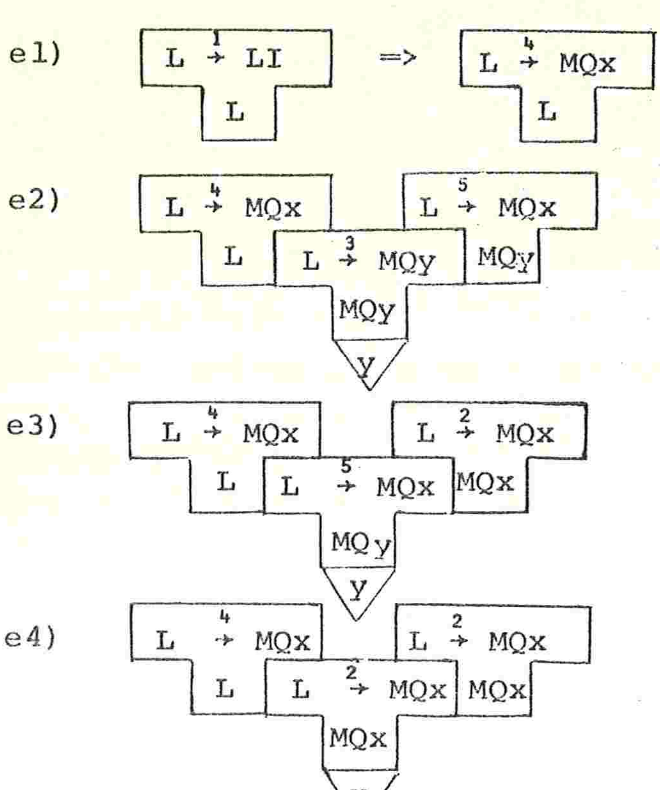
d4) idem a b2

3.1.3 - Compiladores Autocompiláveis

Quando um compilador pode ser escrito na própria linguagem que compila, isto é LC=L, uma variação pode ser feita no "half bootstrapping".

Suponhamos primeiramente que L já está implementada numa máquina y, isto é, temos também



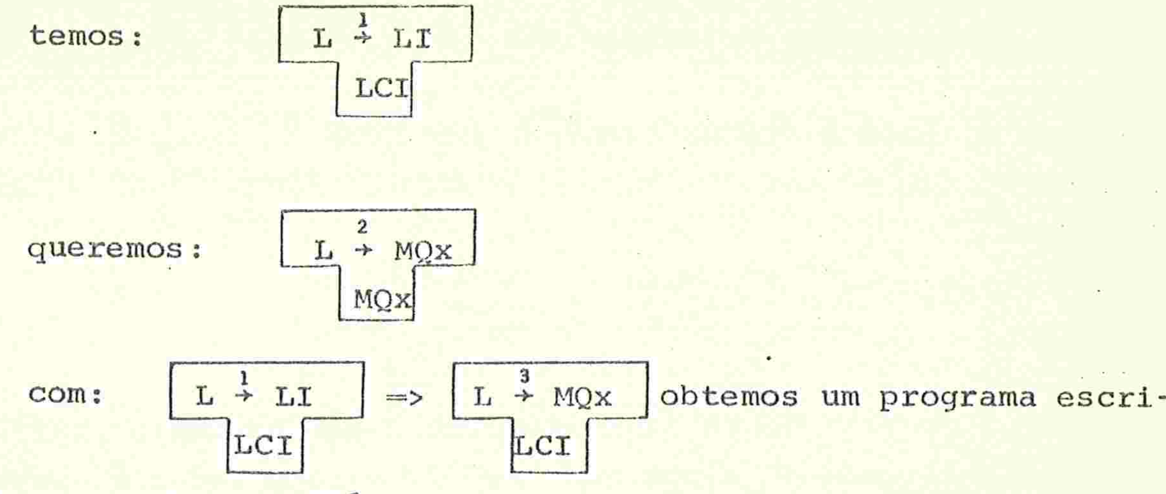


O passo e4 embora desnecessário, mostra que o compilador portátil é agora autocompilável em x.

A necessidade de uma implementação inicial faz com que a primeira versão do compilador deva ser escrita para uma máquina y, numa linguagem de alto nível existente em y ou mesmo na linguagem de montagem de y. Esta primeira versão não será portátil e sua eficiência não vem ao caso pois ela pode ser usada apenas para implementar a primeira versão portátil em y.

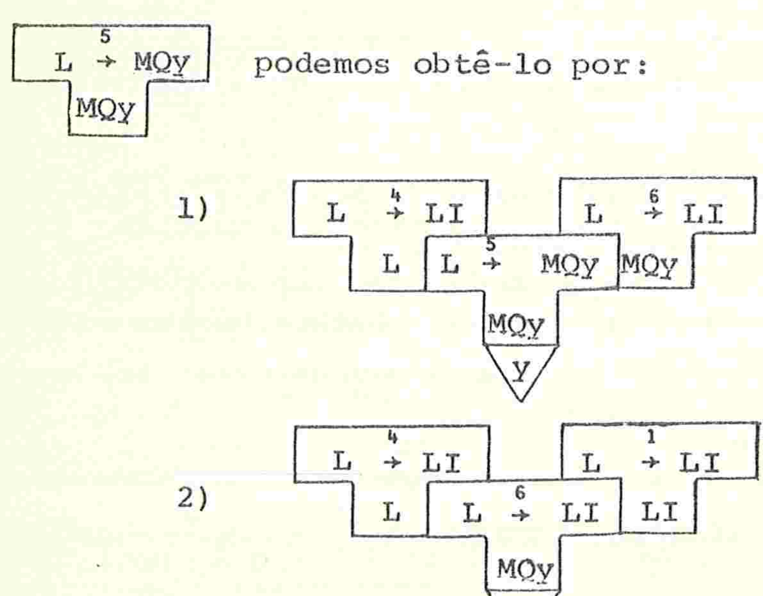
3.2 - Compiladores em Linguagem Intermediária

Consideremos agora um compilador portátil da linguagem L, gerando a linguagem intermediária LI e escrito na linguagem intermediária LCI. Deseja-se implementá-lo na máquina x.



obtemos um programa escrito na linguagem intermediária LCI e o problema resume-se em implementar este programa em x. No capítulo anterior (II.2) estudamos a portabilidade de programas em linguagens intermediárias e os métodos lá estudados aplicam-se completamente aqui.

O compilador [L →¹ LI / LCI] pode ser obtido mecanicamente o que é muito útil, pois escrevê-lo manualmente pode ser um trabalho demorado e sujeito a muitos erros. No caso dos compiladores autocompiláveis LCI e LI podem ser a mesma linguagem e tendo-se [L →⁴ LI / L] e



podemos obtê-lo por:

4. ALGUNS COMPILADORES PORTÁTEIS

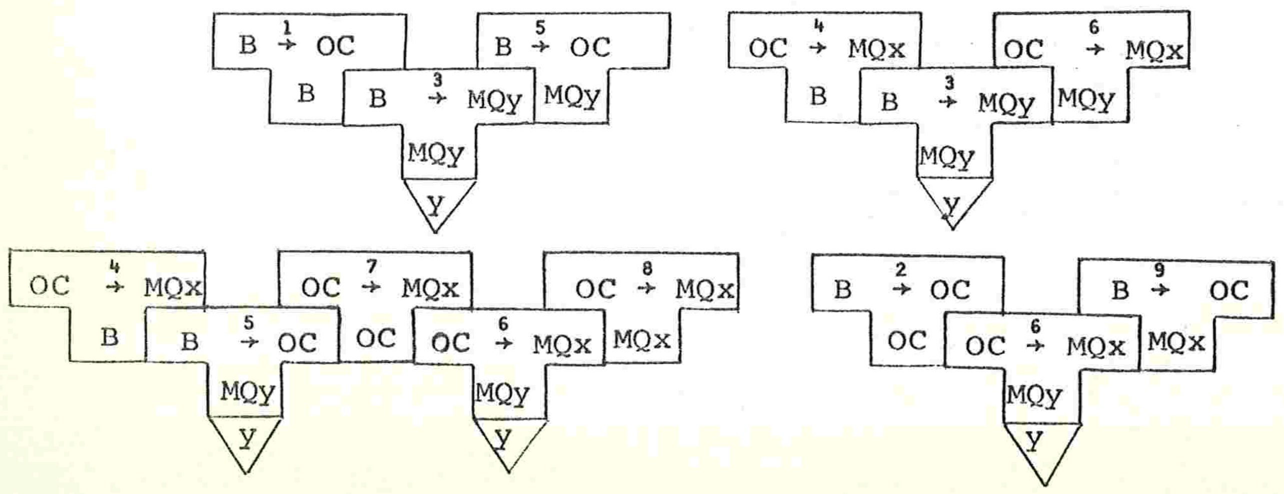
Vamos descrever a portabilidade de dois compiladores cujos métodos apresentam algum interesse. Outros exemplos são encontrados em /P74/.

4.1 - BCPL

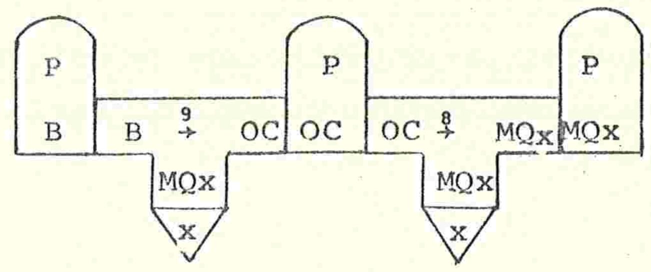
O projeto desta linguagem /R69/ foi bastante influenciado pela portabilidade do compilador /R71/. O compilador é autocompilável e gera a linguagem intermediária OCODE (ver 2.2.2.1). Existem várias implementações (360, 7094, XDS Sigma5, 6600 e outros) que usaram basicamente três métodos, descritos abaixo (abreviando BCPL por B e OCODE por OC).

1º) Usando uma implementação já existente numa máquina y ("half-bootstrapping"). São fornecidos [1: B → OC / B] e [2: B → OC / OC] e tem-se [3: B → MQy / MQy]

Deve-se escrever um gerador de código [4: OC → MQx / B], e o "bootstrapping" é realizado (Fig. 10a) tornando possível a compilação de programas em BCPL na máquina x (Fig. 10b).



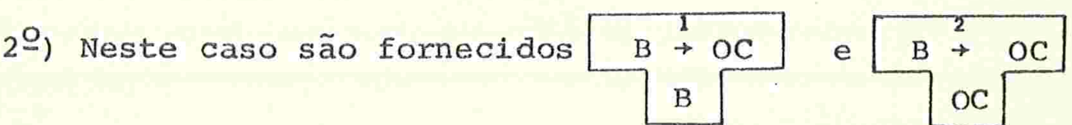
a) "Bootstrapping" do compilador



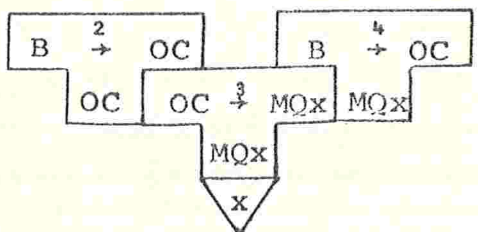
b) Compilação em x

Figura 10 - "half-bootstrapping"

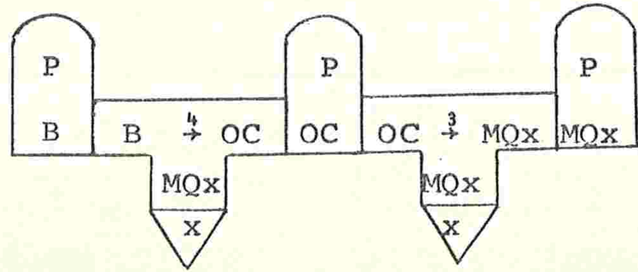
2º) Neste caso são fornecidos



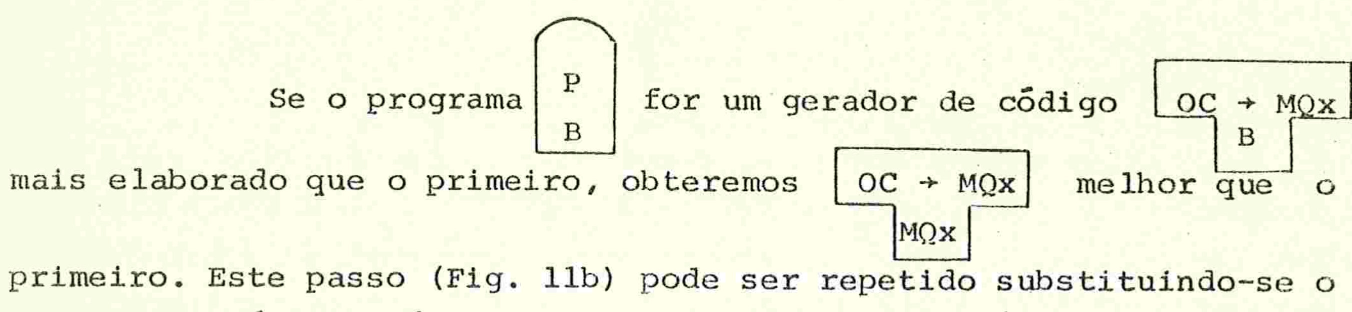
Escreve-se um gerador de código em alguma linguagem existente em x obtendo [OC 3 → MQx / MQx] que será usado apenas durante a implementação ("full-bootstrapping") não sendo relevante portanto a sua eficiência. Obtém-se então um compilador de trabalho (Fig.11a) que pode ser usado para compilar as novas rotinas de geração de código escritas em BCPL e otimizadas que serão ligadas mais tarde à versão final do compilador (Fig. 11b).



(a) Compilador de trabalho



(b) Compilador em x



Se o programa [P / B] for um gerador de código [OC → MQx / B] mais elaborado que o primeiro, obteremos [OC → MQx / MQx] melhor que o primeiro. Este passo (Fig. 11b) pode ser repetido substituindo-se o gerador de código até que se obtenha uma versão satisfatória deste tradutor.

3º) Com o objetivo de minimizar o tempo de implementação do compilador (2 homens-mes nos métodos anteriores) Richards projetou a linguagem interpretativa INTCODE de nível mais baixo que OCODE /R74/. Com o uso de INTCODE (abreviada IC) a implementação cai para 1 homem-mes segundo o autor.

São fornecidos [B →¹ OC / B] , [B →² OC / IC] e [IC ³ B]

O interpretador fornecido (3) pode ser usado como documentação para escrever-se um outro numa linguagem existente em x obtendo [IC ⁴ MQx] . O compilador fonte (1) é também fornecido só para documentação.

Como no segundo método obtém-se [OC →⁵ MQx / MQx] e um compilador de trabalho está disponível para a implementação (Fig. 12) e pode ser usado para obter um gerador de código melhor.

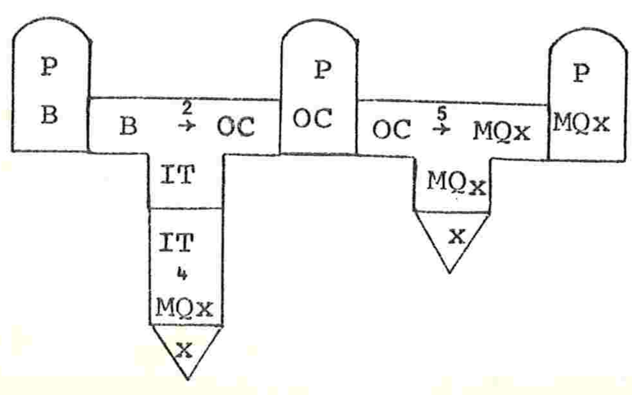


Figura 12 - Compilador de trabalho interpretado

4.2 - PASCAL

A possibilidade de um compilador autocompilável para esta linguagem foi usada na primeira implementação para a série CDC-6000 por Wirth /W71c/. Primeiramente um compilador para um subconjunto da linguagem (60%) foi escrito neste próprio subconjunto e traduzido a mão para uma linguagem existente no CDC-6000 obtendo-se um compilador (1) usado apenas para a implementação da versão final que foi escrita neste subconjunto (2) e posteriormente usando toda a linguagem (3) (Fig. 13).

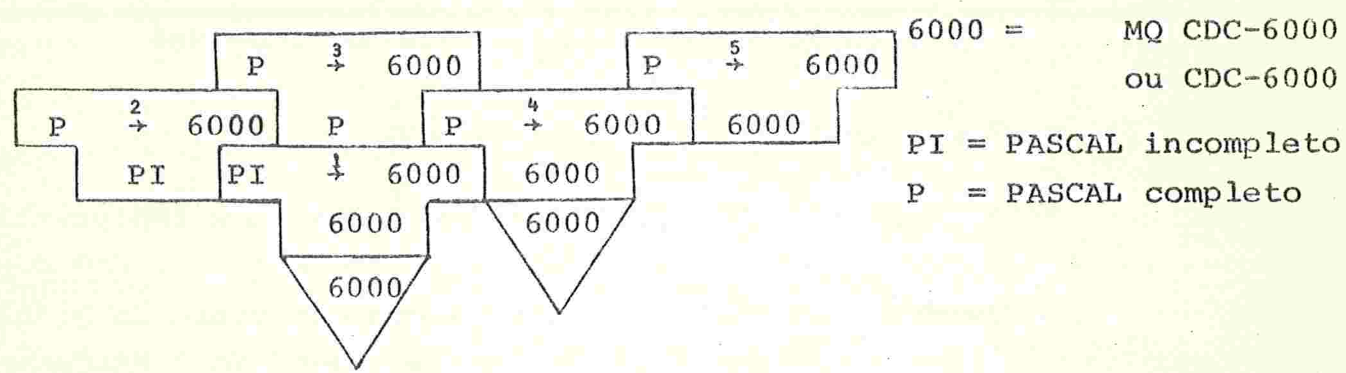


Figura 13 - Compilador PASCAL no CDC-6000

Em /LP78/ são descritas várias implementações, a maioria delas tendo como base a implementação inicial. Vários parâmetros, como tempo de implementação, eficiência do compilador, eficiência do código gerado são avaliados comparativamente. Mostraremos três delas escolhidas devido às diferentes características.

1º) Implementação no computador ICL-1900

Welsh-Quinn /QW72/ apresentam uma implementação ICL-1900. Um "half-bootstrapping" usando a versão fonte (3) e a versão objeto (5) do compilador PASCAL no CDC-6000 e sendo testado por um interpretador do ICL-1900 no CDC-6000 (7) (Fig. 14).

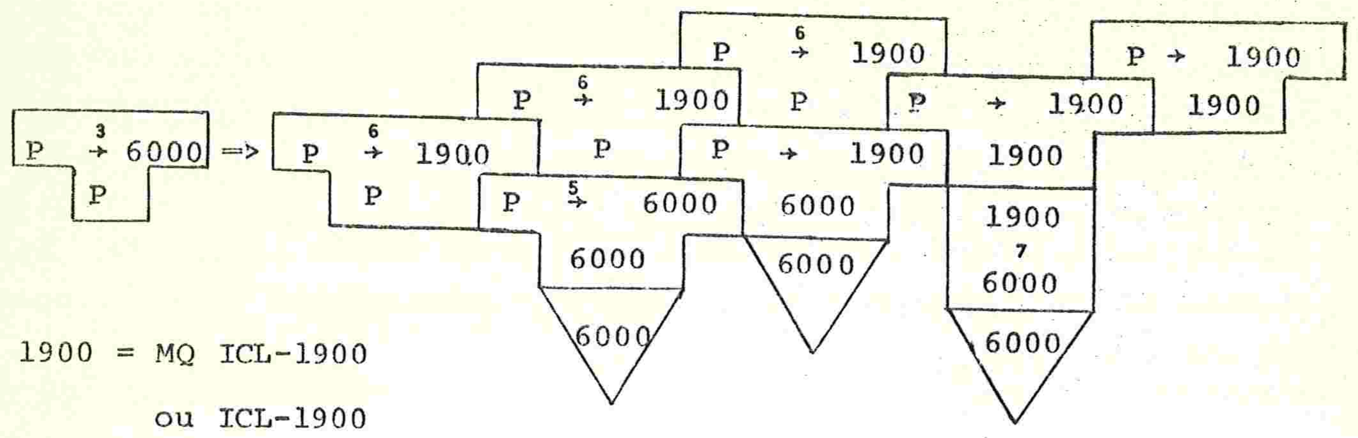


Figura 14 - Compilador PASCAL no ICL-1900

2º) Implementação no computador PDP-10

Nagel-Grosse-Lindeman /NG76/ apresentam uma implementação no PDP-10. Um "full-bootstrapping" usando um compilador PASCAL-P escrito na linguagem de uma máquina abstrata com estrutura de pilha e gerando também esta linguagem (6). Uma versão fonte do PASCAL-P (7) e um interpretador da máquina abstrata (8) também foram utilizados (Fig. 15).

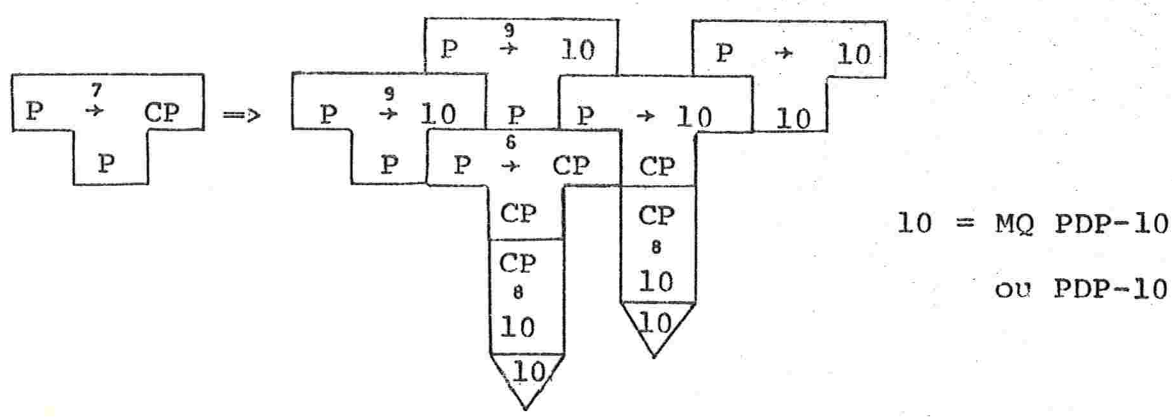
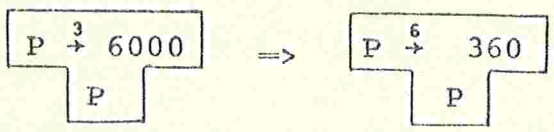


Figura 15 - Compilador PASCAL no PDP-10

3º) Implementação no computador IBM-360

Russell-Sue /RS76/ apresentam uma implementação no IBM-360 com um "full-bootstrapping". É usado um compilador de trabalho escrito em PL/I (4) e o compilador PL/I deste computador (5). Este compilador de trabalho é usado para compilar a versão fonte do compilador PASCAL no CDC-6000 (3). (Fig. 16).



360=MQ IBM-360

ou IBM-360

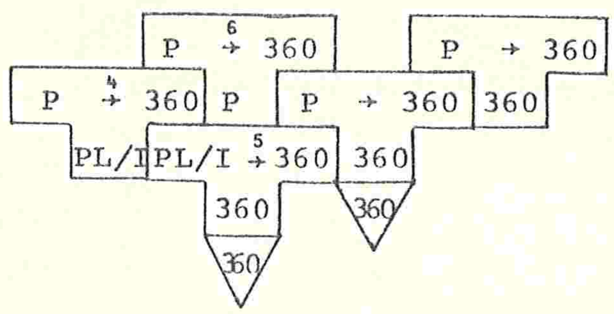


Figura 16 - Compilador PASCAL no IBM/360

5 - CONCLUSÃO

Não resta dúvida que para obter-se um compilador com bom desempenho numa particular máquina o melhor é escrevê-lo sob medida para esta máquina aproveitando todos os recursos de hardware e adaptação ao sistema operacional existentes. No entanto, como em II.3 a eficiência final quando comparada com a facilidade de implementação pode não ser o fator mais importante; neste caso um compilador portátil se apresenta como uma alternativa razoável.

Um compilador portátil encoraja implementações em novas máquinas evitando assim a proliferação de "dialetos" de uma mesma linguagem, que é uma consequência de novas implementações de um mesmo compilador (ver II.1).

Quanto ao método que deve ser usado, vimos nos casos estudados que cada um pode apresentar características particulares ao caso. Lecarme e Peyrolle-Thomas /LP78/ apontam como fatores influenciado a escolha do método: *o computador* que pelo "software" oferecido pode possibilitar um "half" ou "full-bootstrapping"; *a equipe de implementação* cuja competência e conhecimentos pode ser o fator mais importante na escolha do método e o *resultado pretendido* onde quase sempre o método mais fácil não leva a melhor implementação.

Poole /P74/ sugere na maioria dos casos um "full-bootstrapping" pois a utilização em conjunto de duas máquinas num "half-bootstrapping", pode causar demoras se as máquinas estão fisicamente distantes, no caso de se ter de repetir processos devido a erros.

Mesmo que exista uma versão em linguagem intermediária, parece uma idéia bastante aceita que se tenha também uma versão numa "boa" linguagem de alto nível, pois no mínimo esta versão servirá como documentação. Além disso, se a linguagem permite um compilador autocompilável isto é sempre feito, talvez devido ao fato de que apenas uma linguagem de alto nível estará envolvida no processo.

CAPÍTULO IV

A PORTABILIDADE DO COMPILADOR PARA A LINGUAGEM LEAL

O trabalho descrito neste capítulo teve dois objetivos:

1) Desenvolver um compilador para uma linguagem educacional que vem sendo usada no IMEUSP para o ensino básico de computação.

2) Realizar uma aplicação prática do estudo que fizemos sobre portabilidade.

O resultado foi o projeto e construção de um compilador portátil para aquela linguagem.

Assim, na seção 1 damos as características principais da linguagem, na 2 descrevemos a estrutura do compilador, na 3 a linguagem intermediária para geração de código, na 4 o transporte do compilador e na 5 nossas conclusões e justificativas.

1 - A LINGUAGEM LEAL

A Linguagem de Programação LEAL (Linguagem Educacional ALgorítmica), foi desenvolvida por V.W.Setzer /SS77/ visando principalmente o ensino básico de computação. Os objetivos principais que orientaram o projeto da linguagem foram:

o1) Conter somente os conceitos fundamentais de linguagens orientadas para a construção de algoritmos /S76b/.

o2) Facilitar a formulação de algoritmos sob forma de um programa estruturado.

o3) Permitir compiladores eficientes e compactos, visando implementações em mini-computadores.

A construção de um compilador portátil para esta linguagem motivou a inclusão de algumas construções especiais. Estas construções, não alteraram o objetivo geral da linguagem e facilitaram a tarefa do compilador ser escrito na própria linguagem LEAL. As construções incluídas foram:

i1) Inicialização de variáveis declaradas, permitindo economia de memória no compilador e simplificando a programação.

i2) Extensão do comando *mova*, para permitir uma manipulação de caracteres mais eficaz .

i3) Comparação de elementos contíguos de matrizes,permitindo comparação de cadeias de caracteres, que é uma operação muito comum no compilador.

i4) Comando de desvio incondicional, para facilitar a programação do particular analisador sintático escolhido.

As inclusões i1, i2,e i3 foram incorporadas, enquanto que i4 é reservado ao compilador.

Sintaticamente, a LEAL é uma linguagem do tipo ALGOL /N63/ com palavras reservadas em Português. As principais diferenças em relação ao ALGOL 60 são:

d1) Parâmetros de procedimentos e funções:

Existem duas classes de parâmetros, indicando se um parâmetro pode ou não ser alterado dentro do corpo do procedimento ou função (classe *constante* ou *variável* respectivamente).

d2) Manipulação de elementos de matrizes:

Elementos contíguos de matrizes podem ser movidos ou comparados de uma só vez (comando *mova* e comparação de elementos).

d3) Entrada e Saída:

Comandos extremamente simples de entrada e saída e somente para leitura de cartões e impressão de linhas, porém possibilitando todos os recursos necessários aos objetivos da linguagem.

d4) Comandos iterativos:

Dois comandos iterativos, além do comando *para* correspondendo ao <u>for</u> do ALGOL 60 foram incluídos (comandos *enquanto* e *repita*).

d5) Escape de comandos, procedimentos e funções:

A interrupção da execução de um comando rotulado, um procedimento ou uma função pode ser efetuada (comando *abandone*).

No Apêndice 1 damos os diagramas sintáticos da LEAL.

2 - A Estrutura do Compilador

A estrutura geral do compilador (Fig. 1a) é aproximadamente aquela apresentada em III.2. A sequência das fases é executada ciclicamente, isto é, o compilador apresenta um só passo (Fig. 1b , mesmas convenções da tabela III.1).

2.1 - A Parte Independente de Máquina

Como em III.2, é constituída pelas fases de Análise Léxica, Sintática e de Contexto, Alocação de Espaço e Tradução. As três últimas são na verdade uma só fase, pois constituem uma rotina que é chamada quando a correspondente construção sintática é reconhecida.

2.1.1 - Análise Léxica

Alguns parâmetros que são usados nesta fase para a obtenção dos ítens léxicos como por exemplo o comprimento máximo de identificadores, comprimento máximo de cadeias de caracteres, valor máximo de constantes inteiras e reais, etc..., são dependentes da implementação. Assim, no nosso analisador léxico dispomos de variáveis, inicializadas na declaração, que especificam aqueles valores limites. Para isso fornecemos a documentação necessária à alteração na forma de comentários no próprio compilador.

2.1.2 - Análise Sintática

Usamos a técnica de produção Floyd-Evans convertidas manualmente para código otimizado, isto é, não interpretadas. Uma descrição deste método é dada por Haynes /H69/.

2.1.3 - Análise de Contexto, Alocação de Espaço e Tradução

Esta fase é constituída por um conjunto de rotinas, cada uma correspondente a alguma construção sintática que envolve algumas destas três ações.

A fase de Alocação de Espaço é realizada apenas pela associação de um número na Tabela de Símbolos a cada identificador declarado correspondente à ordem dentro do programa fonte em que foi feita a declaração. Como veremos na seção seguinte, existem instruções de declaração na linguagem intermediária que possibilitam que a reserva efetiva de espaço seja feita na parte dependente de máquina.

Na fase de Tradução são usadas informações das tabelas internas do compilador: pilha semântica, tabela de símbolos e tabela de constantes. A primeira é uma pilha paralela à sintática, contendo informações de contexto e de código gerado do item sintático correspondente. Os elementos necessários à formação de uma cadeia de caracteres que constituem a instrução intermediária são obtidos dessas tabelas. Em seguida é feita uma chamada à rotina correspondente a esta instrução com a cadeia de caracteres como parâmetro.

2.2 - A Parte Dependente de Máquina

Esta parte é constituída por um conjunto de rotinas, uma para cada instrução da Linguagem Intermediária na forma apresentada em III.2.3.1. Estas rotinas são fornecidas vazias e devem ser escritas para cada implementação. Recebem um único parâmetro que é um vetor de caracteres contendo a instrução intermediária.

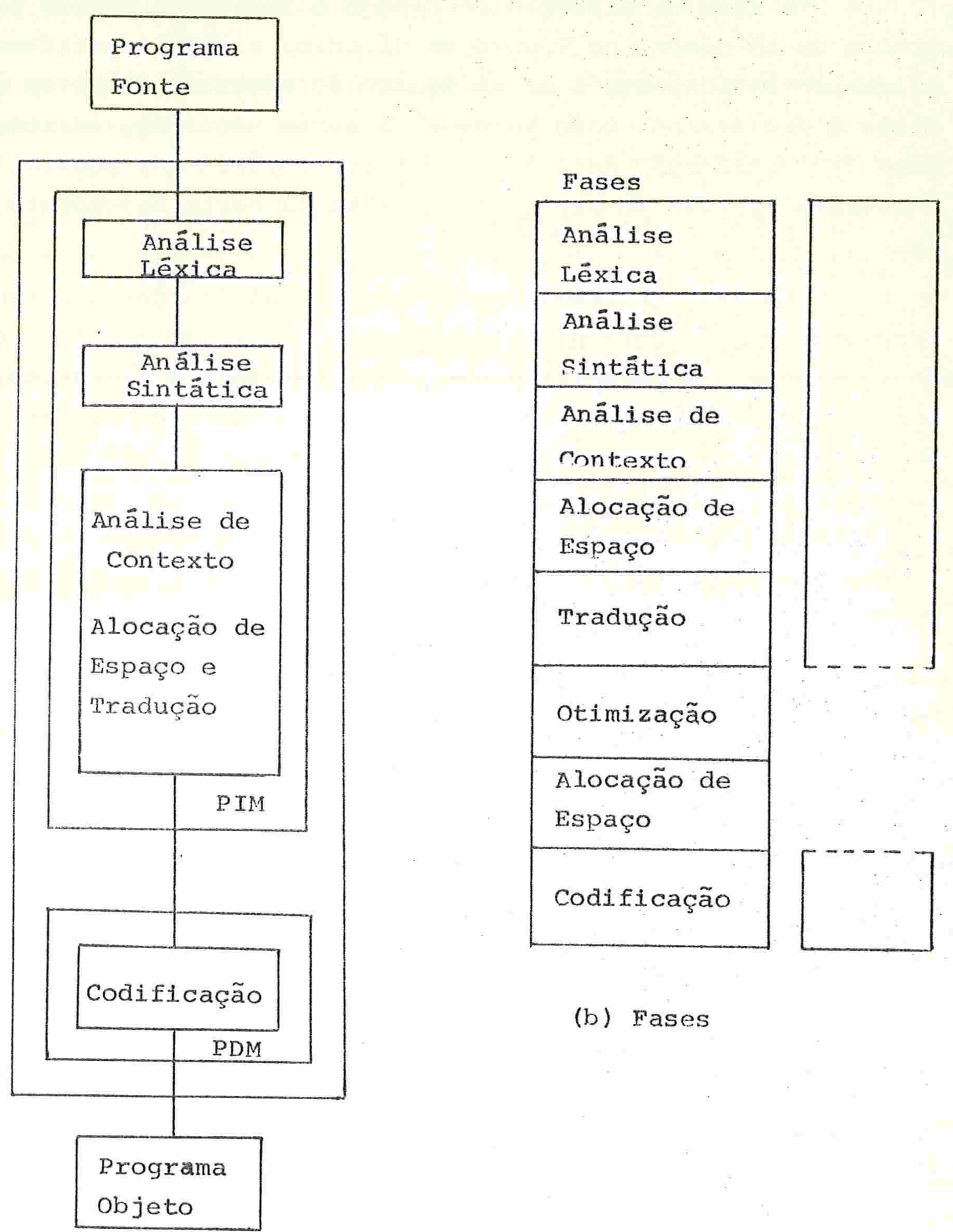


Figura 1 - Compilador da LEAL

3 - A LINGUAGEM INTERMEDIÁRIA PARA GERAÇÃO DE CÓDIGO

As instruções da Linguagem Intermediária operam sobre constantes, variáveis, valores de expressões, endereços e rótulos.

A descrição da Linguagem Intermediária é feita a seguir, de uma forma simbólica. Foi projetada uma forma numérica codificada visando a economia de espaço e aumento de eficiência de conversão para código executável ou de interpretação.

O formato de uma instrução é:

código	operandos

O número de operandos depende da instrução.

Nesta seção apenas descreveremos as instruções deixando para a seção final deste capítulo as justificativas do projeto.

O tipo de dados que ocorre em algumas instruções é indicado por t, onde $t \in \{I,R,C,L\}$, correspondendo aos tipos *inteiro*, *real*, *caráter* ou *lógico*. A classe dos parâmetros formais é indicado por c, onde $c \in \{V,C\}$ correspondendo às classes *variável* ou *constante*. Os operandos que ocorrem nas instruções são numerados. Essa numeração é indicada por x,y, e z e tem relação com a ordem na qual os elementos ocorrem dentro do programa (ver apêndice 2). Se α representa uma construção na LEAL indicamos por $[\alpha]$ sua correspondente tradução na Linguagem Intermediária.

3.1 - Declarações

Correspondente aos comandos de declaração na LEAL existem instruções de declaração de variáveis, que são informações para alocação de espaço, e de procedimentos e funções, indicando apenas o seu início.

3.1.1 - Variáveis Simples

DEC Vtx	Declara variável simples x do tipo t.
DECIN Vtx v	Declara variável simples x do tipo t, inicializando-a com o valor v do tipo t.

3.1.2 - Variáveis Indexadas

DEC n Vtx $i_1\ s_1\ i_2\ s_2\ \ldots\ i_n\ s_n$

Declara variável indexada x do tipo t com n índices, cujos limites inferiores e superiores de cada um são dados pelos pares de inteiros

$$(i_1, s_1), (i_2, s_2), \ldots, (i_n, s_n).$$

DECIN n Vtx $i_1\ s_1\ i_2\ s_2\ \ldots\ i_n\ s_n\ v_1\ v_2\ \ldots\ v_p$

Declara variável indexada como *DEC* e inicializa os elementos em ordem com os valores $v_1\ v_2 \ldots v_p$ do tipo t. Se p é menor que o total de elementos, os demais elementos são inicializados com v_p.
A ordem é aquela obtida variando-se em primeiro lugar os índices mais a direita.

3.1.3 - Parâmetros Formais

DEC n $Fctx$	Declara o parâmetro formal x de classe c e tipo t e com n índices. Sem n, se $n=0$.

3.1.4 - Procedimentos

DEC $PROCx$	Início de declaração do procedimento x.

3.1.5 - Funções

DEC *FUNCtx* Início de declaração da função *x* de tipo *t*.

3.1.6 - Exemplos

Damos a seguir exemplos de declarações em LEAL e correspondente tradução na Linguagem Intermediária.

O número *x* que aparece nas declarações, e dado a cada uma das variáveis, procedimentos e funções dos exemplos, foi escolhido de forma arbitrária.

1º) *int* *I,J;* *real* *A,B;* *car* *C;* *log* *L;*

```
DEC     VI1
DEC     VI2
DEC     VR17
DEC     VR18
DEC     VC9
DEC     VL1
```

2º) *int* *K1* *inic* *0,* *K2* *inic* *-1;*
real *PI* *inic* *3.14;*
car *D* *inic* *"D";*

```
DECIN   VI7    0
DECIN   VI8    -1
DECIN   VR33   314E+1
DECIN   VC22   D
```

3º) *int* *X [0..10], Y[5..9][0..7];*
real *Z[-2..7] ;* *car* *W[-5..-1];*

```
DEC  1  VI15   0   10
DEC  2  VI16   5   9   0   7
DEC  1  VR41   -2  7
DEC  1  VC25   -5  -1
```

4º) <u>*int*</u> *DIG*[*0..9*] <u>*inic*</u> *0,1,2,3,4,5,6,7,8,9;*

<u>*real*</u> *NUM*[*1..2*] [*0..1*] <u>*inic*</u> *0.10,0.11,0.20,0.21;*

<u>*car*</u> *TEXTO*[*1..10*] <u>*inic*</u> *"BLA";*

```
DECIN  1  VI37  0  9  0  1  2  3  4  5  6  7  8  9
DECIN  2  VR42  1  2  0  1  1E-1  11E-2  2E-1  21E-2
DECIN  1  VC28  1  10  BLA
```

5º) <u>*proc*</u> *P(*<u>*cons*</u> <u>*int*</u> *X1,X2,*<u>*real*</u> *X3,* <u>*var*</u> <u>*int*</u> *X4):*

```
DEC    PROC1
DESV   R32
DEC    FCI0
DEC    FCI1
DEC    FCR2
DEC    FVI3
```

6º) <u>*func*</u> <u>*int*</u> *F (*<u>*var*</u> <u>*int*</u> *V1,* <u>*cons*</u> <u>*real*</u> *C1, C2):*

```
DEC    FUNCI4
DESV   R73
DEC    FVI0
DEC    FCR1
DEC    FCR2
```

7º) <u>*proc*</u> *MULT(*<u>*cons*</u> <u>*real*</u> *MAT*[ ] [ ], *VET*[ ] ,

<u>*int*</u> *N,* <u>*var*</u> <u>*real*</u> *RES* [ ]*):*

```
DEC    PROC5
DESV   R19
DEC    2  FCR0
DEC    1  FCR1
DEC    FCI2
DEC    1  FVR3
```

3.2 - Expressões

A sequência de instruções correspondentes ao cálculo de uma expressão corresponde à forma pósfixa desta expressão.

3.2.1 - Início, Fim e Valor de expressão

EXPtx	Indica o início do cálculo da expressão *x* do tipo *t*.
FEXPtx	Indica o final do cálculo da expressão *x* do tipo *t*.
VEXPtx	Usado para referenciar o valor da expressão *x* do tipo *t* calculada anteriormente.

3.2.2 - Instruções Aritméticas

SOMI	soma de inteiros.
SUBI	subtração de inteiros.
MULI	multiplicação de inteiros.
DIVI	divisão de inteiros.
EXPI	exponenciação de inteiros.
NEGI	inversão de sinal de inteiro.
MOD	resto da divisão de inteiros.
SOMR	soma de reais.
SUBR	subtração de reais.
MULR	multiplicação de reais.
DIVR	divisão de reais.
EXPR	exponenciação de reais.
NEGR	inversão de sinal de real.
INT	conversão de real para inteiro truncando.
REAL	conversão de inteiro para real.

3.2.3 - Instruções de comparação

MAI	maior para inteiros.
MEI	menor para inteiros.
IGI	igual para inteiros.
DIFI	diferente para inteiros.
MAIGI	maior ou igual para inteiros.
MEIGI	menor ou igual para inteiros.
MAR	maior para reais.
MER	menor para reais.
IGR	igual para reais.
DIFR	diferente para reais.
MAIGR	maior ou igual para reais.
MEIGR	menor ou igual para reais.
MAC	maior para caracteres.
MEC	menor para caracteres.
IGC	igual para caracteres.
DIFC	diferente para caracteres.
MAIGC	maior ou igual para caracteres.
MEIGC	menor ou igual para caracteres.

3.2.4 - Instruções Lógicas

NÃO	Negação lógica.
E	Conjunção lógica.
OU	Disjunção lógica.

3.2.5 - Variáveis Indexadas em Expressões

INDx v A expressão sendo calculada é usada como índice de ordem x da váriável indexada v que pode ser *Vty* ou *Fcty*.

INDV v VEXPIy

Obtém o valor da variável v que pode ser *Vtx* ou *Fctx*, indexada com o valor da expressão inteira y.

3.2.6 - Exemplos

Damos a seguir, exemplos de expressões em LEAL e correspondente tradução na Linguagem Intermediária.

As variáveis usadas nos exemplos são aquelas declaradas em 3.1.6. Além disso usamos as seguintes constantes com os respectivos códigos.

0	*CI5*
1	*CI6*
1.2	*CR4*
"X"	*CC13*
falso	*CL0*

Os números dados às expressões são arbitrários.

1º) *I + J*

EXPI1
VI1
VI2
SOMI
FEXPI1

2º) *- A/B + 1.2 * * A*

EXPR7
VR17
VR18
DIVR
NEGR
CR4
VR17
EXPR
SOMR
FEXPR7

3º) *I > J*

EXPL5
VI1
VI2
MAI
FEXPL5

4º) *A + 1.2 = B* <u>*e*</u> <u>*não*</u> *I > = <u>inteiro</u> (A) + 1*

EXPL8
VR17
CR4
SOMR
VR18
IGR
VI1
VR17
INT
CI6
SOMI
MAIGI
NAO
E
FEXPL8

5º) *L* <u>*e*</u> <u>*falso*</u> <u>*ou*</u> *C = "X"*

EXPL5
VL1
CL0
E
VC9
CC13
IGC
OU
FEXPL5

$6^{\underline{o}}$) *X* [*I+J*] * *Y* [*I+1*] [*J*]

```
EXPI15
EXPI16
VI1
VI2
SOMI
IND1    VI15
FEXPI16
INDV    VI15    VEXPI16
EXPI17
VI1
CI6
SOMI
IND1    VI16
VI2
IND2    VI16
FEXPI17
INDV    VI16    VEXPI17
MULI
FEXPI15
```

3.3 - Endereços de Variáveis

Algumas instruções da Linguagem Intermediária operam com endereços de variáveis simples ou indexadas. O endereço de uma variável simples *x* do tipo *t* é especificado por *Vtx* e o de um parâmetro formal *x* do tipo *t* e classe *c* por *Fctx*.O endereço de uma variável indexada envolve o cálculo de expressão aritmética inteira. Este endereço é especificado por *ENDtx* que é calculado através da instrução abaixo:

ENDtx	*v*	*VEXPIz*	Calcula o endereço *x* do tipo *t* da variável indexada *v* que pode ser *Vty* ou *Fcty* indexada com o valor da expressão inteira *z*.

3.4 - Comparação de Elementos de Variáveis Indexadas

As seis instruções abaixo comparam o número de elementos especificados pelo valor da expressão inteira x de duas matrizes. A comparação é feita elemento por elemento, a partir dos especificados pelos endereços y e z do tipo t.

op   VEXPtx   ENDty   ENDtz

Representamos por *op* os operadores *MAE*, *MEE*, *IGE*, *DIFE*, *MAIGE*, *MEIGE* correspondendo respectivamente às comparações maior, menor, igual, diferente, maior ou igual e menor ou igual para elementos.

Usando as variáveis de 3.1.6 e constantes de 3.2.6 segue um exemplo:

I <u>*elem*</u>   *X[J] > Y[I+1] [J-1]*

```
EXPL8
EXPI32
VI1
FEXPI32
EXPI33
VI2
IND1        VI15
FEXPI33
ENDI27      VI15      VEXPI33
EXPI34
VI1
CI6
SOMI
IND1        VI16
VI2
CI6
SUBI
IND2        VI16
FEXPI34
ENDI28      VI16      VEXPI34
MAE         VEXPI32   ENDI27    ENDI28
FEXPL8
```

3.5 - Desvios

Estes comandos alteram a sequência normal de execução de um programa na Linguagem Intermediária, para um determinado local do programa identificado por um rótulo, que pode ocorrer antes de qualquer instrução. O rótulo *x* aparece como *Rx*.

3.5.1 - Desvio Incondicional

DESV *Rx* — Desvia para a instrução seguinte ao rótulo *x*.

3.5.2 - Desvio Condicional

DESVC *VEXPLx* *Ry*

Desvia para a instrução seguinte ao rótulo *y* se o valor da expressão lógica *x* for falso.

3.5.3 - Desvio Incondicional de Nível

DESVN *Rx* — Desvia para a instrução seguinte ao rótulo *x*, que pode estar num bloco de nível sintático menor que esta instrução de desvio.

3.6 - Movimentos

ATR *v* *VEXPty* — Atribui o valor da expressão *y* do tipo *t* ao endereço *v* que pode ser uma variável simples *(Vtx* ou *Fctx)* ou indexada (*ENDtx*).

MOVAE *VEXPIx* *ENDty* *ENDtz*

Move o número de elementos especificado pelo valor da expressão inteira *x*, do endereço *z* do tipo *t* para o endereço *y* do tipo *t*, a partir destes.

MOVAV *VEXPIx* *ENDty* *VEXPtz*

Move o número de elementos especificado pelo valor da expressão inteira x, o valor da expressão z do tipo t para o endereço y do tipo t, a partir deste.

MOVAC *ENDtx* n Ctx_1 Ctx_2 ... Ctx_n

Move as n constantes $x_1, x_2, \ldots x_n$ do tipo t para o endereço x do tipo t, a partir deste.

MOVAFE f v *ENDCy*

Move com conversão especificada pelo formato f, os caracteres a partir do endereço y do tipo caráter, para a variável aritmética v que pode ser simples (*Vtx* ou *Fctx*) ou indexada (*ENDtx*); $t=I$ ou R. O formato f é especificado por Iz, Fz_1z_2 ou Ez_1z_2, correspondentes aos valores inteiros, reais fracionários ou reais em notação exponencial respectivamente.

MOVAFV f *ENDCx* *VEXPty*

Move com conversão especificada pelo formato f, o valor da expressão y do tipo t para o endereço x do tipo carater $t=I$ ou R. O formato f é como na instrução anterior.

3.7 - Leitura e Impressão de Valores

LEIAV	*v*	Obtém o próximo valor a ser lido atribuindo-o à variável *v* que pode ser simples (*Vtx* ou *Fctx*) ou indexada (*ENDtx*).
LEIAM	*v*	Obtém os *n* próximos valores a serem lidos atribuindo-os aos elementos da matriz *v* (*Vtx* ou *Fctx*) em ordem (variando primeiro os índices mais a direita). *n* é o número de elementos da matriz.
IMPRV	*VEXPtx*	Imprime o valor da expressão *x* do tipo *t*.
IMPRM	*v*	Imprime todos os elementos da matriz *v* (*Vtx* ou *Fctx*) em ordem (variando primeiro os índices mais à direita).
MUDCAR		O próximo valor a ser lido, será o primeiro do próximo cartão.
MUDLIN		A próxima impressão será feita no início da próxima linha.
MUDPAG		A próxima impressão será feita no início da primeira linha da próxima página.

3.8 - Chamadas de Procedimentos e Funções

As instruções abaixo são usadas para efetuar a chamada a procedimentos e funções.

IPROCx	Indica o início de uma chamada do procedimento *x*.
IFUNCtx	Indica o início de uma chamada da função *x*

do tipo t.

PARCx *p*	O parâmetro x de classe constante, de alguma chamada de procedimento ou função, é dado por p. p pode ser o valor de uma expressão *(VEXPty)* ou uma matriz *(Vtx* ou *FCtx)*.
PARVx *p*	O parâmetro x de classe variável, de alguma chamada de procedimento ou função é dado por p. p pode ser uma variável simples ou uma matriz *(Vty* ou *Fcty)* ou uma variável indexada *(ENDty)*.

CPROCx n $PARc_1 x_1$ $PARc_2 x_2 \ldots PARc_n x_n$

Chamada do procedimento x com parâmetros x_1 de classe c_1, x_2 de classe $c_2, \ldots, x_n$ de classe c_n.

CFUNCtx n $PARc_1 x_1$ $PARc_2 x_2 \ldots PARc_n x_n$

Idem a *CPROCx* para funções.

RPROCx	Retôrno do procedimento x.
RFUNCtx *VEXPty*	Retôrno da função x do tipo t com o valor da expressão y do tipo t. Análoga a *RPROCx* com uma possível diferença no tratamento do valor desta função.

3.9 - Comando de Atribuição

variável := expressão

[*variável*] (cálculo do endereço v de *variável*)

EXPtx

[*expressão*]

FEXPtx

ATR *v* *VEXPtx*

3.10 - Comando Condicional *se*

<u>*se*</u> *expressão* <u>*então*</u> *comando*

EXPLx

[*expressão*]

FEXPLx

DESVC *VEXPLx* *Ry*

[*comando*]

Ry

<u>*se*</u> *expressão* <u>*então*</u> $comando_1$
<u>*senão*</u> $comando_2$

EXPLx

[*expressão*]

FEXPLx

DESVC *VEXPLx* *Ry*

[$comando_1$]

DESV *Rz*

Ry

[$comando_2$]

Rz

3.11 - Comando Iterativo *enquanto*

<u>*enquanto*</u> *expressão* <u>*faça*</u> *comando*

Rx

EXPLy

[*expressão*]

FEXPLy

DESVC VEXPLy Rz

[*comando*]

DESV Rx

Rz

3.12 - Comando Iterativo *repita*

<u>*repita*</u> *comando* <u>*até*</u> *expressão*

Rx

[*comando*]

EXPLy

[*expressão*]

FEXPLy

DESVC VEXPLy Rx

3.13 - Comando Iterativo *para*

<u>*para*</u> *variável* <u>*de*</u> $expressão_1$ <u>*até*</u> $expressão_2$ <u>*incremento*</u> <u>*(decremento)*</u> $expressão_3$ <u>*execute*</u> *comando*

$EXPIx_1$

[$expressão_1$]

$FEXPIx_1$

$ATR \quad VIx_4 \quad VEXPIx_1$

$EXPIx_2$

[$expressão_2$]

$FEXPIx_2$

$EXPIx_3$

[$expressão_3$]

$FEXPIx_3$

Rx_5

$EXPLx_6$

VIx_4

```
VEXPIx₂
MEIGI    (MAIGI)
FEXPLx₆
DESVC    VEXPLx₆   Rx₇
[comando]
EXPIx₈
VIx₄
VEXPIx₃
SOMI     (SUBI)
FEXPIx₈
ATR     VIx₄     VEXPIx₈
DESV   Rx₅
Rx₇
```

3.14 - Comando *mova*

3.14.1 - Elementos de Matrizes

<u>*mova*</u> *expressão* <u>*elementos*</u> $var.indexada_1$:= $var.indexada_2$

EXPIx

[*expressão*]

FEXPIx

[$var.indexada_1$] (cálculo do endereço *ENDty* de $var.indexada_1$)

[$var.indexada_2$] (cálculo do endereço *ENDtz* de $var.indexada_2$)

MOVAE *VEXPIx* *ENDty* *ENDtz*

3.14.2 - Expressões

<u>*mova*</u> $expressão_1$ <u>*elementos*</u> *var.indexada* := $expressão_2$

EXPIx

[$expressão_1$]

FEXPIx

[*var.indexada*] (cálculo do endereço *ENDty* de *var.indexada*)

EXPtz

[*expressão*$_2$]

FEXPtz

MOVAV *VEXPIx* *ENDty* *VEXPtz*

3.14.3 - Constantes

<u>*mova*</u> *var.indexada* := $c_1, c_2, \ldots, c_n$

[*var.indexada*] (cálculo do endereço *ENDtx* de *var.indexada*)

MOVAC *ENDtx* *n* Ctx_1 $Ctx_2 \ldots Ctx_n$

3.14.4 - Formatação

<u>*mova*</u> <u>*com*</u> *formato var.aritmética* := *var.indexada*

[*var.aritmética*](cálculo do endereço *v* de *var.aritmética*)

[*var.indexada*] (cálculo do endereço *ENDCx* de *var.indexada*)

MOVAFE *f* *v* *ENDCx*

<u>*mova*</u> <u>*com*</u> *formato var.indexada* := *expressão aritmética*

[*var.indexada*] (cálculo do endereço *ENDCx* de *var.indexada*)

EXPty

[*expressão aritmética*]

FEXPty

MOVAFV *f* *ENDCx* *VEXPty*

3.15 - Comando *leia*

leia $u_1, u_2, \ldots, u_n$

u_i indica uma variável ou matriz .

Daremos a sequência apenas para u_1, nos demais $u_2, \ldots, u_n$ a sequência é análoga.

Se u_1 é variável:

[u_1] (cálculo do endereço v de u_1)
LEIAV v

Se u_1 é matriz:

LEIAM [u_1]

3.16 - Comando *imprima*

imprima $u_1, u_2, \ldots, u_n$

u_i indica uma expressão ou matriz.
Daremos a sequência apenas para u_1, nos demais $u_2, \ldots, u_n$ a sequência é análoga.

Se u_1 é expressão:

EXPtx
[u_1]
FEXPtx
IMPRV *VEXPtx*

Se u_1 é matriz:

IMPRM [u_1]

3.17 - Comandos mudacartão, mudalinha e mudapagina

mudacartão

MUDCAR

mudalinha

MUDLIN

mudapagina

MUDPAG

3.18 - Comando de chamada de procedimento

nome-de-procedimento $(p_1, p_2, \ldots, p_n)$

p_i acima indica uma expressão ou matriz
q_i abaixo indica o correspondente a p_i

$IPROCx$
$[p_1]$
$PARc_1x_1$ q_1
$[p_2]$
$PARc_2x_2$ q_2
⋮
$[p_n]$
$PARc_nx_n$ q_n
$CPROCx$ $PARc_1x_1$ $PARc_2x_2 \ldots PARc_nx_n$

3.19 - Chamada de Funções em Expressões

A chamada de uma função só pode ocorrer dentro de uma expressão em LEAL.

nome-de-função $(p_1, p_2, \ldots, p_n)$

p_i acima indica uma expressão ou uma matriz
q_i abaixo indica o correspondente a p_i

Lembramos que parâmetros de função devem ser de classe constante.

$IPROCx$
$[p_1]$
$PARCx_1$ q_1
$[p_2]$
$PARCx_2$ q_2
⋮
$[p_n]$
$PARCx_n$ q_n
$CFUNCx$ $PARCx_1$ $PARCx_2 \ldots PARCx_n$

3.20 - Comando abandone

3.20.1 - Abandone Comando

Quando um comando em LEAL possui rótulo, é gerado um rótulo *Rx* no final de sua tradução . Seja α *abandone rótulo* β, um comando

rótulo: α
abandone rótulo
β

[α]
DESVN *Rx*
[β]
Rx

3.20.2 - Abandone Procedimento

A instrução de retôrno de procedimento *RPROC*, é a última instrução gerada para o procedimento. Seja

α *abandone nome-de-procedimento* β,o comando que constitui o corpo do procedimento *x*.

α
abandone nome-de-procedimento
β

[α]
DESVN *Ry*
[β]
Ry
RPROCx

3.20.3 - Abandone Função

Analogamente a 3.20.2

α
<u>*abandone*</u> *nome-de-função* <u>*com*</u> *expressão*
β

[α]
EXPty
[*expressão*]
FEXPty
DESVN *Rz*
[β]
Rz
RFUNCtx *VEXPty*

3.21 - Blocos

Seja <u>*abrabloco*</u> α <u>*fechebloco*</u> um comando

<u>*abrabloco*</u>
α
<u>*fechebloco*</u>

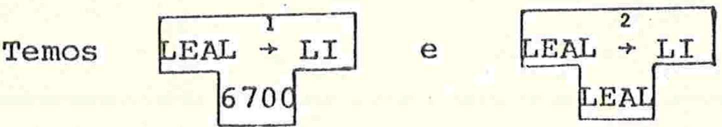
ABRA
[α]
FECHE

4 - O Transporte do Compilador

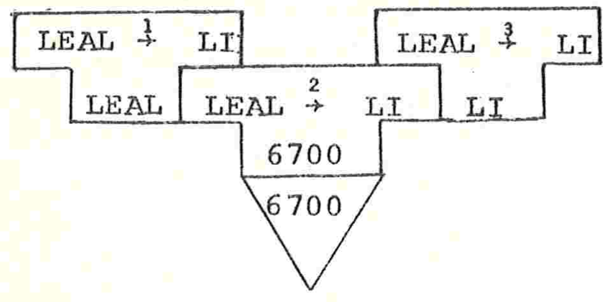
Um compilador de trabalho foi escrito na linguagem ALGOL do computador B6700 gerando a mesma Linguagem Intermediária e que permite que a versão portátil escrita em LEAL seja compilada.

Na descrição abaixo indicaremos por LEAL a linguagem LEAL, LI a Linguagem Intermediária da LEAL e por 6700 o computador B6700 e sua linguagem de máquina.

Temos



Podemos obter uma versão em LI por:



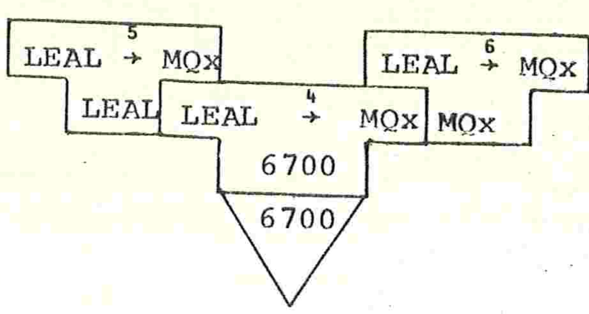
Com os três elementos acima ([1], [2] e [3]) vamos examinar algumas possibilidades de transporte do compilador para uma máquina x. Não estamos no caso apontado em III.3.1.3 pois não dispomos ainda de uma implementação numa máquina y.

4.1 - "Half-bootstrapping" (usando o 6700)

Escrevendo-se cada rotina de geração de código de [1] e [2] (que são vazias nessas versões), para gerar linguagem de máquina de x obtêm-se:

4: LEAL → MQx / 6700 e 5: LEAL → MQx / LEAL

O compilador para x é obtido por



Embora com este método a geração de código de dois compiladores devam ser escritas, a tarefa é essencialmente a mesma nos dois.

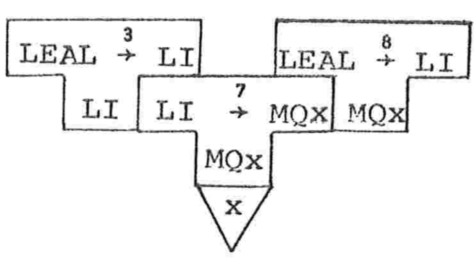
4.2 - "Full-bootstrapping"

Existem duas possibilidades que resultarão em compiladores de um ou dois passos.

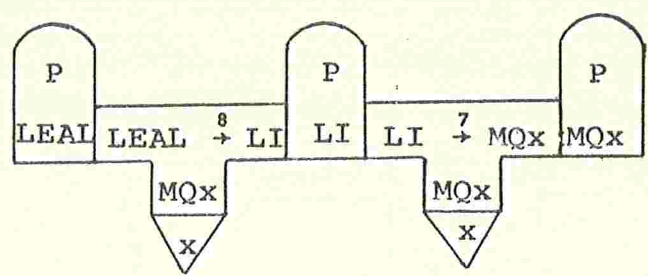
4.2.1 - Dois Passos

Escrevendo-se em alguma linguagem de x um tradutor de LI para a linguagem de x, obtendo-se [LI →7 MQx / MQx] .

Um tradutor de LEAL para LI é obtido por:



A compilação de um programa P é obtida por:



4.2.2 - Um Passo

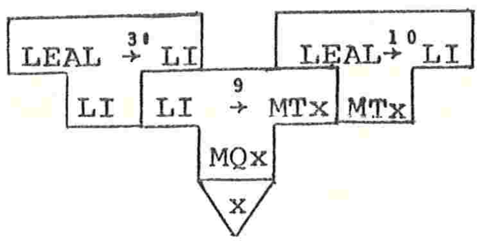
O tradutor [3] pode ser fornecido sem as rotinas de geração de código

LEAL $\xrightarrow{3'}$ LI
LI

Escreve-se em alguma linguagem de x um tradutor de LI para a linguagem de montagem de x, obtendo-se

LI $\xrightarrow{9}$ MTx
MQx

O compilador em linguagem de montagem de x é obtido por:



Escreve-se as rotinas de geração de código G em linguagem de montagem de x,

G
MTx

Agora o tradutor [10] e o conjunto de rotinas G podem ser montados em conjunto e neste caso o nome das rotinas chamadas em [10] deve coincidir com aquelas de G. Podem também ser montados em separado e depois ligados ("link-edited") e neste caso alguma alteração ma-

nual pode ser necessária nos dois módulos para declarar alguns símbolos como externos.

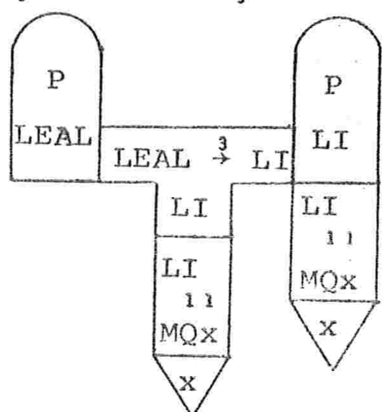
Na verdade, podemos substituir a linguagem de montagem acima por qualquer linguagem existente em x.

4.3 - Interpretação

Se numa implementação em x a eficiência não for um fator essencial (ver II.2.2.2) um interpretador para LI pode ser escrito em alguma linguagem de x, obtendo-se

LI
MQx

A compilação e execução de um programa P é obtida por:



Este método está sendo usado para testar o compilador portátil no computador B6700.

5 - Conclusões e Justificativas

Nesta seção final apresentaremos as justificativas que consideramos importantes no projeto descrito neste capítulo, bem como nossas conclusões sobre o mesmo.

5.1 - Sobre a Linguagem LEAL

Esta linguagem mostrou-se bastante versátil para escrever o compilador. Embora seus objetivos iniciais tivessem sido apenas educacionais, acreditamos que possa ser usada para programar-se "software portátil" para aplicações numéricas /C76/ e não numéricas /W73/.

É interessante notar que comparada com outras linguagens a LEAL é extremamente simples, apesar da versatilidade demonstrada no decorrer do nosso trabalho.

5.2 - Sobre a Estrutura do Compilador

A estrutura do nosso compilador não foge da estrutura geral de um compilador portátil (ver Fig. III.2 e Fig. 1a.). A menos do número de passos, o nosso compilador (1 passo) tem a mesma estrutura do compilador BCPL /R71/ (3 passos) (ver Tabela III.1 e Fig. 1b).

Escolhemos o método III.2.3.1 para implementar a parte dependente de máquina devido aos objetivos do compilador que visavam a ficiência /SS77/, como concluímos no citado ítem.

5.3 - Sobre a Linguagem Intermediária

O objetivo final da Linguagem Intermediária era a sua tradução para um código final, em geral linguagem de máquina de diversos tipos de computadores, e foi este o ponto mais importante no seu projeto. Assim, algumas justificativas sobre a mesma devem ser feitas.

As instruções de declaração de variáveis existem visando uma possível implementação em computadores com alocação dinâmica de espaço (B-6700, HP-3000, etc...). Porém, é importante notar que a menos de procedimentos recursivos com variáveis locais, o espaço máximo necessário pode ser determinado durante a compilação, já que não

hã estruturas com tamanho variável na execução.

Algumas instruções não terão correspondência com instruções em linguagem de máquina numa implementação real (*IPROCx* , *EXPtx* , *FEXPtx*, etc...), porém apresentam informações úteis para a geração de código. Podemos considerá-las como pseudo instruções da Linguagem Intermediária. Tentamos gerar, todas as pseudo-instruções que achamos que poderiam ser usadas em alguma implementação.

Adotamos a forma pós-fixa para a sequência de instruções correspondente a uma expressão, porque esta nos parece a forma que apresenta maior facilidade de tradução tanto para computadores a pilha (B6700, HP300, etc..), como com registradores para as operações (IBM-360, CDC-6000, etc...).

A manipulação de variáveis indexadas depende da forma com que os elementos estão dispostos na memória. As nossas instruções correspondentes (*IND*, *INDV* e *END*) não supõe uma particular disposição, mas foram projetadas visando as duas maneiras mais comuns (dispostos em linhas ou colunas, ou através de vetores de Iliffe /M78/). Devido a variação implícita de índices que é suposta no comando *mova* a primeira forma nos parece a que seria implementada com maior eficiência pelas instruções.

Uma chamada de procedimento ou função correspondente a diversas ações, como guardar endereço de retorno e estados dos registradores, parâmetros para esta chamada, etc.., que são bastante dependentes de máquina devido à diversificação com que isto é feito nos diferentes sistemas. Assim, as instruções correspondentes (*IPROC*, *IFUNC*, *PARC*, *PARV*, *CPROC*, *CFUNC*, *RPROC* e *RFUNC*) não podem ter um significado explícito. Cremos que a informação nelas contida é suficiente para a implementação nos computadores mais conhecidos. Devido à recursividade da chamada de procedimentos da LEAL, pode-se garantir que algumas das instruções acima farão referência a uma pilha contendo informações a serem usados no decorrer da execução.

A redundância de informações apresentada por determinadas instruções foi introduzida visando facilitar a tradução para uma larga variedade de máquinas e permitir maior clareza nos programas em Linguagem Intermediária. Nesse aspecto, os nossos objetivos são os

mesmos apresentados por Coleman-Poole-Waite na família JANUS de Máquinas Abstratas /CPW74/. Com a implementação da LEAL em vários computadores, poder-se-á reduzir eventualmente o código intermediário gerado eliminando-se instruções nunca utilizados.

Embora orientada para as construções da LEAL, suas idéias básicas aplicam-se às construções das linguagens do tipo ALGOL 60. Achamos perfeitamente possível que esta Linguagem Intermediária possa ser extendida com facilidade de forma a servir para geração de código de compiladores de outras linguagens daquele tipo.

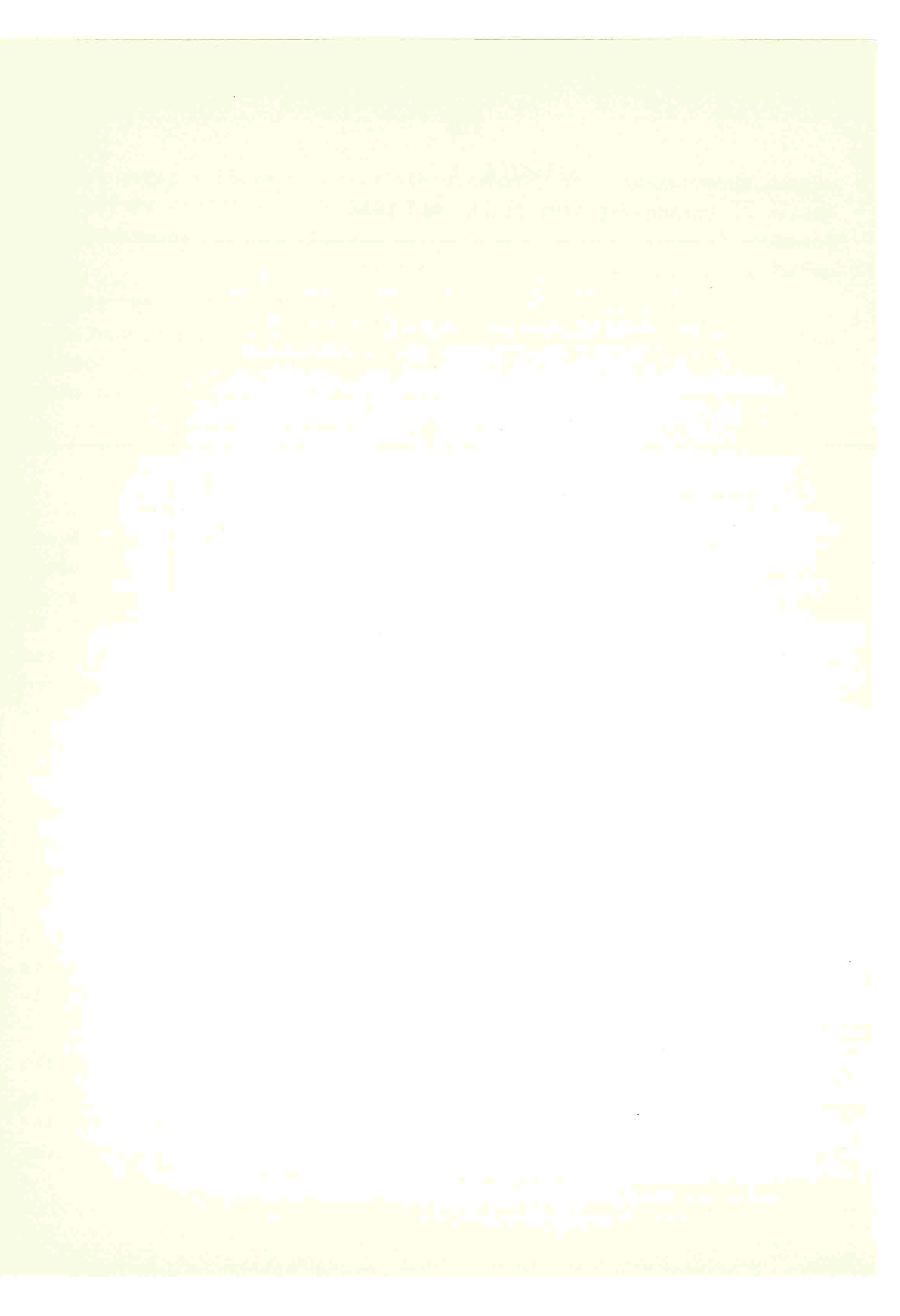
5.4 - Sobre o Transporte do Compilador e Seu Estado Atual

Uma vez que ainda não foram feitos transportes para máquinas reais não dispomos de exemplos concretos para analisar os métodos sugeridos na seção 4. No entanto, podemos usar os argumentos gerais apresentados em III.5 para fazer uma avaliação inicial. Assim do ponto de vista de eficiência os métodos 4.1 e 4.2.2 são os melhores (1 passo). Do ponto de vista de facilidade de implementação, o método 4.2.1 apresenta grande independência do sistema durante a fase de programação. O método 4.3 pode ser bastante útil em implementações experimentais ou mesmo educacionais.

Não é essencial para portabilidade que um compilador seja autocompilável (ver III.3). No entanto, tendo surgido essa possibilidade resolvemos fazê-lo. Dessa forma, dependemos apenas dos nossos instrumentos para o transporte.

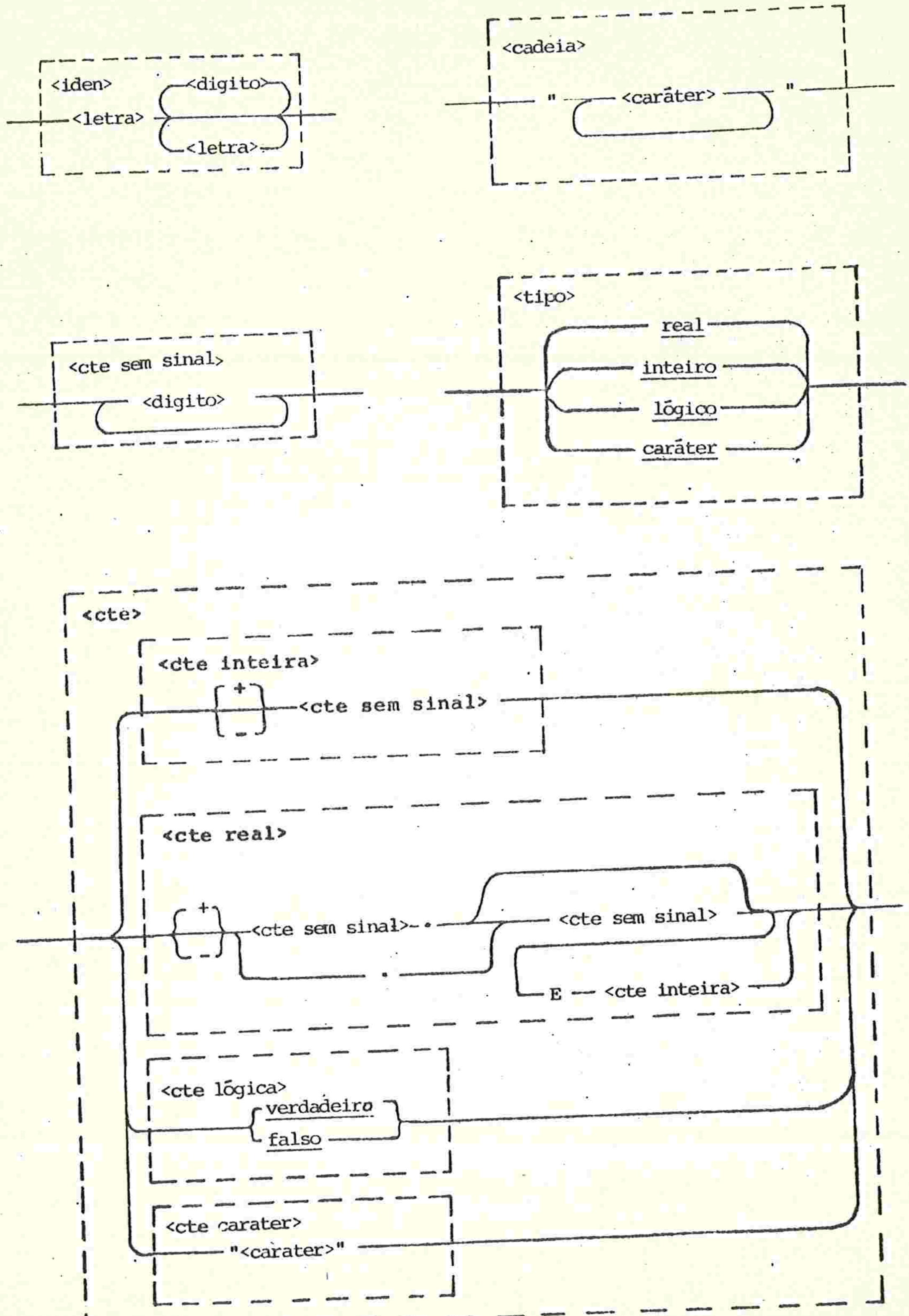
Eventuais modificações podem ser feitas se isto for útil numa determinada implementação. Procuramos fornecer uma versão fonte com estrutura e comentários suficientes para que isso possa ser feito.

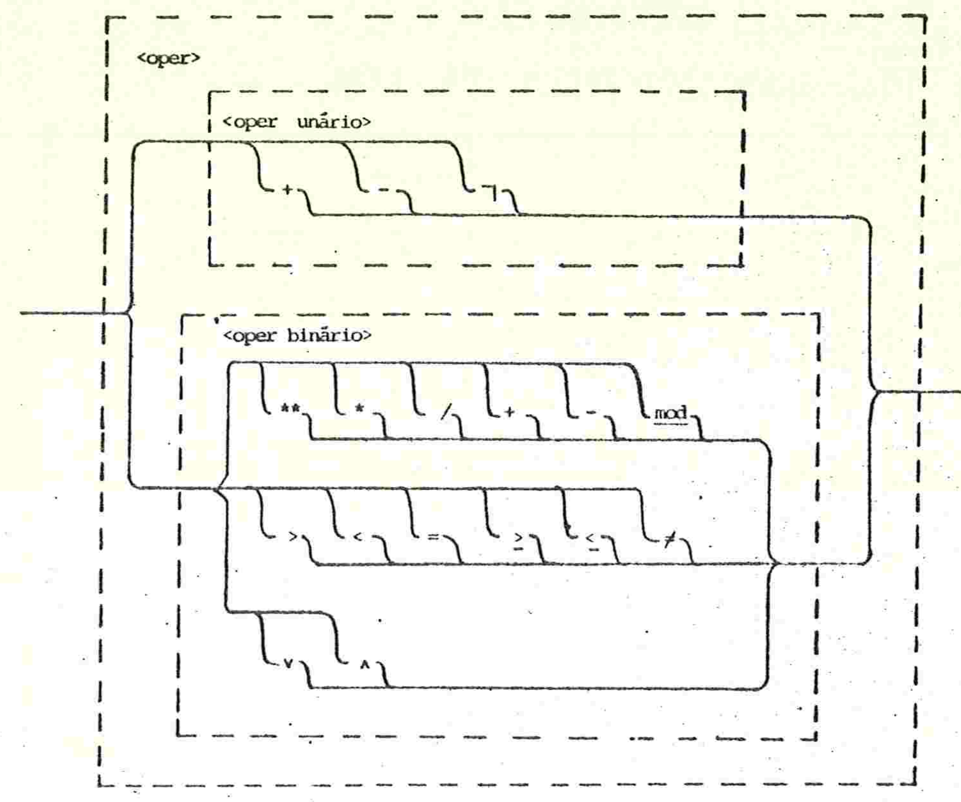
No presente momento, a construção do compilador de trabalho e do compilador portátil ainda não foram concluídas, porém encontram-se em estágio adiantado. Ambos estão sendo desenvolvidos em paralelo. Para testá-los construímos um interpretador da linguagem intermediária no Computador B6700



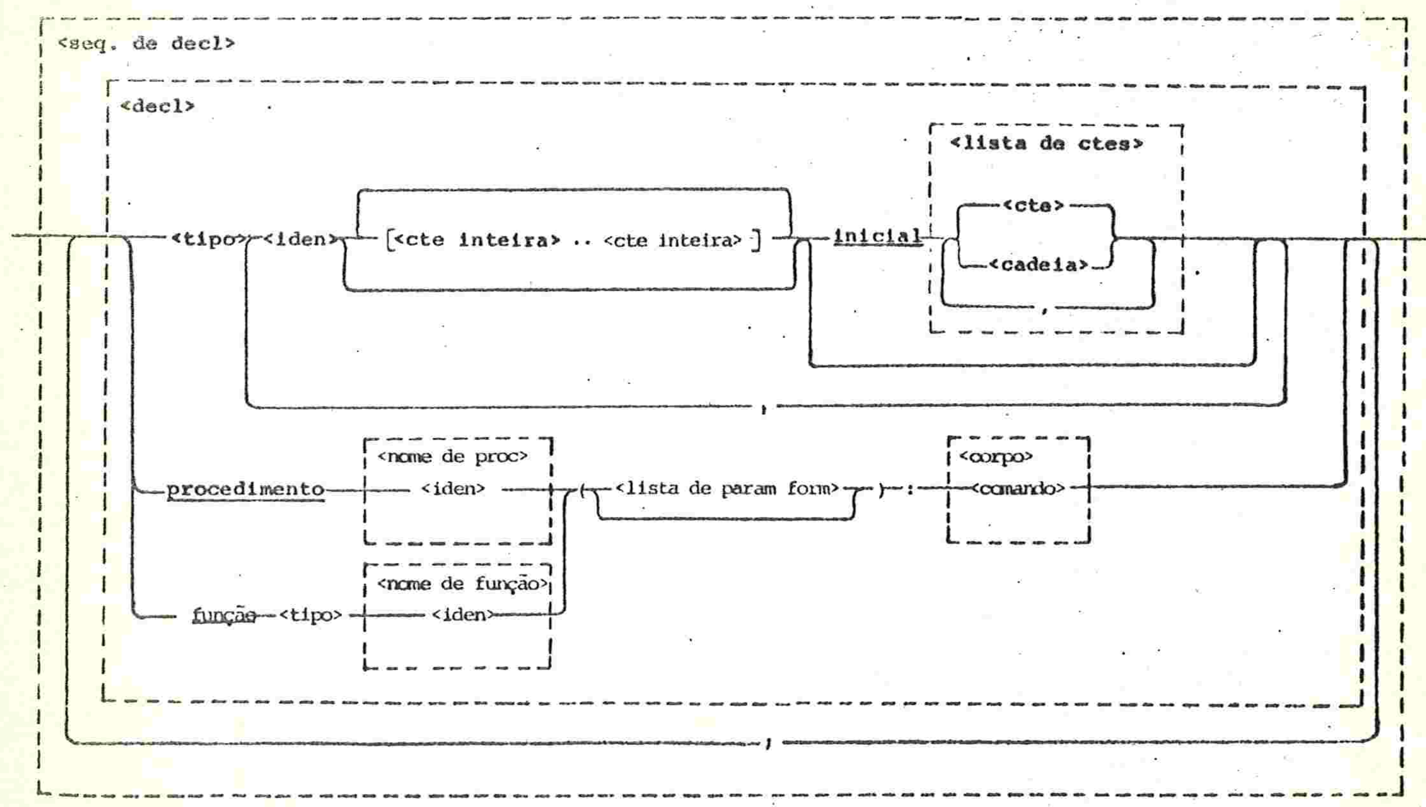
APÊNDICE I

DIAGRAMAS SINTÁTICOS DA LEAL

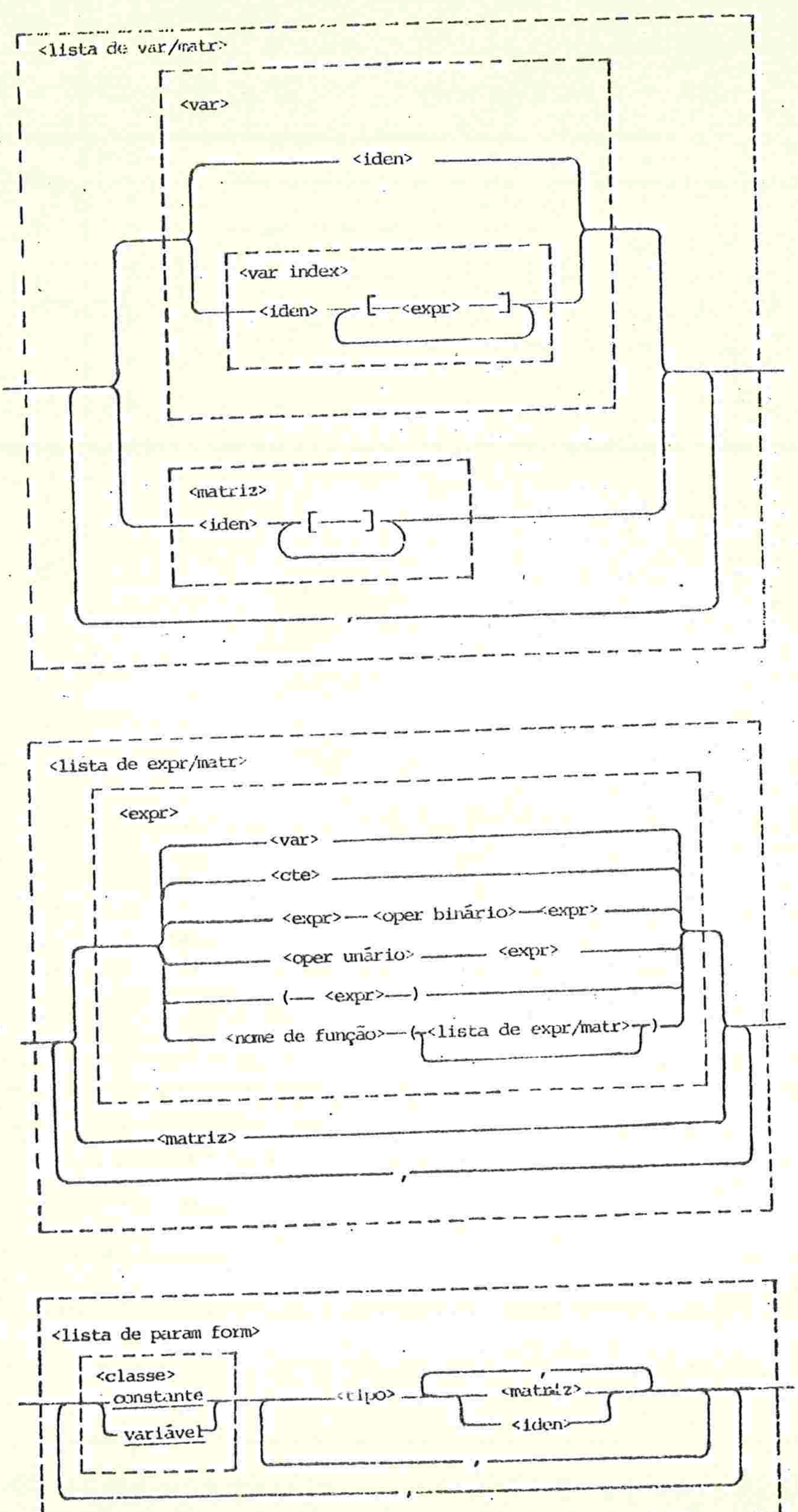


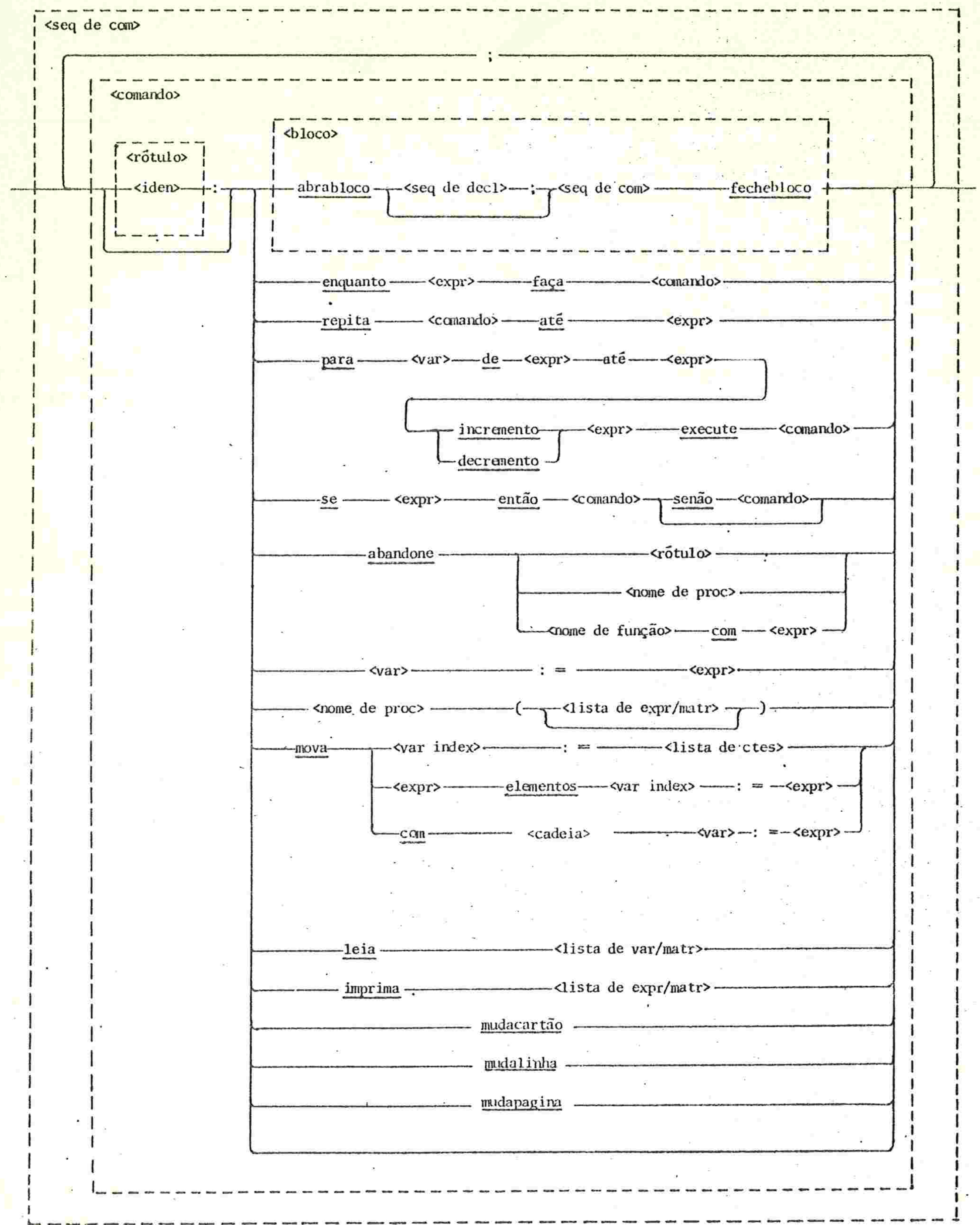


<oper>
<oper unário>
+
-
¬
<oper binário>
**
*
/
+
-
mod
>
<
=
≥
≤
≠
∨
∧



<seq. de decl>
<decl>
<lista de ctes>
<tipo>
<iden>
[<cte inteira> .. <cte inteira>]
inicial
<cte>
<cadeia>
,
procedimento
<nome de proc>
<iden>
(
<lista de param form>
)
:
<corpo>
<comando>
função
<tipo>
<nome de função>
<iden>
;


<lista de var/matr>
<var>
<iden>
<var index>
<iden>
[
<expr>
]
<matriz>
<iden>
[
]
,
<lista de expr/matr>
<expr>
<var>
<cte>
<expr>
<oper binário>
<expr>
<oper unário>
<expr>
(
<expr>
)
<nome de função>
(
<lista de expr/matr>
)
<matriz>
,
<lista de param form>
<classe>
constante
variável
<tipo>
<matriz>
<iden>
,
,
,


<seq de com>
;
<comando>
<bloco>
<rótulo>
<iden>
:
abrabloco
<seq de decl>
;
<seq de com>
fechebloco
enquanto
<expr>
faça
<comando>
repita
<comando>
até
<expr>
para
<var>
de
<expr>
até
<expr>
incremento
decremento
<expr>
execute
<comando>
se
<expr>
então
<comando>
senão
<comando>
abandone
<rótulo>
<nome de proc>
<nome de função>
com
<expr>
<var>
: =
<expr>
<nome de proc>
(
<lista de expr/matr>
)
mova
<var index>
: =
<lista de ctes>
<expr>
elementos
<var index>
: =
<expr>
com
<cadeia>
<var>
: =
<expr>
leia
<lista de var/matr>
imprima
<lista de expr/matr>
mudacartão
mudalinha
mudapagina

APÊNDICE II

1. A CORRESPONDÊNCIA ENTRE AS DECLARAÇÕES EM LEAL E AS DECLARAÇÕES NA LINGUAGEM INTERMEDIÁRIA

Aos vários elementos declarados num programa em LEAL correspondem elementos na linguagem intermediária aos quais é atribuido um número de identificação dependendo do elemento. Esta numeração é feita da seguinte forma:

Variáveis: Numeradas sequencialmente a partir de zero para cada tipo, levando em conta a estrutura de blocos. Assim, variáveis do mesmo tipo declaradas em blocos de mesmo nível, podem ter o mesmo número.

Parâmetros Formais: Numerados sequencialmente a partir de zero para cada procedimento ou função declarada.

Procedimentos e Funções: Numerados sequencialmente a partir de zero.

Rótulos: Numerados sequencialmente a partir de zero.

O exemplo abaixo mostra as declarações de um programa em LEAL e as correspondentes declarações na linguagem intermediária.

```
abra
   int      I, J;
                 DEC     VI0
                 DEC     VI1
   real     X, Y [0..5];
                 DEC     VR0
                 DEC     1     VR1     0     5
   log      L;
                 DEC     VL0
   carater C    inic "*"
                 DECIN   VC0     *
   proc     P (cons int X1, X2, real X3, var int X4):
                 DEC       PROC0
                 DESV      R0
                 DEC       FCI0
                 DEC       FCI1
                 DEC       FCR2
                 DEC       FVI3
     .           .
     .           .
     .           .
                 R0
   proc     MULT (cons real MAT [ ] [ ], VET [ ], int N, var real RES[ ]
                 DEC       PROC1
                 DESV      R1
                 DEC       2     FCR0
                 DEC       1     FCR1
                 DEC       FCI2
                 DEC       1     FVR3
     .           .
     .           .
     .           .
                 R1
```

```
func      int  F(var int  V1, cons real C1, C2):
                 DEC      FUNCIO
                 DESV     R2
                 DEC      FVIO
                 DEC      FCR1
                 DEC      FCR2
   .             .
   .             .
   .             .
                  R2
   .             .
   .             .
abra.            .
   int      Z1; real W1;
                 DEC      VI2
                 DEC      VR2
   .             .
   .             .
   .             .
feche

abra
   int  K;  real  Z;
                 DEC      VI2
                 DEC      VR2
   .             .
   .             .
   .             .
feche
   .
   .
   .
feche
```

2. EXPRESSÕES, ENDEREÇOS E CONSTANTES

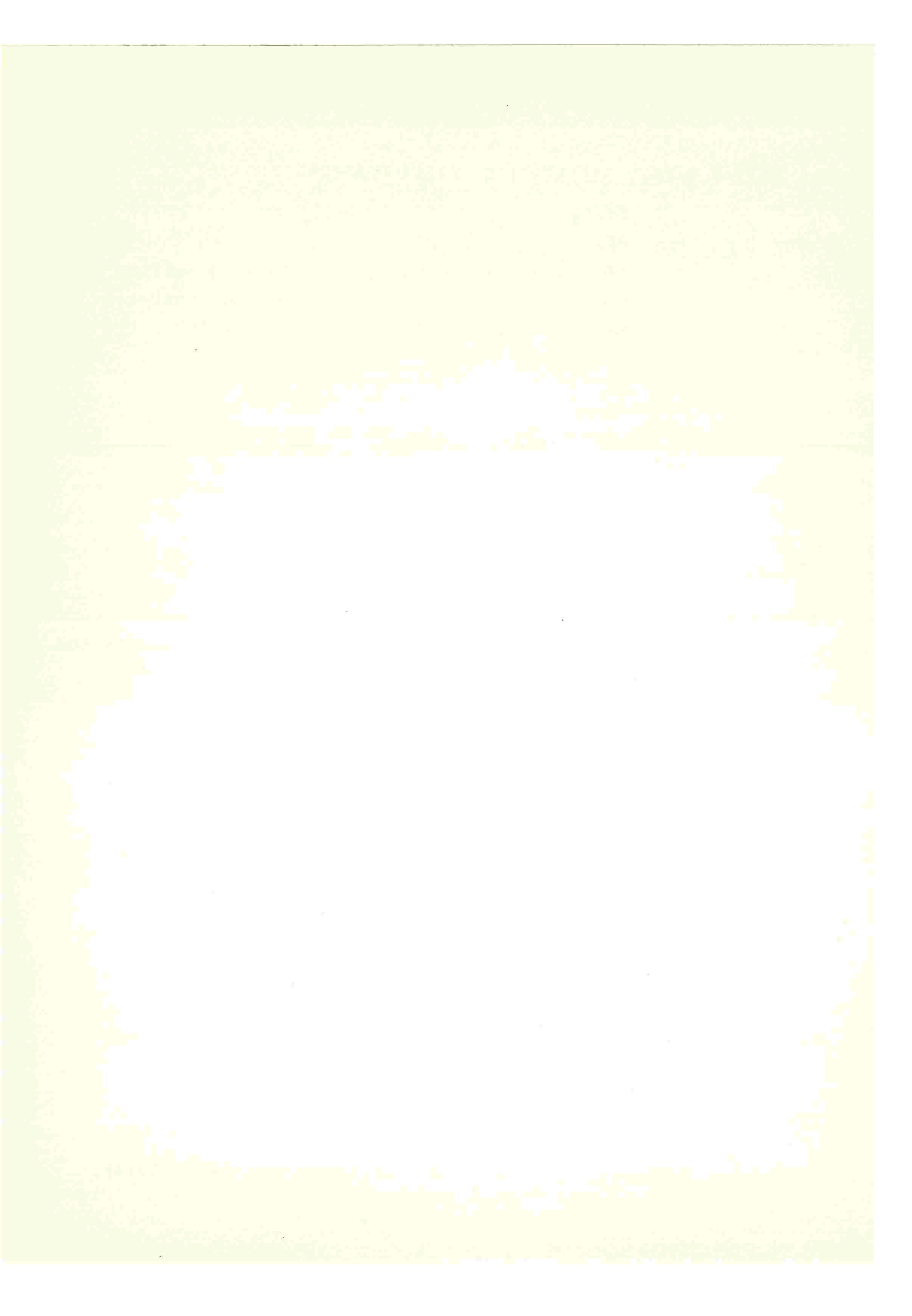
As expressões e endereços calculados no decorrer do programa são numeradas sequencialmente a partir de zero. As constantes sã também numeradas a partir de zero para cada tipo.

```
abra
   int    I, J;
                 DEC     VI0
                 DEC     VI1
   real   X[1..10], A;
                 DEC     1  VR0   1   10
                 DEC     VR1
  I:= 2;
                 EXPI0
                 CI0
                 FEXPI0
                 ATR     VI0     VEXPI0
  A:= 2.7;
                 EXPR1
                 CR0
                 FEXPR1
                 ATR     VR1     VEXPR1
  X[I+1]:= A;
                 EXPI2
                 VI0
                 CI1
                 SOMI
                 IND1    VR0
                 FEXPI2
                 END0    VR0     VEXPI2
                 EXPR3
                 VR1
                 FEXPR3
                 ATR     END0    VEXPR3
```

```
X[I]:= 5.74;

                EXPI4
                VI0
                IND1      VR0
                FEXPI4
                END1      VR0      VEXPI4
                EXPR5
                CR1
                FEXPR5
                ATR      END1      VEXPR5
```

<u>*feche*</u>



REFERÊNCIAS BIBLIOGRÁFICAS

/A76/ Aird, T.J.
The IMSL Fortran Converter: An Aproach to Solving Portability Problems. Portability of Numerical Software Lecture Notes in Computer Science, V57, 368-388.
Springer-Verlag (1976).

/B61/ Bratman, H.
An Alternate Form of the "UNCOL Diagram".
CACM, V4, N3, 142 (Mar 61).

/B67/ Brown, P.J.
The ML/I Macro Processor
CACM, V10, N10, 618-626 (Oct 1964).

/B69/ Brown, P.J.
Using a Macro Processor to Aid Software Implementation
The Computer Journal, V12, N4, 327-331 (Nov 1969).

/B72/ Brown, P.J.
Levels of Language for Portable Software
CACM, V15, N12, 1059-1062 (Dec 1972).

/B77/ Brown, P.J.
Basic Implementation Concepts. Software Portability
Chapter II.B, 20-30, Cambridge University Press (1977).

/C58/ Conway, M.E.
Proposal for an UNCOL
CACM, V1, N10, 5-8 (Oct. 1958)

/C68/ American National Standard COBOL ANSI X3.23 - 1968
American National Standard Institute (1968).

/CPW74/ Coleman, S.S., Poole, P.C., Waite, W.M.
The Mobile Program System, JANUS
Software - Pratice and Experience, V4, 5-23 (1974).

/C76/ Cowell, W. (editor)
Portability of Numerical Software
Lecture Notes in Computer Science, V57 (1976).

/ES70/ Earley, J., Sturgis, H.
A Formalism for Translator Interactions
CACM, V13, N10, 607-617 (Oct 1970).

/F66/ USA Standard FORTRAN
USAS X3.9-1966
USA Standards Institute, N.Y. (1966).

/F69/ CAL 6400 FORTRAN Guide
CDC 6400 FORTRAN
University of California, Computer Center, Berkeley (1969).

/F72/ Burroughs B6700/B7700
FORTRAN Reference Manual
Form 5000458 (July 1972).

/F76/ American National Standards Institute - X3J3 Subcommitte
The Draft Proposed ANSI FORTRAN
SIGPLAN Notices, V11, N3, (mar 1976)

/G62/ Grau, A.A.
A Translator-Oriented Symbolic Programming Language
JACM, V9, 480-487 (1962)

/G71/ Gries, D.
Compiler Construction for Digital Computers
John Wiley & Sons, Inc. (1971)

/G72/ Griswold, R.E.
The Macro Implementation of SNOBOL 4
W.H. Freeman, San Francisco (1972)

/G77a/ Griswold, R.E.
The Macro Implementation of SNOBOL 4
Software Portability, Chapter VI.B, 180-191
Cambridge University Press (1977)

/G77b/ Gardner, P.J.
A Transportation of ALGOL68C
SIGPLAN Notices, V12, N6, 95-101 (June 1977)

/H62/ Halstead, M.H.
Machine-Independent Computer Programming
Spartan Books, Washington, D.C. (1962)

/H65/ Halpern, M.I.
Machine Independence: Its Tecnology and Economics
CACM, V8, N12, 782-785 (Dec 1965)

/H69/ Haynes, H.R.
An Optimizing Compiler for an Extended Version of the
Floyd-Evans Production Language
TRM-12, Computation Center, The University of Texas at
Austin (Mar 1969)

/HLL73/ Hassit A., Lages Shulte, J.W., Lyon, L.E.
Implementation of a High Level Language Machine
CACM, V16, N4, 199-212 (April 1973)

/H74/ Halstead, M.H.
A Laboratory Manual for Compiler and Operating System
Implementation
American Elsevier Publishing Company, INC (1974)

/H76/ Hemker, P.W.
Criteria for Transportable Algol Libraries.Portability
of Numerical Software
Lectures Notes in Computer Science, V57, 145-157,
Springer-Verlag (1976)

/IK77/ Inglis, J., King, P.J.H.
Data Portability
Software Portability, Chapter VI.E, 213-233
Cambridge University Press (1977)

/K64/ Knuth, D.E. et alli
A Proposal for Input-Output Convertions in ALGOL60
CACM, V7, N5, 273-283 (May 1964)

/K76/ Kernighan, B.W.
RATFOR - A preprocessor for Rational FORTRAN
Software-Pratice and Experience, V5, 395-406 (1975)

/LP78/ Lecarme, O., Peyrolle - Tomas, M.C.
Self Compiling Compilers: An Apraisal of Their Implementation and Portability
Software - Pratice and Experience, V8, 149-170 (1978)

/M70/ Morris, J.H.
Design for a Run-Time System of ALGOL60
CS106 Notes - California University at Berkeley (1970)

/MHW70/ McKeeman, W.M., Horning, J.J., Wortman, D.B.
A Compiler Generator
Prentice Hall, Englewood Cliffs, N.J. (1970)

/M78/ Melo, I.S.H.
Alguns Tõpicos de Compilação e uma Implementação da Linguagem LAPA para o Computador PADE.
IMEUSP, Tese de Mestrado (Set. 1978)

/N63/ Naur, P. (editor)
Revised Report on the Algorithmic Language ALGOL60
CACM, V6, N1, 1-17 (Jan. 1963)

/NPW72/ Newey, M.C., Poole, P.C., Waite, W.M.
Abstract Machine Modelling to Produce Portable Software - A Review and Evaluation Software - Pratice and Experience, V2, 107-136 (1972)

/NG76/ Nagel, H.H., Grosse-Lindeman, C.O.
Postlude a PASCAL Compiler Bootstrap on a DEC System - IO
Software - Pratice and Experience, V6, 29-42 (1976)

/NW78/ Neal, D., Wallentine, V.
Experiences with the Portability of Concurrent PASCAL
Software-Pratice and Experience, V8, 341-353 (1978)

/OW69/ Orgass, R.J., Waite, W.M.
A Base for a Mobile Programming System
CACM, V12, N9, 507-510 (Sept 1969)

/PW69/ Poole, P.C., Waite, W.M.
Machine Independent Software
Proc. ACM, Second Symposium on Operating Systems Principles, 19-24 (1969)

/PW70/ Poole, P.C., Waite, W.M.
Input/Output for a Mobile Programming System
Software Engineering, Tou, J.T. (ed), V1, 167-177, Academic Press (1970)

/P71/ Poole, P.C.
Hierarchical Abstract Machines
Symposium on Software Engineering hold at Culham Laboratory, 1-9 (April 1971)

/P73/ Pasko, H.J.
A Pseudo-Machine for Code Generation
Technical Report CSRG-30 (Dec 1973)
Master Thesis - DCS - University of Toronto

/P74/ Poole, P.C.
Portable and Adaptable Compilers
Compiler Construction, An Advanced Course
Lecture Notes in Computer Science, V21, 427-497
Springer-Verlag (1974)

/PD78/ Pyster, A. and Dutta, A.
Error - Checking Compilers and Portability
Software - Pratice and Experience, V8, 99-108 (1978)

/QW72/ Welsh, J., Quinn C.
A PASCAL Compiler for ICL 1900 Series Computers
Software - Pratice and Experience, V8, 149-170 (1978)

/RR64/ Randel, B., Russell, L.J.
ALGOL60 Implementation
Academic Press, New York (1964)

/R69/ Richards, M.
BCPL: A Tool for Compiler Writing and System Programming
Proceedings of the SJCC, V34, 557-566 (1969)

/R71/ Richards, M.
The Portability of the BCPL Compiler
Software - Pratice and Expecience, V1, 135-146 (1971)

/R74/ Richards, M.
Bootstrapping the BCPL Compiler Using INTCODE
Machine Oriented Higher Level Languages,265-272 (1974)

/RS76/ Russell, D.L., Sue, J.Y.
Implementation of PASCAL Compiler for the IBM-360
Software - Pratice and Experience, V6, 371-376 (1976)

/R77a/ Richards, M.
The Implementation of BCPL
Software Portability, Chapter VI.C, 192-202, Cambridge University Press (1977)

/R77b/ Rosin, R.F.
A Graphical Notation for Describing System Implementation
Software - Pratice and Experience, V7, 239-250 (1977)

/S58/ Strong, J. et alli
The Problem of Programming Communication with Changing Machines - A Proposed Solution
CACM, V1, N8-9, 12-18, 9-15 (Aug-Sept 1958)

/S61a/ Sibley, R.A.
The SLANG System
CACM, V4, N1, 75-84 (Jan 1961)

/S61b/ Steel, T.B. Jr.
UNCOL; The Mith and the Fact
Annual Review in Automatic Programming, V2, 325-344, Pergamon Press (1961)

/S65/ Stratchey, C.
A General Purpose Macrogenerator
The Computer Journal, V8, N3, 225-241 (Oct 1965)

/S76a/ Smith, B.T.
FORTRAN Poisoning and Antidotes
Portability of Numerical Software
Lecture Notes in Computer Science, V57, 179-256, Springer-Verlag (1976)

/S76b/ Setzer, V.W.
An Educational Programming Language for Mini-Computers
Anais do Simpósio Internacional sobre Metodologia de Software e Hardware
Rio de Janeiro, 41-44 (1976)

/SS77/ Setzer, V.W., Sanches, M.M.
A Linguagem LEAL para Ensino Básico de Computação
RT-MAP-7704-IMEUSP (Dez 1977)

/WH66/ Wirth, N., Hoare, C.A.R.
A Contribution to the Development of ALGOL
CACM, V9, N6, 413-434 (June 1966)

/W67a/ Weber, H.
A Microprogrammed Implementation of Euler on IBM System/360 Model 30
CACM, V10, N9, 549-558 (Sept 1967)

/W67b/ Waite, W.M.
A Language Independent Macro Processor
CACM, V10, N7, 433-440 (July 1967)

/W70a/ Waite, W.M.
Building a Mobile Programming System
The Computer Journal, V13, N1, 28-31 (Fev 1970)

/W70b/ Waite, W.M.
The Mobile Programming System: STAGE2
CACM, V13, N7, 415-421 (July 1970)

/W71a/ Waite, W.M.
Input/Output Conventions for Abstract Machines
Symposium on Software Engineering hold at Culham Laboratory, 55-64 (April 1971)

/W71b/ Wilcox, T.R:
Generating Machine Code for High-Level Programming Languages
TR 71-103, Cornell University, Ph.D. Thesis (Sept - 1971)

/W71c/ Wirth, N.
The Design of a PASCAL Compiler
Software - Pratice and Experience, V1, N3, 309-333 (1971)

/W73/ Waite, W.M.
Implementing Software for Non-Numerical Applications
Prentice-Hall, Englewood Ciffs, N.J. (1973)

/W76/ Waite, W.M.
Intermediate Languages: Current Status

Portability of Numerical Software
Lecture Notes in Computer Science, V57,269-303 , Springer-Verlag (1974)

/W77/ Wichmann, B.A.
Use of ALGOL 60
Software Portability, Chapter VI.A, 171-179, Cambridge University Press (1977)

/WH78/ Waite, W.M.,Haddon, B.K.
Experience with the Universal Intermediate Language JANUS
Software - Pratice and Experience, V8, 601-616 (1978)