PROGRAMAÇÃO MULTI-LINGUAGEM: UMA PROPOSTA

REGINA CÉLIA DE SOUZA PEREIRA

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

APROVADA POR:

PROF. SÉRGIO EDUARDO RODRIGUES DE CARVALHO
(PRESIDENTE)

PROFª SUELI BANDEIRA TEIXEIRA MENDES

PROFª ANA REGINA CAVALCANTE DA ROCHA

PROFª DÓRIS FERRAZ DE ARAGON

PROF. SÉRGIO DE MELLO SCHNEIDER

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 1988

PEREIRA, REGINA CÉLIA DE SOUZA

Programação multi-linguagem: uma proposta (Rio de Janeiro) 1988.

xii, 171 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 1988)

Tese - Universidade Federal do Rio de Janeiro, COPPE,

1. Linguagens de Programação I. COPPE/UFRJ II. Título (série)

Ao Lucas e à Luciana.

AGRADECIMENTOS

Ao professor Sérgio Eduardo Rodrigues de Carvalho pela orientação.

À professora Sueli Mendes pela co-orientação e o incentivo.

À professora Ana Regina Cavalcante da Rocha, (COPPE/UFRJ), bem como aos amigos da CFA, em particular à Celi Bonfim, pelo apoio e incentivo.

Resumo da Tese Apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor Em Ciências (D. Sc.).

PROGRAMAÇÃO MULTI-LINGUAGEM: UMA PROPOSTA

Regina Célia de Souza Pereira

Setembro de 1988

Orientador: Sérgio Eduardo Rodrigues de Carvalho
Programa: Engenharia de Sistemas e Computação

Este trabalho discute a reunião de trechos de código (módulos) escritos em diferentes linguagens. através das técnicas de programação em grande escala, permitindo segurança, confiabilidade e modularidade em programas assim construídos.

Com essa finalidade, discutem-se as formas de passagem de um módulo para outro de objetos tais como variáveis, constantes, procedimentos, funções, tipos e unidades de encapsulamento, de forma a permitir o uso em um módulo de objetos declarados ou definidos em outros. Discutem-se também os problemas relativos à implementação da passagem desses objetos.

Finalmente, são propostas formas de organizar um ambiente de programação multi-linguagem, em que essas facilidades mencionadas se tornem disponíveis, e formas de construir interfaces amigáveis entre tais ambientes e seus usuários.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfilmment of the requirements for the degree of Doctor (D.Sc.).

MULTI-LANGUAGE PROGRAMMING: A PROPOSAL

Regina Célia de Souza Pereira

September, 1988

Chairman: Sérgio Eduardo Rodrigues de Carvalho
Department: Systems Engineering and Computing

This dissertation discusses the possibility of putting together sections of code (modules) written in different languages, by means of the techniques of large scale programming, in such a way that allows programs built in this manner to be safe, reliable, modular.

Thus, the ways of passing objects from one module to another, are discussed. Objects included are such as variables, constants, procedures, functions, types and packages. The corresponding implementation issues are also presented.

In conclusion, a proposed general organization of multi-language environments is shown, with provisions for friendly interfaces between these environments and their users.

ÍNDICE

INTRODUÇÃO

Fazer um programa é uma tarefa que, em geral, envolve o uso de uma linguagem de programação apenas. Existem aplicações entretanto, em que o uso de mais de uma linguagem se justificaria. Por exemplo, em certas situações, diferentes partes de um problema seriam melhor modeladas em linguagens diferentes. Isso porque as linguagens de programação, mesmo as mais modernas, nem sempre conseguem satisfazer todas as necessidades que surgem em programação. O uso de linguagens diferentes em um programa se justificaria, ainda, para dar liberdade aos programadores envolvidos em um projeto de desenvolverem a sua parte na programação da forma que mais lhes conviesse, ou ainda para aproveitar software que já existisse pronto, visto que o custo de reprogramação é muito alto (WOLBAG [1] e VOUK [2]). Com certeza, encontraríamos muitas outras justificativas para programação com mistura de linguagens, porém, convém ressaltar que muitos são os problemas decorrentes desse tipo de programação, sobretudo aqueles que dizem respeito à execução do código objeto dos trechos escritos em linguagens diferentes como um único programa.

Tentativas de se fazer programas com mistura de linguagens vêm acontecendo há algum tempo e na literatura encontram-

se referências a vários trabalhos desenvolvidos com essa finalidade. Nos *Bell Laboratories* (GENTLEMAN [3]), foi desenvolvida uma biblioteca de programas com interfaces para FORTRAN, Algol 60 e PL/1. O usuário dessa biblioteca pode escolher uma entre essas linguagens para escrever suas rotinas.

Algumas implementações de Algol 60, FORTRAN, PL/1, Pascal, (RIS [4]), têm características que permitem o uso de rotinas escritas em outras linguagens, a mais freqüente sendo a linguagem *assembly* da máquina utilizada. No sistema operacional UNIX, (KERNIGHAN [5]), é possível misturar linguagens de alto nível tais como C, Pascal, FORTRAN. Referências a outros esforços de pesquisas, nesse sentido, são encontrados em X3 [6], DARONDEAU [7] e EINARSSON [8].

Em geral, a mistura de linguagens, nos sistemas mencionados anteriormente, ocorre a nível de código, e os trechos de código que devem funcionar em conjunto apresentam código compatível, tendo sido gerados para a mesma máquina usando os serviços de um mesmo sistema operacional. Assim, a junção ocorre como uma simples transferência da execução de um trecho de código para outro, sem que se faça qualquer espécie de teste. Ao programador cabe garantir a existência de áreas de dados comuns onde se faz a transmissão de parâmetros e se localizam as variáveis globais, de acordo com as características de funcionamento do sistema utilizado.

Com o objetivo de aumentar a confiabilidade de programação multi-linguagem, este trabalho apresenta os princípios para a especificação de Ambientes onde a mistura de linguagens de alto nível é controlada a nível de programação fonte, usando técnicas de programação em larga escala. Esses princípios in-

cluem soluções para programação multi-linguagem a nível do usuário, a nível do código gerado e a nível de segurança na execução de programas assim construídos. Nesses Ambientes o programador define trechos de código (com base em unidades de programa), aqui denominados "módulos", de maneira que cada um deles é desenvolvido e validado separadamente. Para cada módulo o programador especifica quais os objetos (dados e operações) que descrevem a comunicação com os outros módulos que a ele se reunirão para formar um programa. A compilação desse programa inclui testes semânticos entre esses objetos para garantir sua validade.

A linguagem Mesa (MITCHELL [9]) oferece facilidades para programação em larga escala, com módulos escritos nessa linguagem e validados separadamente. Existe, entretanto, definida, à parte, a linguagem de configuração Mesa, através da qual o usuário do Sistema pode definir uma configuração, isto é, pode indicar a montagem de módulos formando um programa em Mesa. Um dos princípios propostos para o Ambiente multi-linguagem é que a junção dos módulos em um programa seja feita, também, por meio de uma linguagem de configuração (abreviada LC), sendo que nesse caso, os módulos são escritos em linguagens diferentes. Além da LC são especificados outros critérios viabilizando essa mistura.

Podemos afirmar que em tais Ambientes, a programação em larga escala se beneficia da mistura de linguagens, uma vez que é definido também, um mecanismo de encapsulamento característico de linguagens de programação de grandes sistemas, permitindo que a técnica de projeto dos módulos incorpore princípios de Engenharia de Software, tais como o da abstração e o de "esconder

informação", fazendo com que o seu programador possa separar o que o módulo faz e envia para uso de outros módulos (interface), de como ele implementa o que faz (corpo do módulo). Esse mecanismo encapsula módulos em qualquer uma das linguagens misturáveis. Dessa maneira, a possibilidade de inclusão em módulos de subprogramas e outros objetos declarados em linguagens que, como Pascal e FORTRAN, não possuem mecanismos de encapsulamento, cria um mecanismo, de certa forma equivalente, que se torna um recurso poderoso, na programação multi-linguagem em larga escala.

Além da Linguagem de Configuração deve ser incluído em um tal Ambiente um conjunto de linguagens misturáveis, com seus compiladores, algumas facilidades de biblioteca, entre as quais convém destacar a Biblioteca de Módulos, onde são mantidos módulos, escritos originalmente em diferentes linguagens, prontos para serem (re-)usados, além de ferramentas com funções específicas. Algumas dessas ferramentas serão descritas como componentes de uma Máquina Abstrata, responsável pela junção dos módulos e por sua execução como um programa únicos.

É possível ainda, nesses Ambientes, a (re-)utilização de pacotes de Software já desenvolvidos para diversas áreas de aplicação, desde que esses programas sejam devidamente adaptados no seu formato (tarefa relativamente simples) e recompilados pelos compiladores específicos do Ambiente multi-linguagem onde vão ser utilizados.

O estudo dos princípios para especificação de um Ambiente multi-linguagem teve como base linguagens algorítmico procedurais, tradicionalmente compiladas como C, Pascal,

Ada, Algol-68, Modula-2, Fortran, Algol 60. Não pretendemos aqui, especificar todos os detalhes de implementação de um sistema que contêm a máquina abstrata mencionada anteriormente, mas fixar os aspectos fundamentais que mostrarão a viaiblidade de tais sistemas, e servirão de especificação para projetos futuros.

Este trabalho está organizado de modo que no Capítulo I é feita uma descrição sucinta dos principais componentes que devem ser parte de qualquer implementação de um Ambiente multi-linguagem e são definidos os conceitos básicos para o entendimento dos critérios a serem especificados posteriormente, viabilizando a programação multi-linguagem. Neste capítulo é definida, também, a sintaxe de módulos escritos em diversas linguagens, com exemplos apropriados.

O Capítulo II descreve o comportamento dos objetos das interfaces. Nesse capítulo são discutidas as possíveis associações de objetos que fazem a comunicação entre módulos, a nível de LC, permitindo aos programadores de um módulo lançar mão das facilidades definidas ou declaradas em outro. São mostradas, também, as dificuldades encontradas para tornar viável, em tempo de execução, a passagem desses objetos, concretizando a junção dos módulos. Além disso, definimos representações (descritores dos objetos) para as quais todos os objetos das interfaces são transformados facilitando posteriormente, a verificação da compatibilidade entre os objetos associáveis.

O Capítulo III descreve as soluções encontradas para a passagem de variáveis e constantes entre módulos de uma configuração, com sugestões de implementação.

No Capítulo IV descrevemos as soluções encontradas para contornarmos as dificuldades na passagem de subprogramas com parâmetros e sugerimos uma implementação.

No Capítulo V é feita a descrição sintática e semântica da LC, com exemplos de configurações.

Finalmente no Capítulo VI são feitas algumas sugestões que servirão de apoio à implementação de Ambientes multi-linguagens tratando inclusive, de casos particulares relacionados com linguagens específicas.

CAPÍTULO I

NOÇÕES BÁSICAS

I.1. INTRODUÇÃO

Este capítulo pretende apresentar o contexto necessário ao desenvolvimento dos capítulos subseqüentes.

Embora no Capítulo VI sejam feitas propostas para uma implementação de um Ambiente multi-linguagem vamos aqui antecipar a idéia da sua organização geral, para que possamos no decorrer desse trabalho, colocar melhor os problemas provenientes de programação multi-linguagem e as soluções por nós sugeridas.

Apresentamos, também, alguns conceitos básicos, necessários ao entendimento global do trabalho, conceitos estes relacionados à definição de módulos e interfaces nesse Ambiente de Programação. Fazemos, ainda a descrição sintática de módulos escritos em algumas das linguagens possíveis de serem misturadas no Ambiente e apresentamos diversos exemplos.

I.2. DESCRIÇÃO SUCINTA DE UMA ORGANIZAÇÃO BÁSICA PARA UMA IMPLEMENTAÇÃO DE UM AMBIENTE MULTI-LINGUAGEM

O Ambiente tem por finalidade dar apoio à escrita de programas com mistura de linguagens. Trechos de programas, compondo unidades de programa, escritos e compilados sepa4adamente — os módulos — são guardados em uma bibiliteca, a Biblioteca de Módulos (BM). O usuário tem acesso a essa Biblioteca, para gravar os seus módulos e também, para consultar os que já existem prontos, reusando-os se for da sua conveniência.

Como mencionado anteriormente, a junção desses módulos, para formar um programa propriamente dito, é especificada pela Linguagem de Configuração (LC); fazemos referência a um programa assim construído como uma "configuração".

Na definição de uma configuração o usuário determina a associação dos objetos que fazem a comunicação entre os módulos (e como ela se processa), além de especificar ordens de execução e/ou ações de inicialização.

Os programas multi-linguagem são projetados para execução em uma Máquina Abstrata. Essa Máquina, através de seus componentes, toma as providências para que a execução desses módulos, escritos em diferentes linguagens, seja possível.

A figura I.1 estabelece graficamente as relações entre os componentes básicos em uma implementação de Ambiente multi-linguagem, dando idéia do seu funcionamento.

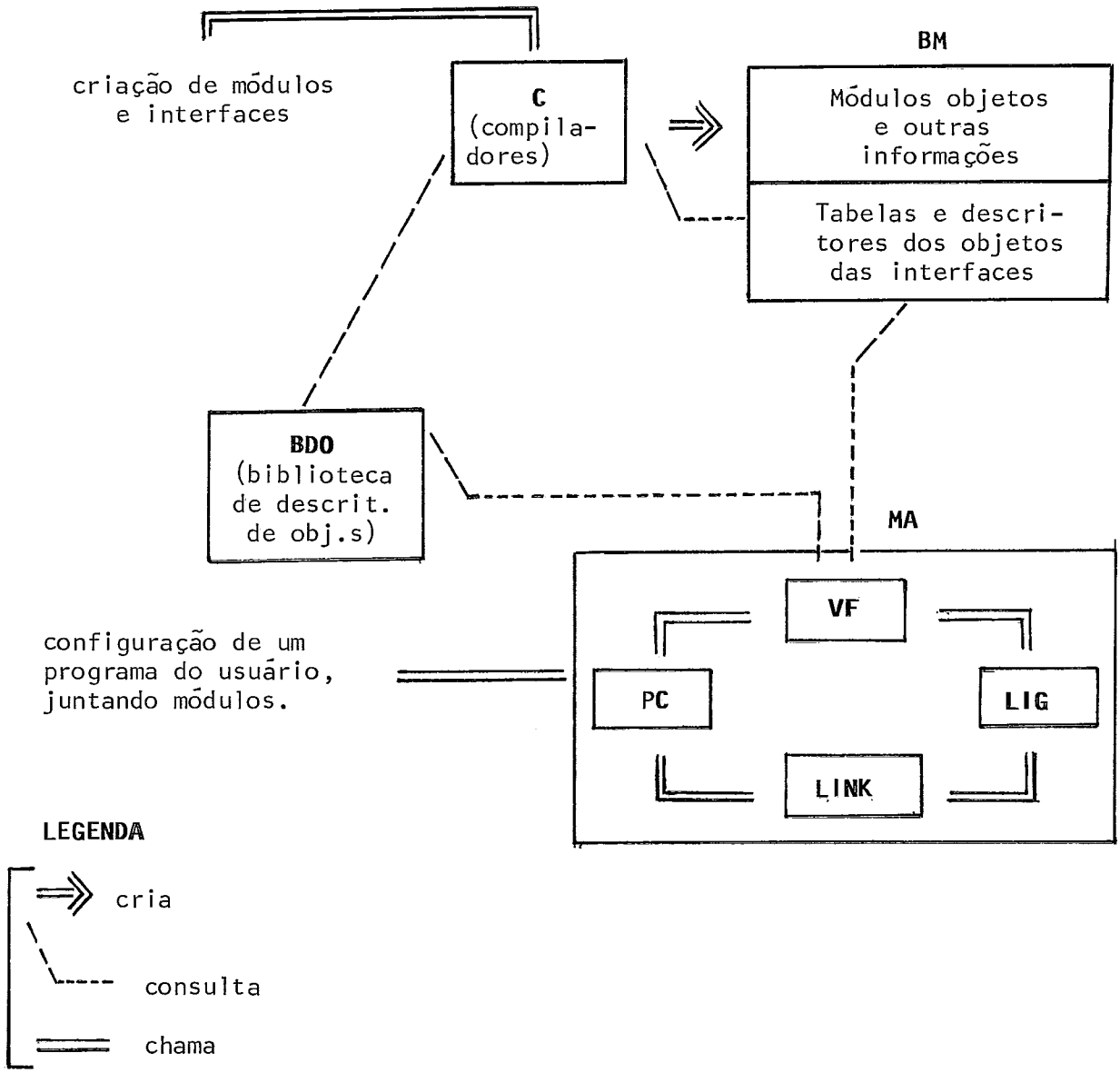


Figura I.1 - Representação gráfica do Ambiente

C - Compiladores das linguagens do Ambiente, especialmente construídos.

BM - Biblioteca de Módulos. A Biblioteca de módulos é um banco de dados, onde são guardados programas propriamente ditos, módulos escritos nas diversas linguagens mais todas as informações necessárias ao processamento de uma configuração.

Ao definir uma configuração, o usuário deve verificar se

os módulos de que necessita fazem parte da BM. Caso isto não ocorra, ele deve definir e compilar os módulos, bem como suas interfaces, guardando-os na BM.

BDO - Biblioteca de Descritores de Objetos. Conjunto de tabelas onde são estabelecidas as regras que permitem a transformação da descrição dos objetos de uma interface originalmente em uma das linguagens do Ambiente, para representações uniformes e genéricas, independentes de linguagens, denominadas descritores dos objetos, construídas a partir de descritores genéricos de tipos, uma vez que os objetos que passam em interfaces são tipados. Essas tabelas são utilizadas pelos compiladores das linguagens para gerar os descritores dos objetos declarados em interfaces, de forma a facilitar a verificação da compatibilidade entre os objetos que passam entre os módulos. Mais detalhes no Capítulo II.

MA - Máquina Abstrata. Formada pelos componentes PC, LINK, VF, LIG, onde:

PC - Processador de Comandos. Controla o processamento dos comandos de uma configuração. Aciona outros componentes da Máquina Abstrata, de acordo com o comando que está sendo analisado;

LINK - Editor de Ligação. Faz a montagem dos módulos de uma configuração;

VF - Verificador. O Verificador é acionado pelo Processador de Comandos (PC). Testa a viabilidade da passagem dos objetos entre os módulos verificando a compatibilidade entre esses objetos;;

LIG - Ligador. Formado por um conjunto de rotinas que faz par-

te do sistema de apoio à execução, cuja função é tomar as providências para que a execução do programa montado através de uma configuração seja possível.

I.3. CONCEITOS FUNDAMENTAIS

I.3.1. INTRODUÇÃO

Em algumas linguagens como FORTRAN e Pascal, não existe o conceito de unidade de encapsulamento (ou cápsula), portanto, não é possível agrupar um conjunto de unidades de programa inter-relacionadas e compilá-las em separado, para formar um módulo da BM. Por outro lado, linguagens como Ada e Módula-2 definem o conceito de cápsula, entretanto, é necessário que as regras por nós usadas para a passagem de objetos entre os trechos de códigos escritos em linguagens diferentes não interfiram de nenhuma maneira nas regras de importação e exportação de objetos entre cápsulas dessas linguagens. Além disso é interessante que possamos reunir um conjunto de cápsulas inter-relacionadas, escritas na mesma linguagem, formando uma cápsula maior.

Como em algumas linguagens não existem cápsulas e em outras as cápsulas não satisfazem inteiramente as nossas necessidades definimos então, no Ambiente, um mecanismo de encapsulamento, denominado módulo, constituído de uma interface onde estão as informações conhecidas do módulo e de um corpo, onde são escondidos os detalhes de implementação. Passamos, em seguida,

à descrição desse mecanismo.

I.3.2. CONCEITO DE MÓDULO

A especificação de um módulo, consiste na definição precisa de sua interface, isto é, na definição com o nível de detalhes apropriado dos objetos e operações que são enviados pelo módulo para uso dos demais e dos objetos e operações externos ao módulo e que são recebidos para seu uso. Cada módulo consiste, portanto, de duas partes: a especificação da sua interface e a implementação do módulo (corpo); essas duas partes podem ser compiladas em separado. Neste caso, a única comunicação entre o implementador do módulo e seu usuário consiste no texto de sua interface ficando outras informações, relativas à implementação, ocultas no corpo do módulo. Isso permite que alterações na implementação de um módulo, que faça parte de uma configuração, sejam invisíveis ("transparentes") para os demais módulos, se a especificação de sua interface continuar inalterada.

Nos Ambientes aqui discutidos, um módulo pode ser formado por um programa propriamente dito, auto-executável, ou por unidades de programa nomeadas tais como procedimentos, funções, pacotes de Ada, módulos de Modula-2, etc. Espera-se, ainda, que um módulo possa ser escrito, compilado e, posteriormente, gravado na Biblioteca de Módulos tornando-se então disponível para uso.

Em algumas linguagens, como Pascal, não é possível compilar um procedimento ou função, ou ainda um conjunto deles, separado do programa que vai usá-los. Dessa maneira, um subprograma ou um conjunto de subprogramas inter-relacionados, escritos

em uma dessas linguagens, não poderia formar um módulo da Biblioteca de Módulos. Com a definição desse mecanismo de encapsulamento, efetivamente, podemos formar uma unidade de compilação em separado encapsulando uma unidade de programa ou um conjunto delas inter-relacionadas, mais outras declarações (se existirem), escritas na mesma linguagem, dessa maneira formando um módulo que pode ser compilado em separado e fazer parte da Biblioteca de Módulos. Este módulo está pronto para se ligar a outros, via Linguagem de Configuração. O mecanismo de encapsulamento tem ainda a vantagem de esconder os objetos internos e protegê-los de acessos de forma não autorizada. Sua forma geral é a seguinte:

```
MDOULO <nome do módulo> : <nome da linguagem>
     <parte de declarações>
     <parte de comandos>
 FIM
```

O programador define o módulo em uma das linguagens do Ambiente (Pascal, C, Fortran, etc.) e usa o mecanismo previsto para encapsulá-lo. O cabeçalho (MODULO nome do módulo: linguagem que está escrito;) serve de diretiva para o compilador da linguagem em que foi escrito, para compilar em separado o seu conteúdo.

Tanto a parte de declarações, quanto a parte de comandos podem ser opcionalmente vazias, e apresentam outras variações de acordo com a sintaxe da linguagem do módulo.

A parte de declarações contém declarações de unidades de programa e de outros objetos existentes na linguagem fonte do módulo e permitidos na interface.

A parte de comandos (ou código de inicialização) contém os comandos necessários à inicialização do módulo, se assim determinar a lógica do programa do qual o módulo faz parte.

Como mencionamos antes, a sintaxe do código de inicialização varia de acordo com a linguagem em que o módulo está escrito. Por exemplo, no caso de FORTRAN, ou de ALGOL, o usuário não espera uma separação explícita entre a lista de declarações e a parte de comandos. Já no caso de Pascal e de outras linguagens a parte de declarações é separada da parte de comandos pela palavra BEGIN. (Podemos alternativamente dizer que a parte de comandos fica encapsulada num comando composto, entre as palavras BEGIN e END).

Cada uma das unidades de programa declaradas em um módulo qualquer, como também a sua parte de comandos é denominada de "componente" do referido módulo.

Por razões de uniformidade na criação de módulos nos Ambientes, mesmo as linguagens que possuem compilação em separado terão seus módulos envolvidos pelo referido mecanismo de encapsulamento.

I.3.3. UMA INTERFACE DE UM MÓDULO

I.3.3.1. INTRODUÇÃO

Cada interface faz a definição dos objetos que estabelecem a comunicação entre os módulos, por meio de listas de nomes de objetos recebidos e listas de nomes de objetos enviados. Nessas listas são definidos, respectivamente, os objetos recebidos

pelo módulo e os objetos por ele enviados para outros módulos.

Nesta seção vamos definir objetos enviados e recebidos, fazer uma descrição geral de interface e descrever como se processa o compartilhamento de informações entre módulos.

I.3.3.2. DEFINIÇÃO DE OBJETOS RECEBIDOS E ENVIADOS

Dizemos que um objeto tem origem em um módulo A, se ele for declarado em A e global aos seus componentes. Nesse caso, seu nome pode aparecer na interface correspondente ao módulo, na lista de objetos enviados de A para montagem com outros módulos, se essa for a intenção do seu projetista.

Um objeto que aparece na lista de objetos recebidos em uma interface de um módulo A, tem origem em outro módulo. A esse objeto é dado um nome local, através do qual ele pode ser referenciado em A. No entanto, a sua origem só vai ser determinada quando o módulo A fizer parte da configuração de algum programa. Nessa configuração é definida a junção dos módulos que formam o programa, entre os quais está o módulo A, e também, as associações entre objetos recebidos por um módulo e enviados por outro. De maneira que nenhum objeto recebido fique sem o correspondente objeto enviado associado, sob pena do programa não ser executado. Em outras palavras, um objeto recebido deve ser associado a apenas um objeto enviado por outro módulo. Entretanto, um objeto enviado por um módulo pode ser recebido por vários outros módulos na mesma configuração.

I.3.3.3. DESCRIÇÃO GERAL DE UMA INTERFACE

A interface é criada por um mecanismo da Linguagem de Configuração. Cada módulo pode possuir uma interface, onde são enviados e recebidos objetos diferentes.

Cada interface contém o cabeçalho do módulo a que pertence, as listas dos nomes dos objetos recebidos e/ou enviados e mais as declarações desses objetos, escritas na mesma linguagem do módulo, o que torna a criação das interfaces bastante simplificada para o usuário.

Objetos enviados e recebidos são objetos de natureza variada: constantes, variáveis, tipos, subprogramas e unidades de encapsulamento (*packages* de Ada, *modules* de Modula-2, etc.). Presume-se que esses objetos sejam usados no módulo correspondente à interface onde são definidos.

O esquema geral de uma interface é o seguinte:

```
INTERFACE <nome do módulo> : <nome da linguagem>

    RECEBE <lista de nomes de objetos recebidos> ;
    ENVIA  <lista de nomes de objetos enviados> ;
    <declarações dos objetos enviados e recebidos>
FIM
```

A declaração de objetos subprograma (recebido e enviados) consiste da declaração do cabeçalho do subprograma correspondente (que inclui a declaração dos parâmetros formais e tipo de resultado, se existirem); por sua vez, a declaração de obje-

tos unidades de encapsulamento consiste da declaração da sua parte visível (por exemplo a especificação de um pacote de Ada ou um módulo de definição de Modula-2).

Exemplo de uma interface e o módulo Ada correspondente:

```
INTERFACE M: Ada
     RECEBE h;
     ENVIA x, p;
     x:integer;
     package p is
          function f(i:integer) returns integer;
     end;
FIM

MODULO M: Ada
     declare
          x:integer;
          package p is
               function f(i:integer) returns integer;
          end;

          with n;
          -- na "package" n se encontra a
          -- function g(i:integer) returns integer;
          package body p is
             function f(i:integer) returns integer is
               return n.g(i)+h(i);
             end;
          end;
     begin
          null;
     end;
FIM
```

I.3.3.4. COMPILAÇÃO DE UMA INTERFACE

Cada interface é compilada em separado, antes do módulo

correspondente, pelo compilador da linguagem em que foi escrita.

Este compilador gera tabelas com informações sobre os objetos enviados e/ou recebidos pelo módulo. Constrói descritores dos objetos aí declarados. Essas informações são utilizadas durante a compilação do módulo propriamente dito e também durante o processamento de uma configuração onde o módulo ocorra.

Como os objetos recebidos são declarados somente na interface, o compilador precisa das informações sobre eles para testes de uso desses objetos no módulo correspondente.

No caso de objetos enviados, o compilador precisa saber quais são esses objetos não só para testes de usos desses objetos no módulo, como também para gerar código adequado à passagem desses objetos para outros módulos durante a execução de uma configuração.

A compilação em separado das interfaces torna possível a compilação de uma configuração, antes mesmo da definição de qualquer um de seus módulos. Isso permite verificar que as associações dos objetos enviados e recebidos estão de acordo com as suas especificações nas interfaces dos módulos. Naturalmente a ligação dos diversos módulos e a execução do programa definido pela configuração não serão possíveis até que todos os módulos estejam compilados e disponíveis na BM.

I.3.3.5. A COMUNICAÇÃO ENTRE OS MÓDULOS DE UM PROGRAMA

Como foi dito anteriormente, a comunicação entre os módulos em um programa multi-linguagem é feita através da

associação de objetos enviados aos objetos recebidos correspondentes, por meio da LC. Para o usuário do Ambiente, entretanto, a passagem de informações entre módulos (através dessa associação) se dá como transmissão de parâmetros de subprogramas. Isto é, a correspondência entre objetos enviados/recebidos por módulos que se juntam é semelhante à correspondência entre parâmetros reais/formais da chamada de um subprograma e sua respectiva definição.

Via de regra o nome de um objeto recebido, declarado na interface de um módulo, deve ser tratado pelo programador como local a esse módulo e visível por seus componentes, de maneira que siga as mesmas regras de visibilidade e escopo de nomes da linguagem em que esse módulo foi escrito.

Para que a comunicação entre os módulos se concretize em tempo de execução, várias providências devem ser tomadas nas diferentes fases do processamento de uma configuração. Essas providências começam a ser tomadas na fase de compilação de um módulo, antes mesmo dele fazer parte de qualquer configuração. O compilador do módulo, de posse das tabelas geradas durante a compilação da respectiva interface, tem informações sobre os objetos enviados e recebidos e, dessa maneira, desempenha uma série de ações (comuns ao conjunto de compiladores das linguagens misturáveis), relativas ao tratamento da passagem de objetos entre os módulos. Entre essas ações temos reservar área de memória para variáveis recebidas e enviadas e gerar código adequado às chamadas de subprogramas recebidos. Outras providências são tomadas pelos componentes da Máquina Abstrata que, acionados durante o processamento de uma configuração, desempenham as ações adequadas tornando possível a execução de um programa com

multi-linguagem.

I.4. ESPECIFICAÇÃO DE MÓDULOS EM DIVERSAS LINGUAGENS

I.4.1. INTRODUÇÃO

Nesta seção vamos definir com detalhes, a estrutura de um módulo escrito em algumas das possíveis linguagens do Ambiente, essa estrutura terá, naturalmente, forma relacionada com a sintaxe da linguagem correspondente.

Com a intenção de ressaltar as diferenças existentes entre módulos em linguagens diferentes, vamos ilustrar a definição de módulos FOTRAN, Algol 60, Pascal e C, reescrevendo o mesmo exemplo, onde é definido um módulo Pilha com operações de empilhar e desempilhar inteiros implementadas através de um *array*.

O exemplo usado para ilustrar escritos em Ada, define um tipo abstrato Pilhas que é declarado na interface como um objeto a ser enviado para outros módulos. Por outro lado, o exemplo para módulos do Ambiente em Modula-2 define um objeto *stack* que é recebido por esse módulo. Considerando que os módulos escritos em Ada e Modula-2 podem ser reunidos pela LC, eles se complementam formando um programa multi-linguagem.

I.4.2. MÓDULOS EM FORTRAN

A lista de declarações de um módulo FORTRAN pode conter subrotinas e funções; a sua parte de comandos não tem separação física da parte de declarações e corresponde ao programa prin-

cipal. Em Fortran subprograma terão acesso as variáveis da interface (recebidas e enviadas). Este acesso será indicado através de declarações COMMON, uma vez que este é o único meio existente na linguagem de simular o compartilhamento de variáveis entre unidades de programa.

Recomenda-se, como disciplina de programação, que este COMMON tenha um nome específico (tal como INTERFACE), e que seja mantido em todas as unidades exatamente da mesma forma, para evitar problemas que não seriam assinalados pelo compilador.

A chamada de um subprograma recebido por um módulo FORTRAN não necessita de nenhuma ação especial por parte do usuário (a não ser pelas já mencionadas anteriormente), para subprogramas recebidos de qualquer outra linguagem.

Exemplo:

```
C     *** TUDO NAS COLUNAS 7 a 72 ***
      I N T E R F A C E  PILHA: FORTRAN
          ENVIA INICIA, PUSH, POP, TOPO, AREA;
          SUBROUTINE INICIA
          SUBROUTINE PUSH(I)
          SUBROUTINE POP(I)
          INTEGER TOPO, AREA(100), I
      FIM
C     M O D U L O  PILHA: FORTRAN
          SUBROUTINE INICIA
          COMMON/INTERFACE/AREA, TOPO
          INTEGER AREA(100), TOPO
               .........
          END
```

```
C
      SUBROUTINE PUSH
      COMMON/INTERFACE/AREA, TOPO
      INTEGER AREA(100), TOPO
           .........
      END
C
      SUBROUTINE POP
      COMMON/INTERFACE/AREA, TOPO
      INTEGER AREA(100), TOPO
           .........
      END
C
C *** PROGRAMA PRINCIPAL OU PARTE DE INICIALIZAÇÃO DO
      MODULO ***
      COMMON/INTERFACE/AREA, TOPO
      INTEGER AREA(100), TOPO
           .........
      END
  F I M
```

I.4.3. MÓDULOS EM ALGOL 60

A estrutura de um módulo Algol é semelhante à de um programa na mesma linguagem com a seguinte forma sintática:

```
MODULO <nome> : Algol 60
     BEGIN
          < lista de declarações >
          < lista de comandos    >
     END
FIM
```

A lista de declarações do módulo corresponde à cláusula <lista de declarações> que pode conter declarações de variáveis, procedimentos e procedimentos com tipo (funções) (RUTISHAUSER , [10]). A parte de comandos do módulo Algol 60 pode conter nomes de objetos enviados e recebidos, tais como: variáveis e procedimentos (tipados ou não).

Exemplo:

```
INTERFACE pilha:  ALGOL60
     ENVIA inicializa, push, pop, topo, area;
     integer array area [1:100];
     integer topo;
     procedure inicializa; end;
     procedure push(i); integer i; end;
     procedure pop(i); integer i; end;
FIM
```

```
MODULO pilha: ALGOL60
     BEGIN
          comment lista de declarações ;
          integer array area[1:100];
          integer topo;
          procedure inicaliza;
          begin
               ......
          end inicaliza ;
          procedure push(i); integer i;
          begin
               ......
          end push ;
          procedure pop(i); integer i;
          begin
               ......
          end pop ;
          comment parte de comandos ou
               inicialização do módulo ;
               ......
     END
FIM
```

I.4.4. MÓDULOS EM PASCAL

Um módulo Pascal deve ter uma estrutura semelhante a de um programa Pascal; entretanto, o cabeçalho de programa não deve ser incluído. Sua lista de declarações pode conter quaisquer objetos declaráveis na linguagem, com exceção de rótulos; sua parte de comandos fica encapsulada como habitualmente, en-

tre os símbolos *begin* e *end*. Uma interface de módulo Pascal po de conter a descrição de objetos recebidos ou enviados; tais como tipos, variáveis, constantes, procedimentos e funções.

Exemplo:

```
INTERFACE pilha: PASCAL
     ENVIA inicializa, push, pop, topo-da-pilha,
           area-da-pilha;
     procedure inicaliza;
     procedure push(i:integer);
     procedure pop(var i:integer);
     var
          area-da-pilha: array[1..100] of integer;
          topo-da-pilha:integer;
FIM

MODULO pilha: PASCAL;
          var
               area-da-pilha: array[1..100] of integer;
               topo-da-pilha:integer;
          procedure inicializa;
               ......
          end; (inicializa)
          procedure push(i:integer);
               ......
          end; (push)
          procedure pop(var i:integer);
               ......
          end; (pop)
     begin
          (código de inicialização)
     end
FIM
```

I.4.5. MÓDULOS EM C

Um módulo em C deve ter uma estrutura semelhante à um programa C, e tem a seguinte forma:

```
MODULO nome: C

     [ <instruções de pre-processamento> ]
     [ <declaração de objetos globais> ]
     main ( )
     {
          /* corpo de função principal, com declarações
     }       de suas variáveis , seus comandos e funções. */
     [
     [ <tipo> ] fun ([<lista de parâmetros>])
     [ <declaração dos parâmetros>]
     {
          /* corpo da função func ( ) com suas declara-
             ções de variáveis, comandos e funções. */
     }
     ]
FIM
```

As cláusulas colocadas entre colchetes são opcionais. As declarações de objetos globais correspondem à declarações de tipos e variáveis.

O código de inicialização de um módulo em C consiste da função main() que pode conter somente o *null statement* (";") e deixar o módulo C, sem código de inicialização; por outro lado a sua parte de declarações consiste de:

```
instruções de pré-processamento;
declarações de objetos globais;
declarações de funções, tipadas ou não
     (excluída a função main() );
```

Uma interface de um módulo C pode conter a descrição de objetos recebidos e enviados tais como: tipos, variáveis e funções (tipadas ou não).

Exemplo:

```
INTERFACE PILHA: C
     ENVIA inicia, push, pop, topo, area[100];
     inicia /* é o cabeçalho da função inicia*/
     push ( i )
     int i;
     pop ( i )
     int i;
     int topo,
     area [100];
FIM
```

```
MODULO PILHA: C
     int topo,
     area [100];
     main()
     {
          /* aqui se define o código de inicialização do
             módulo Pilha*/
     }
     inicia
     {.....}
     push ( i )
     int i;
     {.....}
     pop ( i )
     int i;
     {.....}
FIM
```

I.4.6. MÓDULOS EM ADA

A sintaxe de um módulo Ada é definida a partir da sintaxe do comando bloco (*block*) da linguagem:

```
MODULO <nome> : Ada

     [ DECLARE
          parte-declarativa ]
     BEGIN
          seqüência-de-comandos
     [ EXCEPTION
          tratador-de-exceções ]
     END;
FIM
```

As cláusulas entre colchetes são opcionais, sendo que a seqüência de comandos é obrigatória, o que significa que se o módulo Ada não tem código de inicialização, a seqüência de comandos deve mesmo conter o comando nulo (*null-statement*, ou seja, *null* ;).

A parte declarativa, correspondente à parte de declarações do módulo, pode conter qualquer declaração permitida na linguagem incluindo corpos de pacotes, subprogramas, e outros objetos (ANSI [11]).

Na interface de um módulo Ada constam os nomes dos objetos a serem enviados ou recebidos, objetos tais como: constantes, variáveis, tipos, procedimentos, funções e pacotes.

Exemplo:

```
INTERFACE PILHA1:  Ada
     ENVIA pilhas;
     package pilhas is
          type pilha  is private;
          function push( p:pilha ;i:integer ) returns pilha;
          function pop(p:pilha) returns pilha;
          function empty(p:pilha) returns boolean;
          function top(p:pilha) returns integer;
          procedure init(p:out pilha);
     private
          type no is record
               val:integer;
               prox:pilha;
          end;
          type pilha is access no;
     end pilhas;
FIM
```

```
MODULO PILHA1:  Ada
     declare
     package pilhas is
          type pilha is private;
          function push( p:pilha ;i:integer ) returns pilha;
          function pop(p:pilha) returns pilha;
          function empty(p:pilha) returns boolean;
          function top(p:pilha) returns integer;
          procedure init(p:out pilha);
     private
          type no is record
               val:integer;
               prox:pilha;
          end;
          type pilha is access no;
     end pilhas ;
     package body pilhas is
          function push( p:pilha ;i:integer ) returns pilha is
          begin
          .....
          end push;
          function pop(p:pilha) returns pilha is
          begin
          .....
          end pop;
          function empty(p:pilha) returns boolean is
          begin
          .....
```

```
            end empty;
            function top(p:pilha) returns integer is
            begin
            .....
            end top;
            procedure init(p:out pilha) is
            begin
            .....
            end init;
      end pilhas;
      begin
            null;
      end;
FIM
```

I.4.7. MÓDULOS EM MODULA-2

A parte de declarações de um módulo Modula-2 deve conter *definition modules* e *implementation modules*. Sua parte de comandos (com a inicialização do módulo, se existir) deve ser encapsulada por um *program module* que deve conter uma lista de importação com os objetos declarados em outros componentes necessários a inicialização do módulo.

Na interface de um módulo Modula-2 podem ser declarados como objetos enviados ou recebidos constantes, variáveis, tipos, procedimentos e *definition modules*.

Objetos enviados: Em Modula-2 declarações de objetos tais como constantes, procedimentos, tipos e variáveis são encapsuladas por alguns dos *modules* definidos na linguagem. Esses

objetos para constarem em listas de objetos enviados em uma interface devem ser definidos no módulo correspondente, em algum dos seus *definition modules*. Recomenda-se, como disciplina de programação, que todos os objetos cujos nomes constarem da lista de objetos enviados da interface sejam encapsulados no mesmo *definition module*. Cada um dos outros componentes do módulo Modula-2 deve importar esse *definition module*, tornando então visíveis todos os objetos enviados dentro do módulo Modula-2.

Objetos recebidos: Objetos declarados como recebidos em uma interface para serem usados no módula em Modula-2, devem constar da lista de importação de cada um dos seus componentes, tornando-se desse modo, visíveis em todos eles.

Para adequar os objetos recebidos à regra da linguagem quer permite a referência desqualificada a objetos importados, desde que o seu módulo de origem seja citado na cláusula FROM de uma lista de importação. Essa lista para os objetos da interface deve ter a seguinte forma:

FROM INTERFACE IMPORT <lista de objetos recebidos>;

O nome de um *definition module* recebido não faz parte dessa lista, uma vez que é necessário o seu nome fazer parte também de listas de importação com cláusula FROM, para que os objetos por ele encapsulados possam ser usados desqualificados por cada um dos *implementation modules* ou *modules* do módulo Modula-2.

Exemplo:

```
INTERFACE USA-PILHA: Modula-2
     RECEBE stack;
     DEFINITION MODULE stack;
       EXPORT QUALIFIED pilha, push, pop, empty, top,
                init;
       PROCEDURE push (p:pilha; i:integer): pilha,
       PROCEDURE pop (p:pilha): pilha;
       PROCEDURE empty (p:pilha): boolean;
       PROCEDURE top (p:pilha): integer;
       PROCEDURE init (VAR p:pilha);
     END stack;
FIM
```

```
MODULO USA-PILHA: Modula-2
     MODULE usa-stack;
       FROM stack IMPORT pilha, push, pop, empty, top,
            init;
       CONST
         abrepar  = 1;
         fechapar = 2;
         abrech   = 3;
         fechach  = 4;
         fim      = 0;
       VAR s: ARRAY [1..100] OF integer;
           x: integer;
       PROCEDURE check; boolean;
         VAR i: integer;
             p: pilha;
             ok: boolean;
       BEGIN
         i:=0;
         init (p);
         ok:= true;
         REPEAT
           i:= i+1;
           x:= s[i];
           CASE x OF
            abrepar, abrech :
              push (x);
            fechapar, fechach :
              IF NOT empty(p) AND (top(p) = x-1) THEN
                pop(p);
              ELSE
                ok:= false;
              END;
            fim : ;
           END;
      UNTIL NOT ok OR (x = fim)
      RETURN ok;
   END;
FIM
```

A escolha de cada linguagem para inclusão em uma implementação exige, como vimos nos exemplos anteriores, a definição criteriosa da estrutura sintática de seus módulos, observando-se não só as regras sintáticas da linguagem, como também as definições de módulos e de objetos enviados e recebidos aqui tratados.

No capítulo seguinte, vamos tratar do comportamento dos objetos nas interfaces, descrevendo as possíveis associações entre eles. Descrevemos também suas representações, que posteriormente serão usadas na verificação da compatibilidade entre os objetos que passam entre os módulos de uma configuração.

CAPÍTULO II

COMPORTAMENTO DOS OBJETOS DAS INTERFACES

II.1. INTRODUÇÃO

Para construção de um programa completo, a partir de diversos módulos, escritos em linguagens diferentes, são utilizados os códigos fonte dos módulos, as suas interfaces respectivas, e uma configuração, que descreve a reunião dos módulos para formar um programa, e faz as associações entre os objetos passados nas interfaces.

Para que essa associação seja possível algumas condições devem ser observadas. Se raciocinarmos que o objeto recebido representa no módulo onde é declarado, o objeto enviado a ele associado, vamos concluir que esses objetos devem satisfazer condições que tornem possível essa associação.

Neste capítulo o nosso objetivo é deixar claro como pode ser feita a associação de objetos pelo programador, mostrando as restrições a essas associações impostas, na maioria das vezes, para contornar dificuldades na implementação da mistura de

linguagens.

Para verificação da compatibilidade dos objetos passados entre os módulos, é necessário verificar que sua declaração na interface do módulo, como objeto enviado, corresponde às declarações encontradas nas interfaces dos módulos que se juntam via LC, como objetos recebidos. A forma escolhida para este fim é a de utilizar uma representação uniforme dos objetos (descritores dos objetos) que seja independente das representações usadas nas linguagens individuais. Para cada linguagem são escolhidos os objetos que podem ser passados em interfaces; para todos estes são estabelecidas regras que permitem a transformação da descrição do objeto, feita originalmente em uma das linguagens misturáveis, para essa representação uniforme. Assim, uma variável de um tipo *array*, em Pascal, será representada em Pascal por seu tipo, e esse tipo de Pascal terá então sua representação convertida, quando da compilação da interface do módulo escrito em Pascal, para a representação genérica de *array*. A verificação de compatibilidade de pares de objeto enviados e recebidos, sempre será feita em termos dessa representação, que refletirá ainda o tipo em compatibilidade (em equivalência forte ou fraca) em função de detalhes de implementação desses objetos.

Os conceitos de equivalência forte e fraca entre objetos são apresentados aqui para que o programador possa avaliar as associações por ele previstas, ao montar o seu programa.

II.2. ASSOCIAÇÃO DE OBJETOS ENVIADOS E RECEBIDOS

II.2.1. INTRODUÇÃO

Como regra geral, um objeto recebido deve ser associado a um objeto enviado da mesma natureza. Associações entre objetos da mesma natureza tornam este processo de programação mais confiável e seguro. Os testes feitos com eles, por ocasião de compilação de uma configuração, garantem sua compatibilidade.

Vamos analisar, em seguida, associações das diferentes naturezas de objetos que se passam nas interfaces, colocando as restrições necessárias em cada caso.

II.2.2. ASSOCIAÇÃO DE VARIÁVEIS

Variáveis são definidas em todas as possíveis linguagens do Ambiente podendo, portanto, ser associadas por módulos escritos em quaisquer dessas linguagens.

Variáveis enviadas podem ser variáveis de tipos básicos ou não básicos. Seus nomes devem aparecer na lista de objetos enviados de uma interface, desde que sejam declarados globalmente (ou como às vezes se diz, em nível estático zero) aos componentes do módulo, como qualquer outro objeto.

As variáveis devem ser associadas de acordo com seus tipos, isto é, é necessário que elas tenham tipos equivalentes (ver seções II.3 e II.4).

Do ponto de vista de uma configuração uma variável rece-

bida faz parte do ambiente do módulo em cuja interface foi declarada. Ela é visível por todos os seus componentes, sendo por eles compartilhada. No caso de uma variável enviada, seu escopo se estende até o fim do seu módulo de origem, seu tempo de vida, entretanto, transcende as ativições desse módulo, sendo o mesmo da configuração onde esse módulo ocorra. Portanto, seu valor não se perde entre as ativações do seu módulo de origem ou dos módulos para onde é enviada, mesmo durante períodos em que é invisível ou inacessível, tendo portanto tratamento semelhante ao de variáveis estáticas (*own* em Algol 60 ou variáveis de FORTRAN).

Em tempo de execução a associação de variáveis enviadas e recebidas se concretiza pelo tratamento do valor da variável enviada associada à recebida correspondente, feito de forma dependente do seu modo de passagem. Através desses modos de passagem especificamos localmente como vai ser usada a variável enviada, pela recebida a ela associada por uma configuração.

Para definir esses modos foram escolhidos mecanismos semelhantes aos de passagem de parâmetros de subprogramas, por valor, por referência, por nome, etc. Assim, na interface de cada módulo é associado a cada nome de variável recebida, na lista de variáveis recebidas, o modo de passagem do valor da variável enviada correspondente, por exemplo: — RECEBE A REF, B VALUE; — neste caso é especificado que o valor da variável enviada, associada à variável recebida A, vai ser passado por referência e o da variável enviada associada a B, por valor. Escolhemos esses mecanismos, para facilitar o uso do Ambiente pelos programadores das diferentes linguagens que o compõem. Assim, o programador FORTRAN, por exemplo, pode optar por passa-

gem de variáveis por referência, para ele mais fácil de usar, por já estar habituado a uso semelhante em FORTRAN; o programador PASCAL pode optar por passagem de variáveis por valor ou referência, pelos mesmos motivos que o programador FORTRAN, etc. Assim sendo, o Ambiente tem seu uso facilitado para quem entende de uma linguagem só, que eventualmente está programando o seu módulo para ser juntado a outros, realizados por outras pessoas, como também atende a diferentes necessidades que ocorrem em programação.

II.2.3. ASSOCIAÇÃO DE CONSTANTES

Um módulo pode enviar uma constante para outro se ambos são escritos em linguagens que tem declaração de constantes, sendo, entretanto, observadas algumas restrições. Com efeito, em algumas linguagens, (Ada, por exemplo) uma constante é simplesmente uma variável cujo valor não pode ser alterado durante a execução do programa e não há problemas especiais nessa associação. Em outras linguagens, (Pascal, por exemplo) uma declaração de constante é a declaração de um valor a ser inserido diretamente no código, a cada uso da constante feito pelo programador; neste caso, a recepção de constantes não pode ser permitida, uma vez que seu valor deveria ser conhecido em tempo de compilação do módulo receptor, e não em tempo de ligação.

Considerando-se então, para passagem entre módulos as constantes que podem ser entendidas como casos particulares de variáveis, as mesmas regras para associação de variáveis devem ser aplicadas.

Entretanto, cabe uma exceção à regra geral de só se per-

mitir a associação de objetos da mesma natureza, visto que há linguagens que não possuem definição de constantes, como por exemplo FORTRAN. Neste caso, permitimos a associação de variáveis a constantes, desde que os dois objetos tenham tipos equivalentes e o objeto recebido seja recebido por valor (ver Capítulo III).

II.2.4. ASSOCIAÇÃO DE TIPOS

As linguagens que possuem mecanismos para definição de tipos pelo usuário podem enviar e receber tipos. Para simplificar a implementação e permitir o tratamento, em uma linguagem, de objetos de tipos definidos em outras linguagens, todos os tipos passados são tipos de acesso (apontadores).

Um módulo que envia um tipo, envia também as operações que o manipulam. Com isso, o módulo que recebe um tipo recebe-o como é recebido um tipo opaco de Modula-2, cuja representação é apenas parcialmente conhecida, isto é, um ponteiro para outro tipo que na verdade pode ser um tipo qualquer da linguagem do módulo que o envia.

Exemplo:

```
Interface MR:  Pascal
     Recebe Lista,Entra, Inic;
     Type Lista;
     Procedure Entra (I:integer, var Z:Lista);
     Procedure Inic (var U:Lista);
Fim
```

```
Interface ME:  Ada
     Envia T, Inicio, Entrada;
     Type T is Access R;
     Type R is record
          val:integer;
          prox:T;
     end;
     Procedure Entrada (I:in integer, X: in out T);
     Procedure Inicio (U1: in out T);
Fim
```

O módulo ME, aqui representado por sua interface, envia o tipo T e as operações Entrada e Início para o módulo MR, onde são recebidos com os nomes Lista, Entra e Inic, respectivamente.

II.2.5. ASSOCIAÇÃO DE SUBPROGRAMAS

Subprogramas são objetos possíveis de serem associados por módulos escritos em quaisquer das linguagens do Ambiente.

Um subprograma recebido por um módulo vai ser chamado por um dos componentes desse módulo. Essa chamada naturalmente, corresponde à chamada do subprograma enviado associado.

Um subprograma recebido e o seu enviado correspondente para serem associados devem possuir as seguintes características:

1 - o mesmo número de parâmetros, que serão implicitamente associados entre si;

2 - se o subprograma é do tipo função, o tipo do resultado do subprograma recebido e do enviado correspondente devem ser equivalentes (ver seção II.3).

Nesta associação existem, entretanto, algumas dificuldades. Uma delas diz respeito à associação dos objetos passados como parâmetros, que devem seguir as regras de associação de objetos de mesma natureza. Outra dificuldade está relacionada com o modo de passagem desses parâmetros, isto é, o subprograma recebido define o modo de passagem de seus parâmetros não necessariamente, de forma igual ao do subprograma enviado. Por exemplo: se o módulo que recebe o subprograma está escrito em FORTRAN, o modo de passagem de seus parâmetros é por referência; se o módulo que envia está escrito em Pascal, o modo de passagem se seus parâmetros pode ser por valor ou referência. Assim, um parâmetro recebido pode estar associado a um parâmetro enviado com modo de passagem diferente do seu. É necessário então, encontrar soluções para esses problemas, levando em consideração os modos de passagem de parâmetros de cada linguagem, em cada caso. Para isso uma rotina do sistema de apoio à execução simula o modo de passagem de parâmetros para o subprograma enviado no ponto de chamada e, no fim da execução, simula o modo de passagem do chamador (no módulo recebedor), para revolução dos resultados. Os detalhes se encontram no Capítulo IV.

II.2.6. ASSOCIAÇÃO DE UNIDADES DE ENCAPSULAMENTO

Unidades de encapsulamento são unidades de programa tais como *packages* de Ada, *modules* de Modula-2, *clusters* de Clu, cuja finalidade é reunir as declarações de diversos objetos da

linguagem, de alguma maneira logicamente associados, em um único objeto composto.

A associação de unidades de encapsulamento se dá pela associação dos seus nomes no comando apropriado em uma configuração.

Implicitamente é feita a associação de cada objeto definido na interface (parte visível) da unidade recebida com o correspondente objeto da unidade enviada. Desse modo então, a associação de objetos nas interfaces de duas unidades de encapsulamento que se juntam é feita seqüencialmente, na ordem em que ocorrem suas declarações. Esse é o procedimento do Verificador, ao testar a compatibilidade entre unidades de encapsulamento.

Recomenda-se, então, como disciplina de programação que o programador ao juntar unidades de encapsulamento, além de verificar a ordem dos objetos aí declarados, procure utilizar, sempre que possível, nomes semelhantes para os objetos a serem associados implicitamente.

Um exemplo dessa associação é dado no Apêndice B, entre unidades de encapsulamento das linguagens Ada e Modula-2.

II.3. EQUIVALÊNCIA FORTE E EQUIVALÊNCIA FRACA ENTRE OBJETOS

As duas formas de equivalência anteriormente mencionadas se baseiam nos princípios seguintes:

1 - Quando a equivalência entre um objeto R recebido e um correspondente objeto E enviado é forte, o módulo que recebe R pode tratá-lo como se estivesse definido na própria linguagem. Nenhum tratamento especial é necessário, e basta que o módulo que recebe R tenha de alguma forma acesso a E, usualmente o endereço de E. Este é o caso mais simples de todos, do ponto de vista da implementação, uma vez que não requer nenhum cuidado adicional.

2 - Quando a equivalência entre um objeto R recebido e um correspondente objeto E enviado é fraca, o módulo que recebe R precisa tratá-lo de forma especial, uma vez que podem ser necessárias conversões de valor, ou outras formas especiais de tratamento, para que se possa de alguma maneira simular o uso de E na linguagem de R.

O critério para se considerar dois objetos como fortemente equivalentes se baseia em:

1 - para o programador, os dois objetos podem representar abstrações semelhantes. Se isso não acontecer, não interessa a associação, uma vez que ela não seria utilizada;

2 - as formas de implementação dos dois objetos, em suas respectivas linguagens, são exatamente as mesmas em todos os casos.

Assim, por exemplo, dois tipos inteiros de duas linguagens que não admitem exatamente os mesmos intervalos de valores

não serão considerados fortemente equivalentes, apesar de representarem a mesma abstração: o conjunto dos inteiros. O mesmo aconteceria se os dois tipos inteiros representassem exatamente os mesmos intervalos de valores, mas na implementação houvesse alguma diferença (por exemplo uma inversão na ordem dos bytes). A equivalência fraca, pelo contrário, se baseia apenas na possibilidade de representação das mesmas abstrações. Assim, qualquer tipo inteiro, seja qual for o intervalo permitido de valores, é fracamente equivalente a qualquer outro, independente do intervalo de valores associado a este, ou das formas de suas implementações.

Por facilidade de definição, consideraremos que a equivalência forte é um caso particular da equivalência fraca, ou seja, sempre que valer a equivalência forte, consideraremos válida também a fraca.

A princípio, podemos afirmar, que dois objetos quaisquer, definidos em linguagens diferentes, mas representando abstrações semelhantes podem ser associados por uma configuração. Na prática, entretanto, torna-se necessário fazer algumas restrições quanto ao conjunto de objetos que poderão passar em interfaces, visto que há casos em que se torna excessivamente dispendiosa ou complicada a passagem do objeto para outros módulos. No Capítulo VI, sugerimos um conjunto de tipos equivalentes (limitando desse modo o conjunto de objetos) das linguagens que podem ser usados em uma implementação do Ambiente.

Para que a compatibilidade entre objetos associados possa ser verificada é necessário, entretanto, que eles possam ser representados de uma forma padronizada em uma representação abs

trata que, posteriormente, vai ser utilizada nos testes de compatibilidade dos objetos associados por uma configuração. Vamos tratar, em seguida, da definição dessa representação abstrata.

II.4. REPRESENTAÇÃO DOS OBJETOS DAS INTERFACES

II.4.1. INTRODUÇÃO

Definiremos, em seguida, representações uniformes e genéricas das diferentes naturezas de objetos que passam nas interfaces, a partir das quais vão ser construídos os descritores dos objetos aí declarados, que posteriormente, vão ser utilizados pelo Verificador.

Uma vez que objetos declarados em interfaces estão relacionados com tipos, direta ou indiretamente, (isto é, através de seus componentes) definimos, primeiramente, um conjunto de descritores genéricos de tipos, agrupando os tipos das linguagens do Ambiente, por meio de uma classificação pelas suas semelhanças conceituais. Cada um desse descritores é representado por meio de uma estrutura de dados em forma de árvore – uma árvore de sintaxe abstrata – de modo que na árvore são ressaltados os aspectos sintáticos com conteúdo semântico da espécie de tipo que representa.

Para descrever e representar abstratamente o conjunto de declarações de objetos de cada natureza, possível de ocorrer nas interfaces dos módulos, precisávamos encontrar um conceito em linguagens de programação, que designasse e representasse os elementos de cada um desses conjuntos pelas suas características semelhantes, suprimindo as suas diferenças. Escolhemos o

conceito de tipo abstrato como o que melhor traduz esse raciocínio. Vamos então definir um tipo abstrato para cada natureza de objetos que podem ocorrer em uma interface. Escolhemos como representação de cada um desses tipos abstratos, uma estrutura em forma de árvore de sintaxe abstrata — o descritor de objetos daquela natureza.

II.4.2. DESCRITORES GENÉRICOS DE TIPO

Nas linguagens incluídas em uma implementação do Ambiente existem tipos conceitualmente semelhantes que vão corresponder a objetos associáveis por uma configuração, como por exemplo: a estrutura de tipo acesso de Ada que tem semelhanças com a de tipo ponteiro de Pascal e de Modula-2. Tornou-se então conveniente encontrar uma representação padronizada para tipos semelhantes, — denominada descritor genérico do tipo — de modo que a partir dessa representação possamos construir a representação dos tipos abstratos das diferentes naturezas de objetos das interfaces. A representação de cada tipo é desenvolvida a partir de uma classificação dos tipos das linguagens presentes em uma implementação do Ambiente, pelas suas semelhanças conceituais, tendo como base a classificação de tipos de HOARE [12], que deve ser estendida de modo a englobar tipos de todas as linguagens escolhidas para essa implementação. Cada classe de tipos é então representada por meio de uma árvore de sintaxe abstrata onde ficam preservadas as características da classe, que independem da definição sintática do tipo que representa em qualquer das linguagens que o possua. Assim temos por exemplo que o descritor de tipo registro, modelado pela árvore de sintaxe abstrata de tipo registro, representa os tipos *struct* de C,

record de Ada, Pascal, Modula-2, etc. No Apêndice A, é dado um exemplo dessa classificação, para algumas linguagens.

II.4.3. O TIPO ABSTRATO DAS DIFERENTES NATUREZAS DE OBJETOS

II.4.3.1. INTRODUÇÃO

O tipo abstrato dos objetos de uma mesma natureza (daqui por diante denominado de tipo de objeto) guarda as características gerais daquela natureza de objetos nas linguagens do Ambiente. Sua representação em forma de árvore abstrata, construída a partir da representação dos descritores genéricos de tipos, fornece uma descrição das informações necessárias para testes de compatibilidade de objetos associados por uma configuração e para a eventual conversão de valores desse tipo em outros. Desse modo, uma declaração de um objeto em uma interface pode ser vista como uma instanciação do correspondente tipo abstrato. A partir de sua representação – o descritor da natureza do objeto – é construído o descritor do objeto declarado em uma interface como uma instanciação da representação do tipo abstrato correspondente.

II.4.3.2. O TIPO DE OBJETOS TIPO

Modela todas as declarações de tipo possíveis de existir nas interfaces dos módulos. Sua representação é construída a partir da representação dos descritores genéricos de tipos, tratados na seção anterior e tem a seguinte forma geral (fig. II.1).

Uma declaração de tipo em uma interface dá origem à construção de um descritor desse objeto, que é uma instanciação da árvore de descritor de objetos tipo.

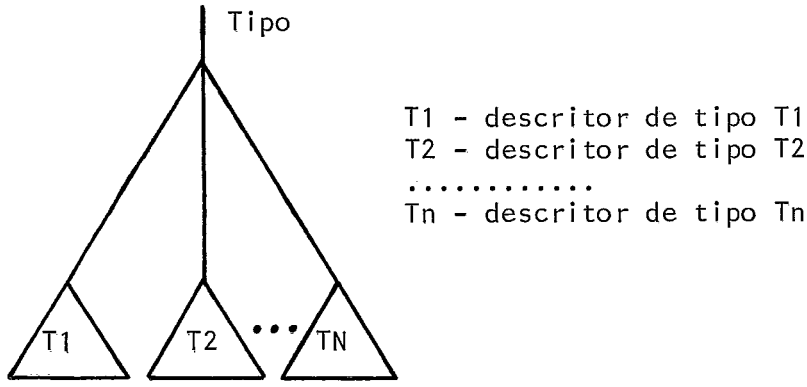


Figura II.1 - Descritor de objetos tipo

II.4.3.3. TIPO DE OBJETOS VARIÁVEL E CONSTANTE

Modela todas as declarações de variáveis e constantes que podem ocorrer nas interfaces de um módulo.

Como são objetos tipados, seus descritores são construídos a partir do descritor de tipo.

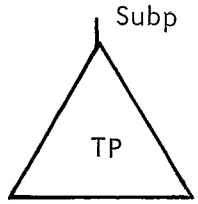
DT - descritor de tipo

Figura II.2 - Descritores de objetos variável e constante

II.4.3.4. O TIPO DE OBJETOS SUBPROGRAMA

Modela todas as formas de declaração de cabeçalhos de

subprogramas possíveis de passar nas interfaces (cabeçalhos de procedimentos e funções, com ou sem parâmetros).

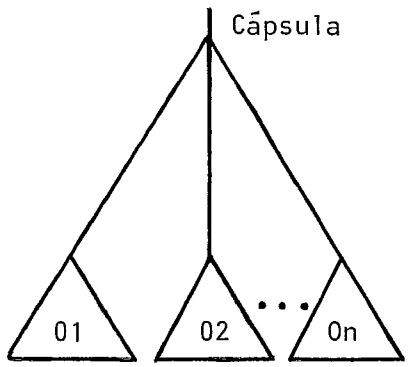


TP - descritor de tipos procedimento
ou
tipos função

Figura II.3 - Descritor de objetos subprograma

II.4.3.5. O TIPO DE OBJETOS UNIDADE DE ENCAPSULAMENTO

Modela as unidades de encapsulamento que podem passar nas interfaces. A estrutura genérica de unidade de encapsulamento é composta pela estrutura genérica dos tipos abstratos tratados anteriormente, de acordo com as declarações de objetos na parte visível da cápsula.



01 - descritor de objeto 01
02 - descritor de objeto 02
.....
On - descritor de objeto On

Figura II.4 - Descritor de objetos cápsula

II.4.4. A VERIFICAÇÃO DA COMPATIBILIDADE

Já vimos como os objetivos de segurança e confiabilidade entre objetos associados por uma configuração nos levam a exigir aprovação para eles num teste de compatibilidade. Se essa verificação termina com sucesso, a compilação da configuração continua. Entretanto, se um desses testes falha, a compilação da configuração é interrompida e o erro detectado.

A representação dos objetos que passam na interface é definida de modo que teremos descritores iguais para objetos fortemente equivalentes: num certo sentido objetos fortemente equivalentes podem ser considerados idênticos, e esse fato se reflete na igualdade de suas representações abstratas. Note que outras informações não relevantes para essa discussão poderão ser acrescentadas a essas representações, dependendo da implementação. As regras de equivalência fraca, entretanto, podem ser definidas em termos das representações dos objetos, na fase de projeto de um Ambiente multi-linguagem. Posteriormente, essas regras deverão ser traduzidas para os conceitos, termos e notações adequados a cada linguagem, de forma que se garanta a compreensão dos usuários.

Para o Verificador, um objeto enviado e seu correspondente recebido são compatíveis se, de um modo geral, entre eles são verificadas as seguintes condições:

1 - Os objetos são instanciações do mesmo tipo abstrato, conseqüentemente seus descritores devem ser construídos a partir da mesma representação de tipo, ou seja, o mesmo descritor de tipo de objeto.

2 - Verificada a condição anterior há dois casos a se considerar:

2.1 - os objetos podem conter os mesmos valores abstratos, entretanto suas implementações são diferentes. Isso se reflete nas suas representações abstratas, que são levemente diferentes. Neste caso, o Verificador detecta equivalência fraca entre os objetos. Então, ao se passar o fluxo de controle de um trecho de programa escrito em uma linguagem, para um trecho escrito em outra, torna-se necessário ajustar o objeto às restrições de implementação da outra linguagem, ou verificar que estas restrições são satisfeitas;

2.2 - os objetos contêm os mesmos valores abstratos e a implementação de um é exatamente igual à do outro. Aqui, desnecessário se torna qualquer ação no sentido de ajustar o objeto enviado à linguagem do módulo que o recebe. Os objetos possuem representações abstratas (descritores) idênticas. Neste caso, o Verificador detecta equivalência forte entre eles.

Na hipótese dos objetos não se enquadrarem nas condições anteriores, o Verificador detecta a incompatibilidade entre eles.

Em alguns casos, dependendo do conjunto de linguagens escolhido para uma implementação, podemos nos desviar dessas regras. Suponhamos, por exemplo, que um módulo Ada envie uma

constante; para que essa constante seja acessível em um módulo FORTRAN, será necessário aceitar uma correspondência entre a constante e uma variável FORTRAN de tipo compatível, recebida por valor.

No Capítulo VI, consideramos alguns casos particulares de associações de objetos de diferentes anturezas, com sugestões de implementação.

CAPÍTULO III

A PASSAGEM DE VARIÁVEIS E CONSTANTES ENTRE MÓDULOS E SUA IMPLEMENTAÇÃO

III.1. INTRODUÇÃO

Neste capítulo vamos tratar da passagem de objetos variáveis e constantes.

A passagem de variáveis entre os módulos nos levou a dificuldades de diversas ordens. Sendo uma configuração formada por módulos independentes, que podem até já estarem prontos na BM, as primeiras questões que nos surgiram diziam respeito a:

1 - Como localizar a variável enviada pelo módulo que a recebe?

2 - Em caso de equivalência fraca entre as variáveis enviada/recebida, como converter o valor da variável enviada para a linguagem do módulo que a recebe?

A primeira questão seria relativamente simples de resolver, caso as variáveis que passam entre os módulos fossem somen

te variáveis simples com equivalência forte. Ao se fazer, então, a ligação entre os módulos, o Editor de Ligações, que tem acesso aos endereços das variáveis enviadas, completaria o código dos módulos que as recebem. Entretanto uma variável composta também pode ser passada entre os módulos. Nesse caso, seu endereço pode mudar durante a execução de uma configuração e, portanto, o Editor de Ligações não terá, de antemão, acesso a esses endereços. Uma solução encontrada foi a de se definir um subprograma de acesso para cada variável enviada (semelhante a um *thunk* de passagem de parâmetro por nome), para calcular o seu endereço. Esse subprograma é gerado pelo compilador do módulo que envia a variável e colocado em endereço conhecido. Cada vez que no módulo recebedor houver necessidade do cálculo do endereço de uma variável enviada é feito uma chamada ao seu subprograma de acesso. Por razões de uniformidade do código gerado para um certo módulo e também porque antes de se definir uma configuração não se sabe quais as variáveis enviadas e recebidas que se correspondem, a implentação da passagem de variáveis para quaisquer dos dois casos de equivalência se faz por meio de chamadas aos seus subprogramas de acesso.

A questão que diz respeito a equivalência fraca entre as variáveis ocorre, na maioria dos casos, por razões de diferenças na implementação dos tipos dessas variáveis que se correspondem em uma configuração. Para superarmos este problema criamos uma rotina do sistema de suporte ao processamento chamada rotina de conversão (RC), com objetivo de contornar as diferenças existentes entre a implementação da variável enviada e sua correspondente variável recebida, segundo as restrições do tipo da linguagem que recebe a variável. Essa rotina é chamada, apro

priadamente, no código do módulo que recebe a variável.

A figura III.1 sintetiza os problemas apresentados anteriormente e suas soluções.

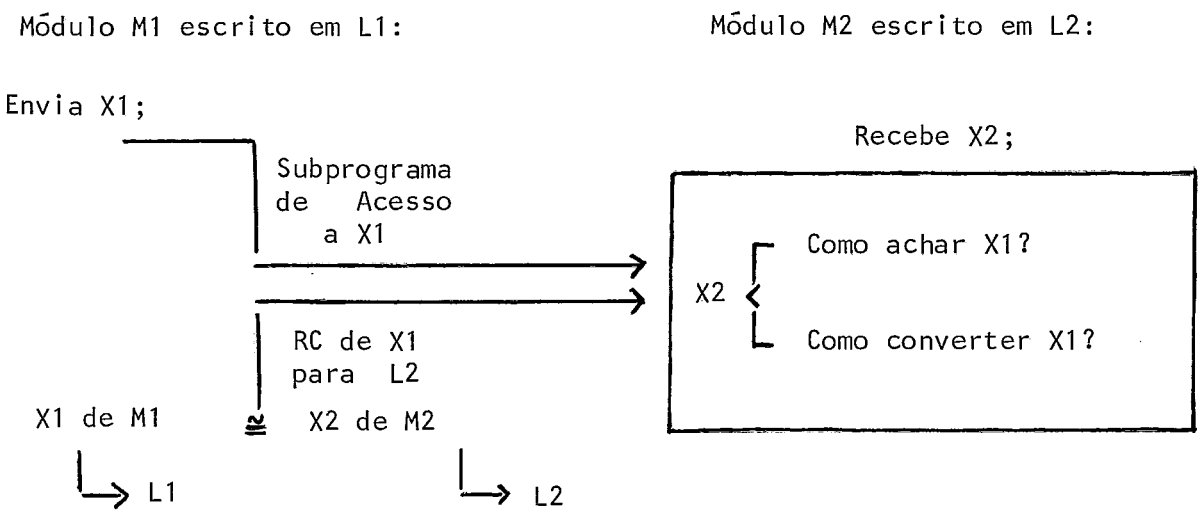


Figura III.1 - Problemas na passagem de variáveis

Para a passagem de variáveis sugerimos diversos modos, podendo o usuário escolher, de acordo com as necessidades do seu programa, se vai ser feita somente cópia da variável enviada (VE), para a variável recebida (VR) correspondente, no módulo que a recebe; ou se vai ser feita cópia e atualização do valor de VE; ou ainda se é o endereço da VE que vai ser colocado na área da VR correspondente, etc. Isto significa que o usuário tem a liberdade de especificar como é que deseja que sejam usadas as variáveis enviadas pelos diversos módulos que constituem a sua configuração.

As constantes que passam entre módulos são consideradas casos particulares de variáveis, por isso o tratamento dado à constante é o mesmo de variáveis, tanto pelo usuário (associando à elas os mesmos modos de passagem), quanto pelo projetista de um Ambiente multi-linguagem que considere os prin-

cípios discutidos nesse trabalho.

Os modos de passagem de variáveis e constantes são aqui denominados REF, NAME, VALUE, RESULT, VALUE-RESULT e IN-OUT. Neste capítulo, vamos descrever esses diversos modos, detalhando a implementação de cada um deles.

III.2. DESCRIÇÃO DOS MODOS DE PASSAGEM E SUA IMPLEMENTAÇÃO

III.2.1. INTRODUÇÃO

O modo de passagem de variáveis, escolhido pelo usuário, é associado a cada variável, na lista de variáveis recebidas da interface do módulo recebedor, de maneira que seu compilador gere o código adequado à essa escolha. Esse código inclui chamadas ao subprograma de acesso de cada uma, para o cálculo do endereço da variável enviada correspondente.

Para variáveis dos modos REF, VALUE, VALUE-RESULT, o cálculo do endereço (e todo tratamento inicial) é feito pelo código de inicialização do módulo onde são declaradas como recebidas. A execução desse código de inicialização se dá a cada entrada de um subprograma enviado por esse módulo, quando chamado em outro módulo que o recebe, ou ainda, na entrada do módulo onde começa a execução da configuração (mais detalhes em III.2.2).

Para variáveis recebidas do modo NAME o código de acesso a variável enviada correspondente é gerado a cada ocorrência da

variável recebida no módulo fonte.

Para o módulo que possui variáveis recebidas dos modos RESULT e VALUE-RESULT é gerado um código de finalização com o tratamento apropriado a essas variáveis. Esse código é executado na saída de um subprograma enviado por esse módulo, quando chamado de um outro módulo que o recebe, ou ainda, no fim da execução da parte de comandos de um módulo que pode ser entendida como um subprograma sem nome chamado pela configuração.

A implementação do mecanismo de passagem de variáveis requer, então, chamadas ao subprograma de acesso, no código gerado para o módulo recebedor para as variáveis recebidas em quaisquer dos modos de passagem disponíveis. Requer, também, a utilização dos códigos de inicialização e finalização, de acordo com o modo de passagem escolhido pelo usuário.

Vamos a seguir descrever esse subprograma de acesso, os códigos de inicialização e finalização, bem como os modos de passagem.

III.2.2. OS CÓDIGOS DE INICIALIZAÇÃO E FINALIZAÇÃO DE VARIÁVEIS E SUA IMPLEMENTAÇÃO

Como foi dito anteriormente, o código de inicialização de variáveis tem por finalidade o tratamento de variáveis modo REF, VALUE e VALUE-RESULT. A execução código se dá a cada entrada em um módulo com variáveis recebidas desses modos. Consideramos, para fins de implementação da passagem de variáveis, que um ponto de entrada em um módulo é caracterizado por uma chamada externa de um subprograma definido nesse módulo ou pela situação semelhante da execução da parte de comandos do módulo

entendida como um subprograma anônimo enviado, chamado diretamente pela configuração.

A razão pela qual excluimos o código de inicialização das chamadas de unidades de programa internas a um módulo, é o fato de que o usuário não tem conhecimento do que se passa internamente a esse módulo. Se especificou na sua interface um certo modo de passagem de variáveis, o programador espera que este modo prevaleça para a variável recebida, como existia no momento da chamada, até o retorno da chamada a uma das unidades enviadas (visíveis) do módulo. Natualmente, uma chamada externa de uma unidade desse módulo pode levar a outras chamadas externas de suas unidades, por ocasião das quais o código de inicialização das variáveis será executado. O programador deve ter conhecimento deste fato, e saber que a execução recursiva de módulos pode criar problemas sérios em virtude da não criação de novas instâncias das variáveis enviadas. Acreditamos, entretanto, que na maioria das aplicações, esta situação não cria problemas significativos.

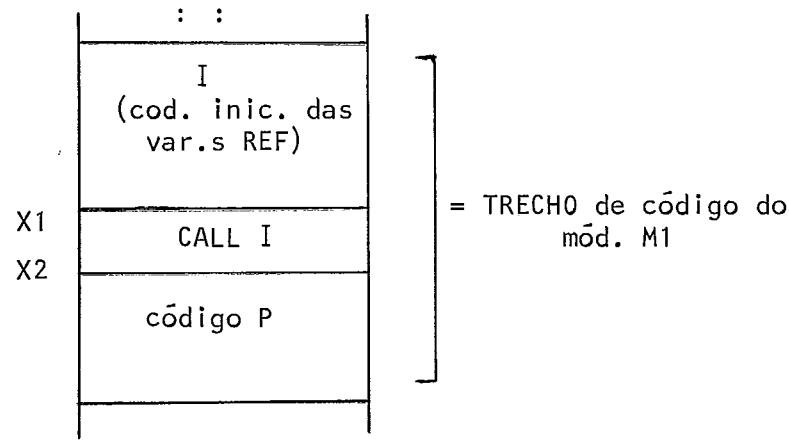
Para implementar o mecanismo descrito anteriormente, basta criar, para cada unidade PR enviada por um módulo, dois pontos de entrada, a serem usados, respectivamente, para chamadas externas e internas. O ponto de entrada para chamadas externas executa o código de inicialização para as variáveis recebidas, de modos REF, VALUE ou VALUE-RESULT e a seguir, transfere o controle para o segundo ponto de entrada, que é o ponto de entrada normal do módulo. O exemplo a seguir ilustra o problema.

```
INTERFACE M1:L1
  ENVIA P;
  RECEBE A REF;
  ...
FIM

MODULO M1:L1
  Procedure P;
   ...
  End;
(1) ...
  Procedure R;
    P;
    ...
  End;
  ...
FIM
```

```
INTERFACE M2:L2
  RECEBE P;
  ...
FIM

MODULO M2:L2
  ...
  Procedure M;
    P;
(2)   ...
  End;
  ...
FIM
```



Em (1) a chamada de P não implica inicialização das variáveis modo REF (que já foram inicializadas na entrada de M1). Portanto, o código de P é executado a partir de X2.

Em (2) a chamada de P (em outro módulo) implica na inicialização das variáveis REF recebidas pelo módulo de origem de P. Isso porque algumas delas (ou todas) podem ser usadas por P

e não teriam sido inicializadas. O código de P começa, então, a ser executado a partir de X1.

Observemos aqui que o cálculo do endereço da variável enviada deve ser feito no ambiente de execução do módulo que a envia e se esse módulo não se encontra presentemente em execução, o trecho de código que calcula o endereço da variável deve ser capaz de simular as condições necessárias.

O código de finalização é análogo ao visto anteriormente. Faz o tratamento adequado às variáveis enviadas definidas do modo RESULT ou VALUE-RESULT, na saída de um módulo. Entendemos, nesse caso, que a saída de um módulo é caracterizada pelo fim da execução de um subprograma por ele enviado, ou ainda, pelo fim da execução da parte de comandos do módulo.

A implementação do código de finalização é semelhante à do código de inicialização. Aqui são também definidos dois pontos de saída: um para chamadas internas e outro para chamadas externas, para cada unidade enviada por um módulo. O ponto de saída de chamadas externas executa o código de finalização e, a seguir, transfere o controle para o ponto de saída normal da unidade executada.

III.2.3. O SUBPROGRAMA DE ACESSO

Para cada variável enviada o compilador do seu módulo de oigem gera um subprograma de acesso ao endereço da variável (semelhantes a um *thunk* de passagem de parâmetros por nome). Esse subprograma é alocado junto ao código do módulo, em endereço conhecido. O seu uso torna eficiente a busca de componentes

de variáveis (tais como array, records, etc), por fazer isso da forma prevista pela linguagem da variável e por dispensar a conversão do valor da variável completa, limitando-se à conversão das componentes realmente utilizadas.

A chamada ao subprograma de acesso de cada variável enviada, só é possível no código dos módulos onde ela é recebida. Isso se dá por meio de código especial gerado pela ocorrência da variável recebida correspondente, de acordo com o modo de passagem escolhido pelo usuário e definido na interface do módulo que a recebe. É o Editor de Ligações que completa as chamadas ao subprograma de acesso de cada variável enviada, nos módulos que as recebem, utilizando-se dos endereços desses subprogramas, guardados em tabelas.

O subprograma de acesso é representado nos nossos algoritmos, como uma função com um único argumento (a variável enviada correspondente). Neste trabalho o resultado de uma chamada será indicado por ENDEREÇO (VE).

III.2.4. ROTINA DE CONVERSÃO DE VALOR

RC (DESC-VR, DESC-VE, L)

Esta rotina faz parte do Ligador e tem a finalidade de converter os valores de um objeto enviado à representação interna do tipo do objeto recebido associado, testando suas restrições, se existirem, e vice-versa.

A rotina copia o valor a ser convertido de um endereço conhecido (representado aqui por END) e o devolve convertido nesse mesmo endereço.

Os parâmetros DESC-VR, DESC-VE e L:

O compilador de um módulo qualquer que possua variáveis enviadas e/ou recebidas gera, para cada uma delas, um descritor contendo informações sobre seu tipo e sua linguagem. Estes descritores são guardados em endereços conhecidos.

A cada chamada da RC no código objeto de um módulo, são passados para ela, o endereço do descritor da variável recebida (DESC-VR) e o endereço do descritor da variável enviada correspondente (DESC-VE). Este último é completado pelo Editor de Ligações durante a associação dos objetos das interfaces dos módulos de uma configuração.

O parâmetro L se refere à linguagem para qual vai ser feita a conversão.

III.2.5. OS MODOS DE PASSAGEM

III.2.5.1. INTRODUÇÃO

Vamos, em seguida, descrever cada um dos modos de passagem de variáveis considerados para os Ambientes Multi-Linguagem. Tentamos descrever, de uma forma mais precisa, a semântica de cada um dos métodos por meio de algoritmos que podem ser considerados como uma sugestão de forma de implementação.

Os algoritmos são descritos em uma linguagem semelhante a Pascal. Usaremos, também, as seguintes abreviações:

VE - variável enviada;

LE - linguagem do módulo que envia a variável;

ME - módulo que envia a variável;

VR - variável recebida;

LR - linguagem do módulo que recebe a variável;

MR - módulo que recebe a variável;

END - endereço conhecido, usado pela rotina de Conversão (RC).

III.2.5.2. DESCRIÇÃO DO MODO REF

Neste caso, de um modo geral, VR deve conter o endereço de VE a partir do qual obteremos o valor de VE na representação usada por LE. A implementação da equivalência forte é simples e basicamente consiste no conjunto de ações do código de inicialização. Já na implementação da equivalência fraca, além do código de inicialização, a ser executado na entrada do módulo, é necessário gerar código a cada uso de VR, para converter o valor de VE para LR e também gerar código a cada definição de VR, atualizando o valor de VE. Vamos, a seguir, descrever as diferentes ações tomadas na implementação de variáveis de modo REF, para equivalências forte e fraca. Antes porém, vamos tratar do código de inicialização para essas variáveis.

Inicialização das variáveis modo REF:

A inicialização consiste do cálculo do endereço de VE a ser colocado na área de VR.

Se VE é uma variável simples, seu endereço permanece constante durante toda a execução e a área correspondente à VR pode ser preenchida em tempo de compilação da configuração, pelo Editor de Ligações. Por outro lado, se VE não é uma variável, simples, para

manter o paralelo com passagem de parâmetros por referência, seu valor deve ser calculado na "entrada de MR", isto é, o endereço de VE deve ser calculado a cada vez que é feita uma chamada externa de uma unidade de programa de MR.

Algoritmo para o código de inicialização:

O código de inicialização consiste apenas em colocar em VR o endereço de VE.

Algoritmo:

```
1 - VR := ENDERECO(VE)
    ** para cada VR é chamado o subprograma de acesso à VE
    ** associada, que calcula seu endereço e coloca-o em
       VR;
```

Implementação da equivalência forte:

É suficiente que VR contenha efetivamente o endereço de VE, porque o código de MR pode, em virtude das igualdades das representações das variáveis, tratar VE como se fosse uma de suas próprias variáveis.

Para variável simples é o próprio Editor de Ligações quem põe o endereço de VE em VR, durante a ligação dos módulos.

Para variável componente, no código de inicialização é feita uma chamada ao subprograma de acesso da variável recebida correspondente, que calcula o seu endereço, que é, então, colocado em VR. (Ver ítem anterior)

Implementação da equivalência fraca:

Para implementação da equivalência fraca, identificamos

três situações diferentes, para a geração de código adequado. Essas situações são descritas pelo seguinte algoritmo:

1 - ** na entrada do módulo: código de Inicialização;

2 - ** Antes de cada uso de VR, (ex. X:=.....VR....) o valor encontrado no endereço de VE deve ser convertido para a representação de LR e usado como sendo o valor de VR. Sugerimos uma implementação, onde o valor convertido é colocado em uma temporária e usado no lugar de VR.

```
END:= VR↑;
    ** põe em END, o valor a ser convertido;
RC(DESC-VR, DESC-VE, LR);
    ** converte VE para LR;
TEMP:= END;
    ** o valor convertido é colocado em uma temporária
       que é usada no lugar de VR;
```

3 - ** A cada definição de VR, (ex. VR:=.....) é necessário atualizar o valor de VE. Antes, porém, o valor alterado de VR é convertido para LE;

```
** TEMP contém o valor de VR a ser convertido;
END:= TEMP;
RC(DESC-VR, DESC-VE, LE);
    ** o valor de VR é convertido para LE;
VR↑:= END;
    ** o valor convertido é colocado em VE;
```

III.2.5.3. O MODO NAME

Descrição do Método

O mecanismo de passagem de parâmetros por nome baseia-se na idéia de que o mesmo nome pode representar distintos objetos durante a execução de um programa, e que portanto, durante a execução do subprograma chamado, devemos utilizar o objeto correspondente àquele nome, em cada ponto. Isso significa que o endereço e (se for o caso) o valor de cada parâmetro deve ser recalculado, a cada uso.

A situação é muito semelhante, no caso aqui estudado, à passagem de variáveis por nome. A cada uso da variável VR, devemos calcular o endereço de VE nesse ponto (usando o subprograma de acesso — ENDEREÇO (VE)). No caso da equivalência forte, obtido o endereço de VE naquele instante, executa-se com VE a ação especificada para VR; no caso da equivalência fraca, será necessário fazer além disso a conversão apropriada do valor.

O cálculo do endereço de VE obtido através do subprograma de acesso ENDEREÇO (VE) deve, naturalmente, ser feito no ambiente de execução de ME. Assim, o código para cálculo deste endereço deve ser gerado pela compilador de LE.

Vamos a seguir, descrever o algoritmo do método para equivalência forte e fraca.

Implementação da equivalência forte:

Na implementação da equivalência forte identificamos duas situações diferentes, exigindo código adequado, referentes respectivamente à definição e uso de VR.

Algoritmo:

```
** A cada definição de VR, o endereço de VE é calculado
   e o valor de VR à ele atribuído;
ENDEREÇO (VE) ↑ := VR;
** põe em VE, o valor de VR;
** A cada uso de VR, calcula-se o endereço de VE nesse
   instante e executa-se com o valor de VE a ação especi
   ficada para VR;
VR:= ENDEREÇO (VE) ↑ ; ** põe em VR o valor de VE;
```

Implementação da equivalência fraca:

A implementação da equivalência fraca é semelhante a implementação da equivalência forte. Aqui, entretanto, é necessário efetuar a correspondente conversão de valor, tanto para uso de VR, convertendo VE para LR, quanto para definição de VR, convertendo VR para LE.

Algoritmo:

```
** A cada uso de VR,
   calcula-se o endereço de VE nesse instante,
   convertendo-se o valor de VE aí encontrado,
   que é colocado em VR;
END:= ENDEREÇO (VE) ↑;
** põe em END o valor de VE a ser convertido para LR
RC (DESC-VR, DESC-VE, LR);
VR:= END;
** VR contém o valor convertido de VE
```

```
** A cada definição de VR,
   converte-se seu valor para LE,
   calcula-se o endereço de VE nesse instante,
   para guardar o valor calculado
END:= VR;
RC (DESC-VR, DESC-VE, LE);
ENDEREÇO (VE) ↑:= END;
** calcula endereço de VE
   e põe lá o valor de VR convertido para LE
```

III.2.5.4. O MODO VALUE

Descrição Geral:

Na passagem de variáveis por valor, VR deve ser entendida como uma variável local a MR, que na entrada de MR deve ter seu valor igualado ao de VE. Esse mecanismo é implementado pelo código de inicialização.

O código de inicialização:

Neste caso, o código de inicialização consiste em copiar o valor de VE para VR. Para isso é necessário, em primeiro lugar, calcular o endereço de VE e converter o valor aí encontrado para LR, se a correspondência VE/VR for em equivalência fraca, antes de copiá-lo de VE para VR.

Algoritmo:

```
** Para equivalência forte
VR:= ENDEREÇO(VE)↑;
** põe em VR o valor de VE
```

```
** Para equivalência fraca
END:= ENDEREÇO(VE)↑;
** põe em END o valor de VE a ser convertido para LR
RC(DESC-VR, DESC-VE, LR);
VR:= END;
** VR contém o valor convertido de VE
```

III.2.5.5. O MODO RESULT

Descrição Geral:

O modo Result, por analogia com passagem de parâmetros de subprogramas, caracteriza a VR como uma variável local à MR, cujo valor final é colocado em VE. Isso é implementado pelo código de finalização, a ser executado na saída do módulo (conforme descrito em III.2.2).

O código de finalização para o modo Result:

O código de finalização consiste em copiar o valor de VR em VE. Antes porém, é necessário calcular o endereço de VE e se a correspondência VE/VR for em equivalência fraca, converter o valor de VR para LE.

Algoritmo:

```
** Para equivalência forte
ENDEREÇO(VE)↑ := VR;
** põe em VE o valor de VR
```

```
** Para equivalência fraca
END:= VR;
RC(DESC-VR, DESC-VE, LE);
** converte o valor de VR para LE
ENDEREÇO(VE):= END;
** põe em VE o valor convertido
```

III.2.5.6. O MODO VALUE-RESULT

Descrição Geral:

O modo Value-Result é uma combinação do modo Value com o modo Result. Dessa maneira, a VR é entendida como uma variável local à MR, inicializada com o valor de VE, e cujo valor final é atribuído à VE atualizando-a, na saída do módulo.

A implementação do modo Value-Result

A implementação, neste caso, é feita com a combinação do código de inicialização do modo Value, a ser executado na entrada do módulo, com o código de finalização do modo Result, a ser executado na saída do módulo.

Algoritmo:

```
** para equivalência forte e fraca:
   ** a ser executada na entrada do módulo:
        algoritmo para equivalência forte ou fraca do
        modo Value, de acordo com a correspondência
        VE/VR; (ver III.2.5.4)
```

** a ser executado na saída do módulo:
algoritmo para equivalência forte ou fraca do modo Result, de acordo com a correspondência VE/VR; (ver III.2.5.5)

III.2.5.7. O MODO IN-OUT

Descrição Geral:

Na entrada do módulo é executado um código de inicialização semelhante ao modo Value (III.2.5.4), copiando o valor de VE em VR. Na saída é executado um código de finalização semelhante ao modo Result (III.2.5.5). Além disso, a cada saída externa, produzida através da chamada de um subprograma recebido pelo módulo, é feita atualização dos valores das variáveis enviadas e o retorno ao ponto de chamada produz a re-inicialização dessas variáveis.

Implementação do modo IN-OUT:

A implementação para equivalência forte ou fraca é semelhante. A diferença consiste na necessidade, em equivalência fraca, de se converter o valor de VE para LR, ao igualar o valor de VR ao de VE e vice-versa, quando houver necessidade de uma atualização do valor de VE, no seu módulo de origem.

Algoritmo:

1 - ** na entrada do módulo:
código de inicialização, conforme III.2.5.4

2 - ** antes de uma saída externa:
(chamada de um subprograma recebido),
atualiza cada VE correspondente,
no seu módulo de origem ;

** para equivalência forte:
ENDEREÇO (VE) ↑ : = VR;

** para equivalência fraca;
END := VR; ** põe VR em endereço conhecido
RC (DESC-VR, DESC-VE, LE);
ENDEREÇO (VE) ↑: = END; ** põe em VE o seu novo valor

3 - ** ao retornar de uma saída externa:

** re-inicialização das variáveis modo In-Out:
aplicação do código de inicialização do modo Value
(ver III.2.5.4);

4 - ** na saída do módulo:
código de finalização igual ao modo Result;
(ver III.2.5.5)

CAPÍTULO IV

A PASSAGEM DE OBJETOS SUBPROGRAMAS ENTRE OS MÓDULOS

IV.1. INTRODUÇÃO

A maneira mais simples de encarar a passagem de subprogramas entre módulos é considerá-los como sendo constantes e/ou variáveis de tipos procedimentos apropriados, mesmo que a linguagem em que foram programados, ou do módulo em que serão ativados, não possua semelhante conceito. Dessa maneira, as técnicas vistas no capítulo anterior aqui também se aplicam sem alteração. Podemos definir equivalência de tipos procedimento de diversas maneiras. Usaremos aqui as seguintes definições:

1. Diz-se que existe equivalência forte entre dois tipos procedimento quando:

   a. os números de parâmetros são os mesmos nos dois tipos;

   b. parâmetros correspondentes tem tipos fortemente equivalentes;

c. os modos de passagem de parâmetros correspondentes são os mesmos;

d. se um dos tipos descreve funções, o mesmo acontece com o outro, e os tipos dos resultados são fortemente equivalentes.

2. Diz-se que existe equivalência fraca entre dois tipos procecimento quando:

a. os números de parâmetros são os mesmos nos dois casos;

b. parâmetros correspondentes tem tipos equivalentes (equivalência fraca ou forte);

c. se um dos tipos descreve funções, o mesmo acontece com o outro, e os tipos dos resultados são equivalentes (equivalência fraca ou forte);

d. não existe equivalência forte entre os dois tipos;

e. as duas linguagens têm a mesma estrutura de execução.

Dessa maneira, podemos separar o caso da implementação em duas formas distintas, correspondentes às duas formas de equivalência definida.

No caso da equivalência forte, nenhuma alteração será necessária para o tratamento de uma chamada de um procedimento recebido de outro módulo, uma vez que pode ser tratado neste caso como se estivesse escrito na mesma linguagem do chamador. Notemos que a equivalência forte não é, estritamente, uma condição suficiente para a afirmação feita: seria necessário garantir adicionalmente que a estrutura dos registros de ativação (R.A.) de ambas as linguagens fosse exatamente a mesma, no que toca à

passagem de parâmetros, e aos elos estáticos e dinâmicos. Não há motivo nenhum, entretanto, para que isto não seja feito, desta maneira, uma vez que os compiladores das diversas linguagens usadas em uma implementação de um Ambiente multi-linguagem serão projetados especificamente para o Ambiente e poderão, portanto, satisfazer estas condições.

No caso da equivalência fraca, podemos tratar o assunto de duas maneiras distintas: na primeira, consideramos que o valor do subprograma recebido deve ser obtido, a partir do valor do subprograma enviado, através da conversão de um valor do tipo-procedimento enviado para o valor do tipo do procedimento recebido. Certamente possível, isto exigiria entretanto uma alteração significativa no código do procedimento enviado, a cada uma de suas chamadas.

A segunda maneira consiste em tratar de forma distinta a equivalência fraca entre variáveis, no caso de tipos procedimento. Em geral, não podemos neste caso admitir qualquer similaridade entre o registro de ativação do procedimento enviado, e do procedimento recebido. Precisamos, assim, admitir a necessidade de introdução de áreas onde a conversão de valores será feita; precisamos aceitar a necessidade de processamento adicional para o acoplamento de uma chamada de procedimento com parâmetros do chamador tratados de uma certa forma, quando os parâmetros do procedimento chamado são caracterizados por outra. Este é o problema que buscamos resolver neste capítulo.

A solução do problema fundamental é, entretanto, relativamente simples. Se observarmos as principais formas de passagem de parâmetros, (nome, referência, valor), encontraremos uma

hierarquia. Assim, no caso da passagem por nome, devemos passar um *thunk*, ou seja, a maneira de cálculo do endereço do parâmetro de chamada a cada momento da execução; a partir dessa informação, podemos obter o endereço e o valor desse parâmetro. No caso da passagem por referência, o endereço do parâmetro de chamada é fornecido; nesse endereço, o valor pode ser encontrado. No caso da passagem por valor, apenas o valor do parâmetro de chamada é fornecido. A existência dessa hierarquia faz com que possamos padronizar a geração de código para chamadas de procedimentos recebidos em equivalência fraca, através da geração de código como se o procedimento enviado sempre especificasse passagem por nome. Assim, um procedimento enviado que deseja tratar um de seus parâmetros por referência deve executar o *thunk* recebido uma única vez, no início de sua execução, e anotar o endereço assim obtido na área correspondente ao parâmetro. Um procedimento que deseja tratar um de seus parâmetros por valor executa o *thunk* correspondente e inicializa o parâmetro com o valor encontrado no endereço fornecido pelo *thunk*.

O outro problema adicional é o da conversão que pode ser necessária devido à equivalência fraca entre os modos de passagem parâmetros. Para resolver essa situação, criamos uma área intermediária entre os registros de ativação da unidade chamadora e da chamada a que denominaremos Registro de Ativação Fictício. Nessa área, a unidade chamadora anotará os endereços dos *thunks* correspondentes aos parâmetros de chamada, e faremos todo o tratamento necessário para que a unidade chamada funcione corretamente. A forma de implementar estes mecanismos é descrita em detalhes nas seções a seguir.

Como a linguagem com estrutura de execução estática mais provável de ser escolhida para uma implementação de um Ambiente multi-linguagem é FORTRAN, vamos tratar à parte, da solução do problema que envolve a mistura de um subprograma FORTRAN com outro escrito em linguagem com estrutura de execução dinâmica.

O tratamento de resultados de funções e o problema de subprogramas usados como parâmetro são discutidos em ítens específicos.

IV.2. DESCRIÇÃO GERAL DA PASSAGEM DE SUBPROGRAMAS ENTRE MÓDULOS

IV.2.1. INTRODUÇÃO

A passagem de um objeto subprograma, para um módulo que o recebe, se concretiza através de chamadas desse subprograma dentro do módulo recebedor, em um ou mais de seus componentes.

Uma chamada a um subprograma recebido produz a execução do subprograma enviado a ela associado, pela definição da configuração.

Várias providências devem ser tomadas em fases diferentes do processamento de uma configuração, para contornar as dificuldades que surgem para que a passagem de subprogramas entre os módulos se efetive em tempo de execução. Na realidade essas providências começam a ser tomadas, na fase de compilação de uma interface e, posteriormente, do módulo correspondente antes mesmo dele fazer parte de qualquer configuração. Ocorre, por exemplo, que o compilador do módulo, de posse das tabelas geradas

durante a compilação de sua interface, tem informações sobre os subprogramas enviados e/ou recebidos e dessa maneira, entre outras coisas, pode gerar código adequado à chamadas de subprogramas recebidos.

Algumas dificuldades já discutidas anteriormente, dizem respeito à passagem de subprogramas com parâmetros devido principalmente, às diferenças de modo na passagem de parâmetros entre subprogramas chamado e chamador.

Outras dificuldades dizem respeito à criação do ambiente de execução dos subprogramas chamado e chamador, bem como à manipulação dos registros de ativação nesse ambiente.

Esses problemas são contornados pela criação de rotinas do Sistema de Suporte à Execução (ou Ligador) que não só ajudam na criação do ambiente de execução dos subprogramas, como também manipulam os seus registros de ativação, ou ainda, tratam de adequar o modo de passagem de parâmetros de acordo com as necessidades.

Vamos tratar, neste capítulo, em detalhes, das soluções encontradas para contornar as dificuldades mencionadas anteriormente.

IV.2.2. OS PARÂMETROS DOS SUBPROGRAMAS ENVIADOS E RECEBIDOS

Entendemos que os parâmetros definem objetos que também vão passar entre os módulos, portanto o problema de passagem de parâmetros entre os módulos recai no problema de passagem de objetos entre módulos. Por isso são permitidos como parâmetros de subprogramas enviados e recebidos constantes, variáveis e

subprogramas (dependendo da linguagem que envia e da que recebe).

Parâmetros enviados e recebidos são declarados, juntamente com os subprogramas correspondentes, nas interfaces dos módulos onde ocorrem (ver definição de interface). Parâmetros recebidos se correspondem em tipo e modo de passagem, aos parâmetros da chamada do subprograma recebido. O compilador do módulo testa o tipo do parâmetro de chamada contra o tipo do recebido correspondente. Os parâmetros enviados correspondem aos parâmetros formais da definição do subprograma recebido (no módulo que o envia).

Durante a compilação da interface de cada módulo, para cada unidade de programa com parâmetros, enviada ou recebida, é criado um descritor de cada um desses parâmetros, contendo informações tais como: linguagens do módulo que envia (ou recebe) a unidade, tipo do parâmetro, modo da passagem do parâmetro. As informações sobre linguagem do módulo e tipo do parâmetro vão ser usadas pelo Verificador, para testes de compatibilidade de tipos entre o parâmetro recebido e o enviado correspondente, verificando entre eles equivalência forte ou fraca, cujo resultado é acrescentado a um desses descritores. Posteriormente, essas informações vão ser usadas para adequar o modo da passagem de parâmetros entre a unidade chamada e chamadora e vice-versa.

IV.2.3. O MÉTODO DE PASSAGEM

Na passagem de objetos subprogramas escritos em linguagens diferentes é necessário que certas ações sejam tomadas no

sentido de permitir a comunicação entre esses subprogramas. Algumas dessas ações dizem respeito ao tratamento do endereço do código do subprograma enviado. Esse problema, semelhante ao do cálculo do endereço da variável enviada, visto no capítulo anterior, tem a mesma solução, ou seja, o compilador do módulo que envia o subprograma gera um subprograma de acesso para o cálculo do endereço do seu código. Outras ações estão relacionadas com a criação do ambiente de execução dos subprogramas chamado e chamador (o conjunto de registros de ativação alcançáveis por esses subprogramas). Na construção desse ambiente algumas rotinas do Ligador desempenham um papel fundamental. Essas rotinas denominadas "rotinas para tratamento de objetos subprogramas" são em número de três: rotina Prólogo, rotina Epílogo e rotina Trata-Par. Vamos descrevê-las, brevemente, a seguir:

Rotina Trata-Par: - simula o modo de passagem de parâmetros do subprograma chamador ao subprograma chamado, no ponto de chamada. No fim da execução do subprograma chamado faz a passagem dos resultados ao chamador, simulando o modo de passagem de parâmetros do respectivo parâmetro de saída (ver ítem IV.2.5).

Rotina Prólogo: - auxilia na criação do ambiente de execução do subprograma recebido, no ponto de chamada. Faz chamadas à rotina Trata-Par. Desempenhas as ações iniciais, necessárias a passagem de um subprograma definido em uma linguagem e chamado por um módulo escrito em linguagem diferente. (detalhes em IV.2.4.2).

Rotina Epílogo: - desempenha as ações necessárias no final da execução do subprograma recebido. Trata os resultados desse subprograma, auxiliada pela rotina Trata-Par e restabele-

ce o ambiente de execução do subprograma chamador.

Além das rotinas descritas anteriormente, para que a passagem de subprogramas seja viável, outras providências são tomadas durante a compilação das interfaces e dos módulos que vão compor a configuração e que são:

1 - o compilador de cada módulo cria um descritor para cada subprograma enviado pelo módulo contendo informações tais como: linguagem do módulo, elo estático do subprograma e endereço do subprograma de acesso para o seu código. Esse descritor é alocado em endereço conhecido, através de uma convenção pré estabelecida para todo o sistema (em geral o topo da pilha de *hardware*) e seu endereço passado como parâmetro para a rotina Prólogo;

2 - o compilador de interfaces gera descritores dos parâmetros enviados e recebidos (conforme descrito em IV.2.1);

3 - o compilador do módulo que recebe um subprograma, ao encontrar uma chamada de tal subprograma, vai tratá-la de modo especial. Algumas das ações que vão ser desempenhadas na geração do código correspondente são as mesmas que seriam geradas na chamada de um subprograma escrito na mesma linguagem. Entretanto, toda chamada de um subprograma (com parâmetros) recebido por um módulo, em equivalência fraca, é tratada, nesse módulo, como se a passagem de parâmetros fosse por nome. Assim, o subprograma chamador gera, no ponto de chamada, um subprograma sem parâmetros (ou *thunk*),

para cada parâmetro de chamada e passa, para o R.A. que está sendo criado, uma lista de endereços desses *thunks*. A rotina Trata-Par, de posse dessa lista de endereços, vai adequá-la ao modo de passagem de parâmetros da unidade chamada.

No fim da execução do subprograma chamado é a rotina Epílogo que desempenha as ações adequadas, passando no final, o controle da execução ao chamador.

IV.2.4. O AMBIENTE DE EXECUÇÃO DA UNIDADE CHAMADA

IV.2.4.1. O REGISTRO DE ATIVAÇÃO FICTÍCIO

No ponto de chamada são colocadas informações no R.A., pela unidade chamadora, criando parte do ambiente de execução da unidade chamada. O código gerado neste ponto, pelo compilador do módulo, poderia por exemplo, ser o seguinte:

1. Avalia os parâmetros de chamada e faz o tratamento de passagem desses parâmetros, por nome.

2. Põe no R.A. do subprograma chamado, o endereço de retorno ao código do chamador.

3. Atualiza elo dinâmico.

4. Verifica se o subprograma chamado é recebido, caso seja, é feita uma chamada a rotina Prólogo (ver seção seguinte).

Os ítens 1, 2 e 3 descrevem algumas das informações no R.A. Fictício, (assim denominado por ser um R.A. auxiliar) que

faz parte do ambiente de execução das unidades chamada e chamadora em equivalência fraca.

O R.A. Fictício desempenha papel importante nas simulações realizadas pela rotina Trata-Par. De acordo com o modo de passagem de parâmetros no ponto de chamada e/ou na saída (no final da execução), o R.A. Fictício pode estar dividido em áreas diferentes. Uma dessas áreas vai conter sempre os endereços dos *thunks* dos parâmetros de chamada, aí colocados pela unidade chamadora. As informações nesse R.A. são completadas pela rotina Prólogo.

Em algumas ocasiões é necessário guardar o endereço dos parâmetros de chamada, depois de executados seus *thunks*. Nesse caso, a rotina Prólogo reserva uma área de tamanho adequado, (aqui denominada PFAS), no R.A. Fictício, antes de colocar informações no R.A. da unidade chamadora.

IV.2.4.2. A ROTINA PRÓLOGO

Se os subprogramas chamado e chamador são compatíveis em equivalência forte, a única ação desempenhada pela rotina Prólogo consiste em desviar o controle da execução, para o código do subprograma chamado.

No caso de equivalência fraca, auxiliando na criação do ambiente de execução da unidade chamada, a rotina Prólogo se utiliza do próprio R.A. Fictício e das informações nele contidas, como também das informações contidas no descritor da unidade chamada e daquelas contidas nos descritores dos parâmetros recebidos e enviados. De uma maneira geral, as ações desempe-

nhadas, neste caso, para criação do R.A. do subprograma chamado são as seguintes:

. Ajusta elo dinâmico que, no caso, aponta para o R.A. Fictício;

. Anota o endereço de retorno, que deve ser o endereço da rotina Epílogo;

. Faz chamadas à rotina Trata-Par para simulação da passagem de parâmetros, de acordo com o modo da linguagem do subprograma chamado;

. Desvia para o código do procedimento chamado, cujo endereço é obtido através da execução de seu subprograma de acesso.

A pilha de execução apresenta a disposição mostrada na figura IV.1, com relação aos registros de ativação, durante a execução de uma unidade recebida em equivalência fraca com a enviada correspondente.

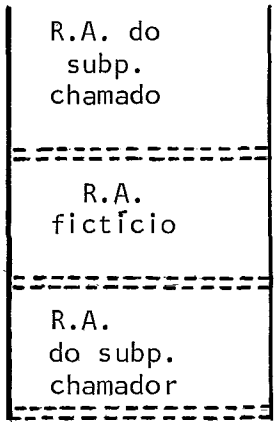


Figura IV.1 - Aspecto da pilha pela ocorrência de uma chamada a um subprograma recebido

IV.2.5. A ROTINA TRATA-PAR - O TRATAMENTO DA PASSAGEM DE PARÂMETROS

IV.2.5.1. INTRODUÇÃO

A rotina Trata-Par gera o código necessário para que a passagem de parâmetros seja realizada. Como os parâmetros podem ser vistos de formas distintas pela unidade chamadora e pela unidade chamada, é necessário um tratamento adicional na chamada e no retorno.

Na entrada do subprograma chamado, o tratamento da passagem de parâmetros é feito de acordo com o modo de cada parâmetro de entrada (definida no descritor do parâmetro enviado correspondente). O tratamento pode se dar, então, por passagem do valor do parâmetro para o R.A. do subprograma chamado (para modo definido como valor constante ou valor). Neste caso, a rotina Trata-Par, de posse dos endereços dos *thunks*, executa-os para obter o endereço do parâmetro de chamada correspondente e a partir desse endereço, obter o seu valor.

O tratamento por passagem de endereços para o R.A. do subprograma chamado (para modo definido por referência ou valor-constante), se dá pela execução do *thunk* pela rotina Trata-Par, que obtêm assim, o endereço procurado do parâmetro de chamada.

O tratamento por passagem dos endereços dos *thunks*, para o R.A. do subprograma chamado (no caso de modo definido por nome), ocorre pela cópia desses endereços no R.A. do subprograma chamado, feita pela rotina Trata-Par.

No fim da execução dos subprograma chamado a rotina Trata-Par devolve os resultados ao subprograma chamador. As ações aí desempenhadas vão depender não só do modo do parâmetro recebido associado a cada resultado, como também das ações que foram realizadas para passagem dos parâmetros, na entrada do subprograma chamado. De acordo com o modo do parâmetro recebido correspondente, cada resultado do subprograma chamado pode ser tratado por passagem de valor (para parâmetros recebidos definidos do modo resultado ou valor resultado), por passagem de endereços (para parâmetros recebidos definidos do modo referência), ou para parâmetros recebidos definidos do modo nome (neste caso, os resultados já se encontram nos endereços correspondentes). Vamos a seguir, analisar as combinações possíveis de passagem de parâmetros entre subprogramas chamado e chamador, descrevendo as simulações desempenhadas, em cada caso, pela rotina Trata-Par, de acordo com as informações sobre o modo de cada um deles, tanto na entrada do subprograma chamado, quanto na devolução dos resultados (se existirem) ao chamador.

Convém observar que, apesar dos parâmetros de chamada se encontrarem ou no R.A. do subprograma chamado, ou estarem alcançáveis a partir deste, nas figuras que ilustram neste texto, para simplificar, consideramos que esses parâmetros se encontram concentrados em uma área a eles destinada (denominada PC), no R.A. do subprograma chamador.

Uma descrição algorítmica vai ser feita, com o intuito de deixar mais claro cada simulação.

Notação:

PF - área de parâmetros no R.A. do subprograma chamado;

PC - área de parâmetros de chamada no R.A. do subprograma chamador;

PFF - área de parâmetros no R.A. Fictício que vai conter o endereço dos *thunks* dos parâmetros de chamada;

PFF_i - i-ésima posição da área de parâmetros no R.A. Fictício;

PF_i - i-ésima posição da área de parâmetros no R.A. do subprograma chamado (ou i-ésimo parâmetro formal);

PC_i - i-ésima posição da área de parâmetros do R.A. do subprograma chamador ou (i-ésimo parâmetro de chamada);

$DESC\text{-}PC_i$ - descritor do i-ésimo parâmetro de chamada (o mesmo descritor do parâmetro recebido correspondente). Contém informações sobre: linguagem do subprograma correspondente; tipo do parâmetro; modo da passagem de parâmetros;

$DESC\text{-}PF_i$ - descritor do i-ésimo formal (é o mesmo descritor do parâmetro enviado correspondente). Contém informações semelhantes às do descritor do parâmetro de chamada;

PFAS - área auxiliar criada no R.A. Fictício, para guardar os endereços dos parâmetros de chamada;

LD - elo dinâmico;

LR - linguagem do subprograma chamador;

LE - linguagem do subprograma chamado;

PR_i - i-ésimo parâmetro do subprograma recebido;

PE_i - i-ésimo parâmetro do subprograma enviado;

TOPO - variável que guarda o topo da pilha de execução.

Vamos, em seguida, ver como a rotina Trata-Par trata da passagem dos parâmetros na entrada e dos resultados (na saída) do subprograma chamado, simulando o modo adequado. O tratamento de subprogramas como parâmetros é estudado à parte, em IV.2.9.

IV.2.5.2. OS SUBPROGRAMAS CHAMADOR E CHAMADO TRATAM POR PASSAGEM DE VALOR

1 - Na entrada

Depois de executados os *thunks*, os valores dos parâmetros de chamada são acessados e copiados na área PF do R.A. do chamador, (figura IV.2).

Os endereços dos parâmetros de chamada vão ser salvos na área PFAS do R.A. Fictício. Isso é conveniente para evitar que o *thunk* de cada parâmetro seja re-calculado, quando no fim da execução, os resultados forem devolvidos ao chamador.

Algoritmo:

```
THUNKi;
     ** executa o thunk do i-ésimo parâmetro de chamada,
        para calcular seu end., devolvido en ENDERi
PFASi:= ENDERi;
     ** guarda end. na área PFAS do R.A. Fictício
IF PEi é parâmetro de entrada THEN
     ** copia seu valor no parâmetro formal correspondente.
        Caso PEi seja parâmetro de saída nenhum valor é passado para o PFi correspondente;
IF PRi e PEi em equivalência fraca THEN
     END:= PFASi↑;
          ** põe em END o valor do parâmetro de chamada
     RC(Desc-PCi, Desc-PFi, LE);
          ** converte PC para a linguagem do PF
     PFi:= END;
          ** o valor convertido é posto no R.A. do subprograma chamado
ELSE
     PFi:= PFASi↑;
          ** o valor do parâmetro de chamada é copiado direto em PFi;
```

2 - Na saída

Os parâmetros de saída, no R.A. do subprograma chamado, são convertidos para a linguagem do subprograma chamador, se necessário. Em seguida, esses valores convertidos são copiados nas posições correspondentes no R.A. do subprograma chamador ou em outro R.A. alcançável a partir deste.

Algoritmo:

```
IF PRi é parâmetro de saída THEN
     ** o valor do PFi é devolvido ao subprograma chama-
        dor
IF PRi e PEi em equivalência fraca THEN
     END:= PFi;
          ** põe em END o valor do parâmetro de saída
     RC(Desc-PCi, Desc-PFi, LR);
          ** converte PF para a linguagem do PC
     PFASi↑:= END;
          ** o valor convertido é devolvido ao  endereço
             guardado no R.A. Fictício
ELSE
     PFASi↑:= PFi;
          ** o resultado em PFi é copiado direto no ende
             reço apontado por PFASi
```

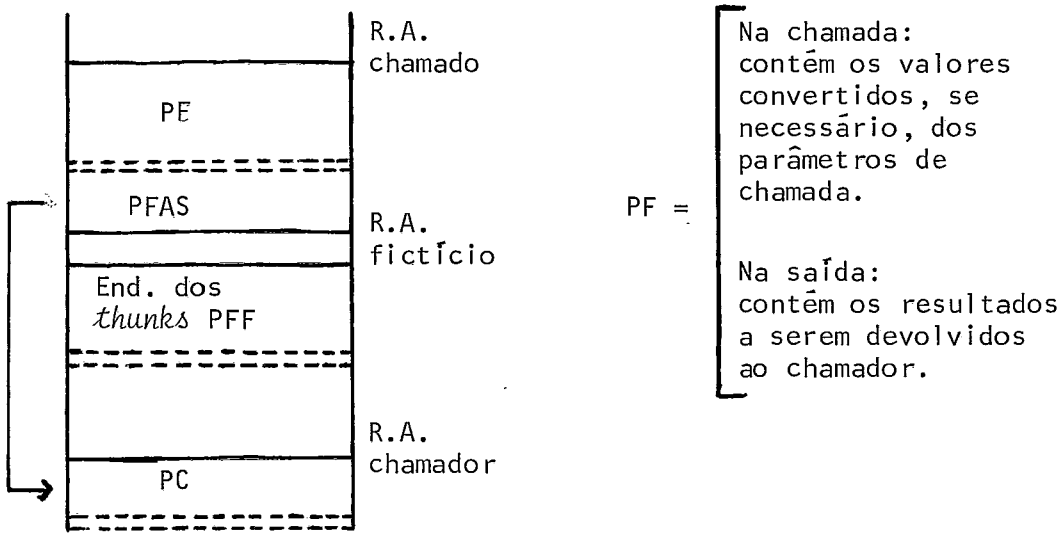


Figura IV.2 - Aspecto da pilha de execução quando subprogramas chamador e chamado tratam parâmetros por passagem de valor

IV.2.5.3. SUBPROGRAMA CHAMADOR E O SUBPROGRAMA CHAMADO TRATAM POR PASSAGEM DE ENDEREÇO

1 - Na entrada

A execução dos *thunks* nos dá os endereços dos parâmetros de chamada. Esses endereços são colocados nas posições, respectivas da área PF do R.A. do subprograma chamado. Se a correspondência entre PEi - PRi é em equivalência fraca, o valor do PCi é convertido para a linguagem do subprograma chamado (LE).

É importante salientar que se o subprograma chamado possuir parâmetros de chamada repetidos, a rotina Prólogo reserva uma só posição para guardar esse valor repetido, na área de parâmetros do R.A. do subprograma chamado.

Exemplo:

```
INTERFACE M1: L1          INTERFACE M2: L2
  RECEBE PP;                ENVIA P;
  PROCEDURE PP(I,J);        PROCEDURE P(L,K);
  ...                       ...
FIM                       FIM

MODULO M1: L1             MODULO M2: L2
  ...                       ...
  PROCEDURE PROC1;          PROCEDURE P(L,K);
  ...                       ...
  PP(K,K)                   END
  ...                       ...
  END                     FIM
FIM
```

M1 e M2 são escritos em linguagens diferentes e M2 envia procedure P(L,K) para M2 que o recebe com o nome PP(I,J).

O procedimento Proc1 chama PP e supomos que é por PROC1

que começa a execução. O R.A. tem o aspecto mostrado na figura IV.3 no momento da execução da chamada PP(K,K).

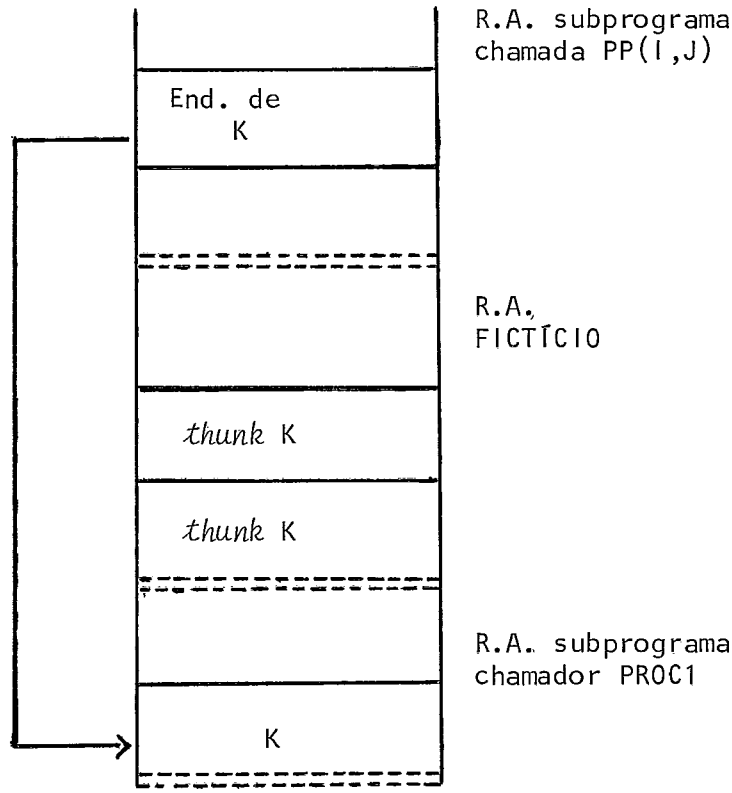


Figura IV.3 - Aspecto do R.A. no momento da execução da chamada PP(K,K)

Algoritmo: (ver figura IV.4)

```
THUNKi:
     ** calcula o end. do i-ésimo parâmetro de  chamada
        e o devolve em ENDERi
PFi:=ENDERi;
IF PRi e PEi em equiv. fraca THEN
END:= PFi↑;
RC(Desc-Pci, Desc-PFi, LE);
PFi↑:=END;
```

2 - Na saída

Os resultados devem ser convertidos, se necessário, para a linguagem do subprograma chamador - LR.

Algoritmo:

```
IF PRi e PEi em equiv. fraca THEN
   END:= PFi↑;
   RC(Desc-PCi, Desc-PFi, LR);
   PFi↑:=END;
```

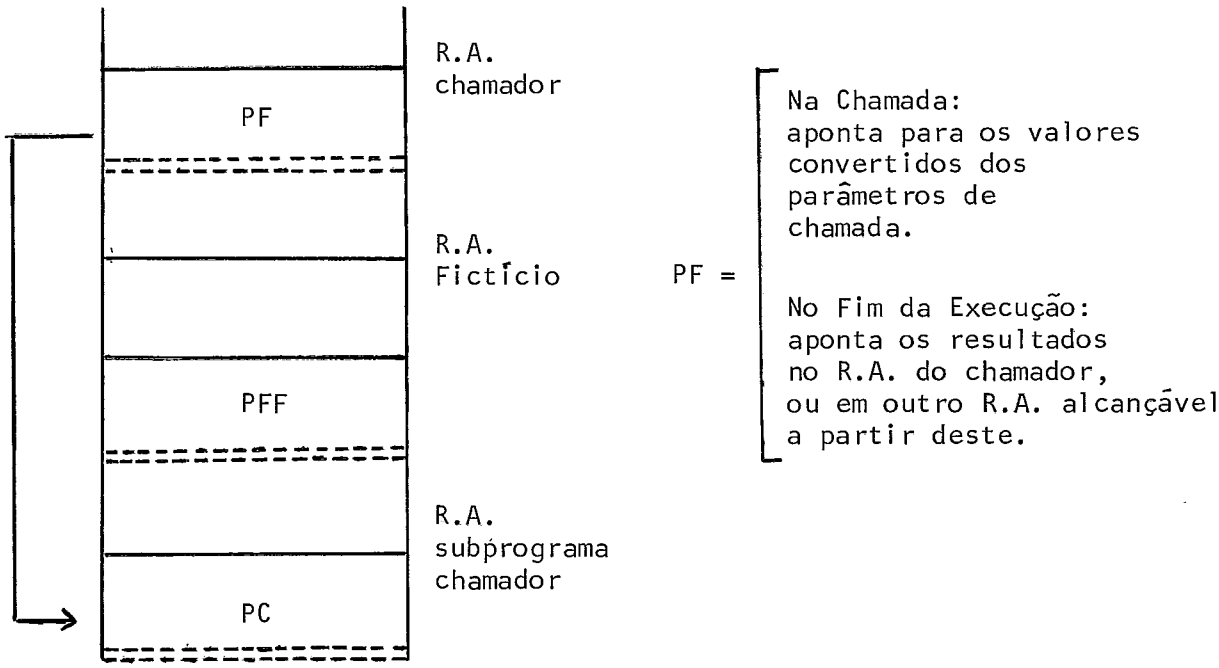


Figura IV.4 - Aspecto da pilha de execução quando subprogramas chamador e chamado tratam parâmetros por referência

IV.2.5.4. O SUBPROGRAMA CHAMADOR TRATA POR PASSAGEM DE ENDEREÇOS E O CHAMADO POR PASSAGEM DE VALOR

1 - Na entrada

Ao serem executados os *thunks*, e calculados os endereços,

são acessados os valores dos parâmetros de entrada e colocados na área de parâmetros do R.A. do subprograma chamado. (Figura IV.5).

Os endereços dos parâmetros de chamada são guardados no R.A. Fictício, evitando-se recalculá-los na saída.

Se houver dois ou mais parâmetros de chamada repetidos, a rotina Prólogo ao criar o R.A. do subprograma chamado reserva uma só posição para tais parâmetros na área PF desse R.A. Isto porque, como os parâmetros são iguais, só existe um endereço associado a tal parâmetro de chamada, para onde vai ser devolvido o respectivo resultado no fim da execução do subprograma chamado.

Algoritmo:

```
THUNKi;
     ** calcula o endereço do i-ésimo parâmetro de chamada e o devolve em ENDERi
PFASi:= ENDERi;
IF PEi é um parâmetro de entrada THEN
     ** copia o valor do parâmetro de chamada no respectivo PFi. Caso PEi seja um parâmetro de saída, o correspondente PEi não recebe nenhum valor.
IF PRi e PEi  em equiv. fraca THEN
     END:= PFASi↑;
     RC (Desc-PCi, Desc-PFi, LE);
     PFi:= END;
          ** põe o valor convertido na posição adequada da área de parâmetros do R.A. do subprograma chamado
ELSE
     PFi:= PFAS↑;
          ** valor do parâmetro de entrada é colocado na área de parâmetros do R.A. do  subprograma chamado
```

2 - Na saída

Como o subprograma chamador trata por referência, os resultados (neste caso, todos os parâmetros) são convertidos, se necessário, para a linguagem do chamador e colocados nos endereços salvos na área PFAS do R.A. Fictício.

Algoritmo:

```
IF PRi e PEi em equiv. fraca THEN
     END:= PFi;
     RC (Desc-PCi, Desc-PFi, LR);
     PFASi↑:= END;
          ** põe resultado convertido no seu endereço
ELSE
     PFAS↑:= PFi;
         ** põe o i-ésimo resultado no seu endereço
```

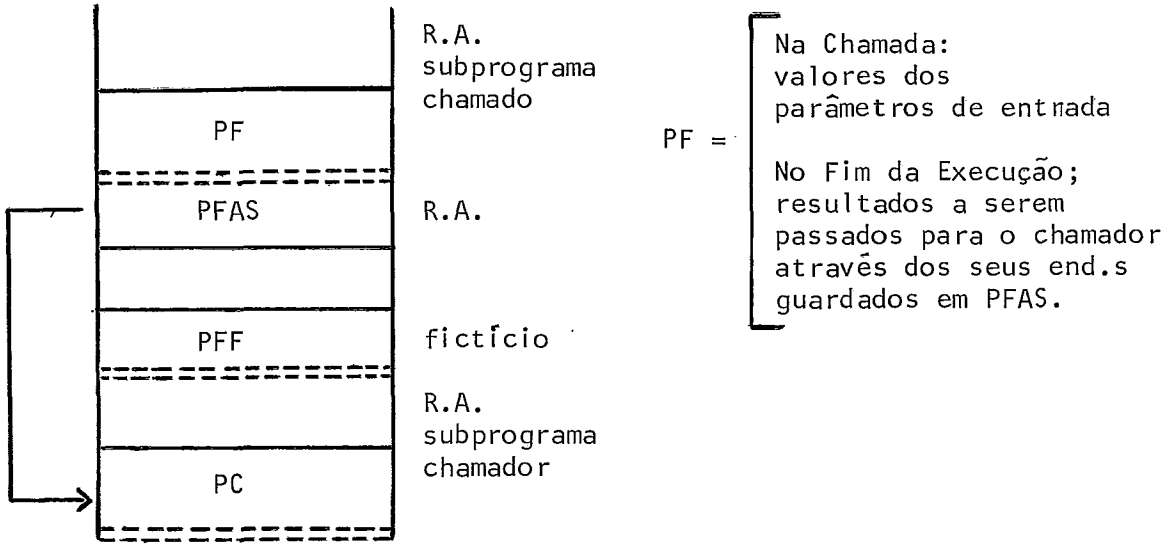


Figura IV.5 - Aspecto da pilha de execução, quando subprograma chamador trata parâmetros por passagem de endereços e o chamado por passagem de valor

IV.2.5.5. SUBPROGRAMA CHAMADOR TRATA POR PASSAGEM DE VALOR E O CHAMADO TRATA POR PASSAGEM DE ENDEREÇO

Ao serem executados os *thunks*, os endereços calculados são colocados na área PF do R.A. subprograma chamado. Os valores por eles apontados, devem ser convertidos para LE, se necessário.

A implementação feita pela rotina Trata-Par, neste caso, leva em consideração o modo dos parâmetros de chamada (definido no descritor do respectivo parâmetro recebido), verificando os parâmetros definidos como de saída (modo resultado ou valor-resultado).

Implementação: O parâmetro formal do chamador vai apontar direto para os parâmetros de chamada do modo resultado ou valor-resultado. Para os parâmetros de chamada do modo valor, os respectivos PF's apontam para o R.A. Fictício (área PFAS). (Ver figura IV.6).

1 - Na entrada

Algoritmo:

```
THUNKi;
IF PEi é do modo resultado ou valor-resultado THEN
     PFi:= ENDERi;
          ** endereço do PCi é colocado no respectivo PF
          ** Se PEi e PRi estão em equivalência  forte,
             nada mais é necessário fazer
IF PEi e PRi em equiv. fraca THEN
     END:= PFi↑;
     RC (Desc-PCi, Desc-PFi, LE);
     PFi↑:= END;
ELSE
          ** PEi é do modo valor
     PFAS:= ENDERi↑;
          ** copia o valor do parâmetro de chamada  na
             área PFAS
IF PEi e PFi em equivalência fraca THEN
     ** converte o valor de PCi para a ling. do chamador
     END:= PFASi;
RC(Desc-PCi, Desc-PFi, LE)
PFASi:= END;
PFi:= TOPO - i;
     ** PFi aponta para o respectivo parâmetro de chama-
        da, cujo valor se encontra em PFASi
```

2 - Na saída

Os resultados só serão convertidos para LR, se a correspondência entre PEi e PRi for em equivalência fraca; nada mais deve ser feito.

Algoritmo:

```
IF PEi e PRi em equiv. fraca THEN
     ** converte os resultados para LR
     END:= PFi↑;
     RC (Desc-PCi, Desc-PFi, LR);
     PFi↑:= END;
```

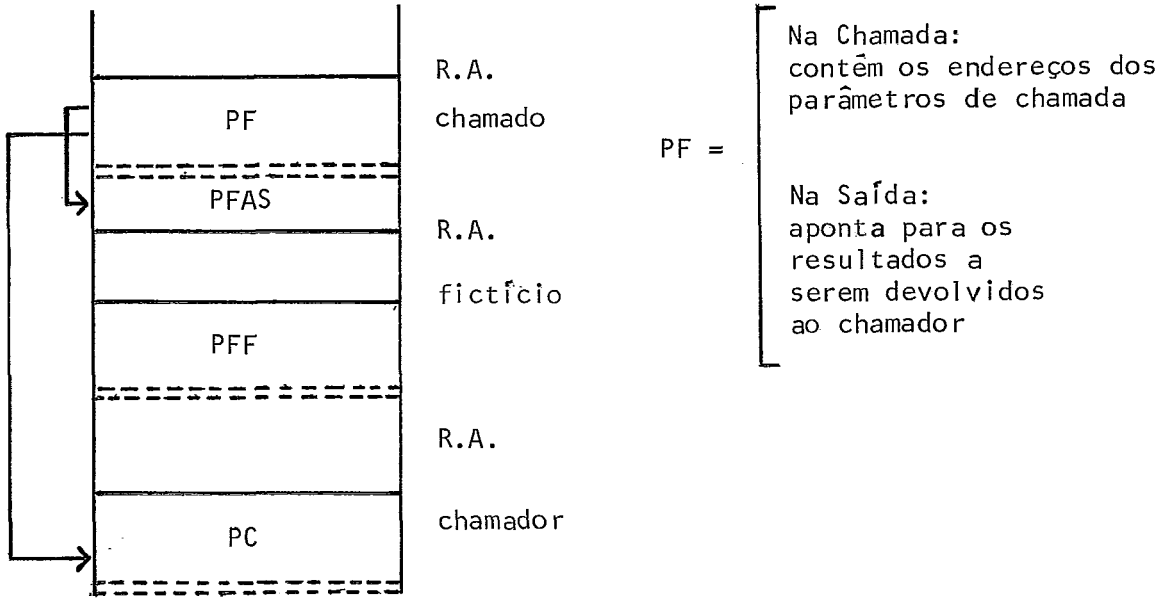


Figura IV.6 - Aspecto da pilha de execução, quando subprograma chamador trata parâmetro por passagem de valor e o chamado por passagem de endereço

IV.2.5.6. SUBPROGRAMA CHAMADOR TRATA POR NOME E O CHAMADO TRATA POR PASSAGEM DE ENDEREÇO

O subprograma chamado espera encontrar, no seu R.A., a lista de endereços dos parâmetros tratados por passagem de endereços.

No R.A. Fictício são colocados os endereços dos *thunks*,

no ponto de chamada. O cálculo do *thunk*, pela rotina Trata-Par, fornece os endereços dos parâmetros de entrada. Esses endereços são colocados na área PF do R.A. do subprograma chamado e se manterão inalterados até o fim da execução desse subprograma. Recai no caso discutido em IV.2.5.3.

IV.2.5.7. SUBPROGRAMA CHAMADOR TRATA POR NOME E O CHAMADO POR PASSAGEM DE VALOR

O subprograma chamado espera encontrar no seu R.A. os valores dos parâmetros de chamada, passados por valor. A rotina Trata-Par, desempenha aí as mesmas ações que em IV.2.5.2, isto é, executa os *thunks* cujos endereços estão no R.A. Fictício e acessa os seus valores, colocando-os no R.A. do chamador (na área PF, ver figura IV.2). No fim da execução, os resultados, (todos os parâmetros), serão devolvidos ao chamador.

IV.2.5.8. SUBPROGRAMA CHAMADOR TRATA POR PASSAGEM DE ENDEREÇOS E O CHAMADO POR NOME

Na entrada, os endereços dos *thunks*, no R.A. Fictício, são passados para a área de parâmetros do R.A. do chamado, pela rotina Trata-Par, que desempenha também, ações adequadas a cada uso ou definição de um desses parâmetros.

A cada uso de um parâmetro no subprograma chamado, seu endereço é calculado e o valor obtido, devendo ser convertido para LE, se necessário. A cada definição de um deles, o seu endereço é calculado e o seu valor é alterado, devendo, se necessário, ser convertido antes, para a linguagem do chamador (LR).

No fim da execução nada mais precisa ser feito, pois todos os parâmetros cujos valores foram alterados, já foram convertidos para a linguagem do chamador.

Algoritmo:

```
        ** a cada uso de PFi, a rotina Trata-Par desempenha
           as seguintes ações
Thunki;
        ** o thunk do PFi é calculado a partir do endereço
           obtido, seu valor é alcançado
IF PRi e PEi estão em equiv. fraca THEN
END:= ENDER↑;
RC(Desc-PCi, Desc-PFi, LE);
ENDER↑:=   END;
        ** ENDER aponta aogra, para o valor convertido
        ** a cada definição o endereço do parâmetro é calcu
           lado e o seu valor, convertido se necessário, e
           aí colocado
Thunki;
        ** calcula o endereço do i-ésimo parâmetro e o de-
           volve em ENDER
IF PEi e PRi em equiv. fraca THEN
     END:=ENDER↑;
     RC(Desc-PEi, Desc-PRi, PR);
     ENDER↑:=END;
          ** ENDER aponta para o endereço do parâmetro
             onde é colocado o seu novo valor convertido
             para a linguagem do chamador (LR).
ELSE
     ENDERi↑:=TEMP;
```

IV.2.5.9. SUBPROGRAMA CHAMADOR TRATA POR PASSAGEM DE VALOR E O CHAMADO POR NOME

Na entrada, a rotina Trata-Par copia os endereços dos *thunks* dos parâmetros no R.A. do chamador. Desempenha, também, as ações convenientes a cada uso ou definição desses parâmetros.

A informação sobre o tratamento do parâmetro pelo chamador (passagem de valor), vai ser utilizada na passagem dos resultados para esse subprograma, se o modo de passagem de valor é por resultado ou valor-resultado. Neste caso, a cada definição, o resultado (convertido, se necessário, para LR) é colocado no endereço obtido pelo cálculo do *thunk*.

Algoritmo:

```
** a cada uso de PFi, o seu thunk é calculado e a partir
   do endereço obtido, seu valor é alcançado como no
   algoritmo do ítem anterior (IV.2.5.8).
** a cada definição, a rotina Trata-Par é chamada e de-
   sempenha as seguintes ações
IF PRi é tratado por resultado ou valor-resultado THEN
     Thunki;
     IF PEi e PRi em equiv. fraca THEN
          END:= TEMP;
          RC(Desc-PEi, Desc-PRi, LR);
          ENDERi↑:=END;
               ** ENDERi aponta para o endereço do parâ-
                  metro onde é colocado seu novo valor
                  convertido para a linguagem do chama-
                  dor (LR).
     ELSE
          ENDERi↑:=TEMP;
ELSE
     ** nada é feito
```

IV.2.5.10. SUBPROGRAMAS CHAMADOR E CHAMADO TRATAM POR NOME

Os endereços dos *thunks*, colocados no R.A. Fictício pelo chamador são colocados no R.A. do chamado, pela rotina Trata-Par.

A cada uso do parâmetro, a rotina Trata-Par calcula seu endereço e converte o seu valor, se necessário para LE.

A cada definição do parâmetro a rotina Trata-Par calcula seu endereço e o novo valor (convertido, se necessário para LR) é aí colocado.

IV.2.6. PARÂMETROS DE MODOS VALOR OU VALOR-CONSTANTE

Parâmetros transmitidos por valor ou valor-constante são parâmetros de entrada, passados ou subprogramas chamado no ponto de chamada.

1 - Na entrada

Os parâmetros do subprograma chamado definidos do modo valor (segundo os descritores dos parâmetros enviados correspondentes), são tratados por passagem de valor ao subprograma chamado, conforme descrito em IV.2.5).

A transmissão de um parâmetro valor constante pode ser implementada por passagem de valor ou por passagem de endereço, dependendo do modo de sua linguagem de origem. Essa informação faz parte do descritor do parâmetro e é passada para a rotina Trata-Par que desempenha as ações adequadas, conforme especificado na seção IV.2.5.

2 - Na saída

Os parâmetros do subprograma chamador definidos do modo valor ou valor-constante (se implementado por passagem de valor) não recebem nenhum resultado do subprograma chamado.

IV.2.7. PARÂMETROS TRANSMITIDOS POR RESULTADO

Um parâmetro transmitido por resultado é um parâmetro de saída, ou seja, não vai ser transmitido nenhum valor ao correspondente parâmetro formal, na chamada de um subprograma.

1 - Na saída

Os parâmetros do subprograma chamador definidos do modo resultado (segundo os descritores correspondentes dos parâmetros recebidos) vão receber os resultados do subprograma chamado por transmissão de valor, conforme discutido na seção IV.2.5 desse capítulo.

IV.2.8. TRATAMENTO DE RESULTADOS DE FUNÇÃO

Se o subprograma chamado é uma função, a rotina TrataPar passa o resultado dessa função para o subprograma chamador. Em seguida, de acordo com a semântica da linguagem dos subprogramas envolvidos, são atualizados os parâmetros de saída, no subprograma chamador, para que sejam mantidos os resultados obtidos como efeitos colaterais ao processamento da função, permitidos em muitas linguagens.

Durante a compilação da configuração, o Verificador tes-

ta a compatibilidade de tipos do resultado da função enviada contra o resultado da função recebida correspondente e determina equivalência forte ou fraca entre eles.

Vamos descrever, através de um algoritmo, um possível tratamento dado a resultados de funções pela rotina Trata-Par.

Notação:

Acrescentamos à notação descrita, anteriormente, os seguintes ítens:

1 - DESC-CH é o descritor de chamada de função, gerado pelo compilador do subprograma chamador, contendo o tipo do resultado e a linguagem em que foi escrito o chamador;

2 - DESC-FC é o descritor da definição da função, gerado pelo seu compilador, com informações semelhantes às descritas no ítem anterior;

3 - RES - é a variável que guarda o resultado da função calculado pelo subprograma chamado;

4 - ARF - variável onde é devolvido o resultado da função ao subprograma chamador.

Algoritmo:

1 - Na entrada

** passagem dos parâmetros de chamada se existirem, pela rotina Trata-Par, de acordo com o modo de tratamento de parâmetros enviados da função. Recai em um dos casos abordados em IV.2.5.

2 - Na saída

```
IF RES e ARF estão em equivalência fraca THEN
     END := RES;
     RC (Desc-CH, Desc-FC, LR);
          ** converte o resultado da função para a lin-
             guagem do chamador (LR);
     ARF := END;
** Atualização dos parâmetros de saída, se existirem, con
   forme descrito a partir de IV.2.5.
```

IV.2.9. SUBPROGRAMA USADO COM PARÂMETRO EM UM MÓDULO QUE O RECEBE

Há dois casos a considerar:

1 - O subprograma usado como parâmetro é recebido de outro módulo;

2 - Um subprograma recebido é parâmetro de outro subprograma também recebido pelo módulo, como no seguinte exemplo:

```
INTERFACE MP1: Pascal
     RECEBE A,B,M1,M2;
     ENVIA X;
     VAR A,B,X: INTEGER;
     PROCEDURE M1(I: INTEGER; FUNCTION M2: INTEGER);
FIM
```

Em qualquer dos dois casos, o módulo de origem do subprograma é ligado junto com os outros módulos que fazem parte da

mesma comfiguração. Dessa maneira, o subprograma ao ser chamado dentro do outro do qual é parâmetro, vai ser tratado como qualquer outro subprograma escrito em linguagem diferente, conforme detalhado neste trabalho.

IV.2.10. TRATAMENTO DE SUBPROGRAMAS ESCRITOS EM LINGUAGENS COM PROCESSAMENTO ESTÁTICO

Há dois casos a serem examinados:

1 - O subprograma CHAMADOR está escrito em FORTRAN:

O subprograma chamador vai colocar na área de dados, na memória, informações equivalentes às que são colocadas no R.A. Fictício. A rotina Prólogo, com acesso a essas informações, cria na pilha, o R.A. Fictício e o R.A. do subprograma chamado. A rotina Trata-Par é chamada para fazer a simulação da passagem de parâmetros, conforme o caso.

No fim da execução, a rotina Trata-Par desempenha as ações necessrias à passagem dos resultados à área de dados do subprograma FORTRAN. A rotina Epílogo restabelece o ambiente de execução do subprograma FORTRAN.

2 - O subprograma CHAMADO está escrito em FORTRAN:

O subprograma chamador cria na pilha, o R.A. Fictício. A rotina Prólogo se encarrega, então, de passar as informações necessárias à área de dados do subprograma FORTRAN . A rotina Trata-Par faz a simulação da passagem de parâmetros considerando a área de dados do subprograma FORTRAN como se fosse o R.A. do subprograma chamado.

No fim da execução, a rotina Trata-Par simula a passagem dos resultados ao subprograma chamador (de acordo com os modos de passagem dos parâmetros recebidos correspondentes) considerando a área de dados do subprograma FORTRAN, como a R.A. do subprograma chamado. A rotina Epílogo faz os ajustes necessários e restabelece o ambiente de execução do subprograma chamador.

CAPÍTULO V

A LINGUAGEM DE CONFIGURAÇÃO

V.1. INTRODUÇÃO

Para tornar a tarefa de definir uma configuração bastante simplificada, considerando-se que podem existir usuários de uma implementação do Ambiente com conhecimentos de uma única linguagem, a LC é relativamente simples, possuindo mecanismos que torna possível a montagem de módulos formando um programa.

As chamadas às ferramentas para manipulação e uso do Ambiente, bem como o acesso às facilidades que permitem a mistura de linguagens, ficam por conta de uma ferramenta denominada interface do Ambiente, descrita no capítulo seguinte.

A LC possui, portanto, mecanismos para definição de módulos e suas respectivas interfaces (tratados no Capítulo I), mais os comandos necessários à definição de uma configuração, que descrevemos a seguir.

No Apêndice B é apresentado um exemplo completo de um

programa definido através de uma configuração.

V.2. Os Comandos da LC

V.2.1. Introdução

Para definir uma configuração é necessário reunir os módulos que a constituem, associar os diferentes objetos que fazem a comunicação entre eles e, finalmente, indicar a ordem de execução desses módulos. Para isso definimos três tipos de comandos, com as funções especificadas, denominados: Junte, Associe e Execute, que passamos a descrever.

Convém ressaltar que o uso de comentários é permitido entre os diversos elementos léxicos da definição de uma configuração.

V.2.2. Comando Junte

Forma Geral:

<comando Junte>::= Junte <lista de nomes de módulos>;

Faz a montagem dos módulos que compõem um programa multi-linguagem, acionando o Editor de Ligações.

V.2.3. COMANDO ASSOCIE

Forma Geral:

<comando Associe>::= Associe <lista de pares associados>;

<lista de pares associados>::=

<par associado> |

<par associado> , <lista de pares associados> ;

<par associado>::=

<nome do objeto recebido> de <nome do módulo>

com <nome do objeto enviado> de <nome do módulo>;

Para ilustrar a sintaxe desse comando, suponha que os módulos M1, M2 e M3 se juntam por uma configuração e que:

M1 envia os objetos X1, Y1, Z1;

M2 envia X2, Y2 e envia Z2;

M3 recebe X3, Y3, Z3;

podemos definir, então, o comando:

Associe

X2 de M2 com X1 de M1,

Y2 de M2 com Y1 de M1,

X3 de M3 com X1 de M1,

Y3 de M3 com Z2 de M2,

Z3 de M3 com Z2 de M2;

com cinco pares de objetos associados.

A compilação de um comando Associe produz chamadas ao Verificador para os testes necessários entre os objetos associa-

dos; no caso de incompatibilidade entre algum par de objetos, o erro é detectado e a compilação da configuração é interrompida. A compilação desse comando aciona também o Editor de Ligações, para resolver as referências externas dos objetos associados.

V.2.4. O COMANDO EXECUTE

Forma Geral:

<comando Execute>::= Execute <lista de nomes de módulos>;

A <lista de nomes dos módulos> indica a ordem lógica de execução das partes de comandos dos módulos aí citados. Se um módulo não tem parte de comandos ou se a sua execução não interessa para a configuração definida, seu nome não precisa ser citado na referida lista.

V.3. A DEFINIÇÃO DE UMA CONFIGURAÇÃO

A definição sintática de uma configuração tem a seguinte forma geral:

```
CONFIG
     <comando Junte>
     <comando Associe>
     <comando Execute>
FIM
```

Os termos CONFIG e FIM, nesse caso, funcionam para o compilador como delimitadores da configuração.

A compilação de uma configuração aciona o Editor de Ligações para juntar os módulos e resolver as referências externas entre eles, sendo por isso necessário que o código objeto de todos esses módulos já estejam na BM. Nessa fase é gerado, também, o código adequado ao controle da execução dos módulos que passam a formar, então, um programa executável.

CAPÍTULO VI

SUGESTÕES PARA UMA IMPLEMENTAÇÃO

VI.1. INTRODUÇÃO

Trataremos aqui de alguns pontos não abordados nos capítulos anteriores e que são subsídios para implementações de Ambientes multi-linguagem.

Na seção VI.2 fazemos uma análise dos diversos modos de passagem de variáveis, com sugestões para otimização, de maneira que o projetista possa escolher os modos de passagem mais convenientes para sua implementação.

Nas seções VI.3 e VI.4 tratamos, respectivamente, da passagem de unidades de encapsulamento e da associação de objetos de espécies diferentes. Ambas as seções contêm subsídios importantes para uma implementação, dependendo do conjunto de linguagens escolhido.

A seção VI.5 serve de diretriz para a definição dos objetos equivalentes em uma implementação, qualquer que seja o con-

junto de linguagens escolhido.

Finalmente, na seção VI.6 sugerimos, através de uma descrição sucinta, a criação de uma ferramenta a ser utilizada em qualquer implementação, denominada Interface do Ambiente, através da qual o usuário terá acesso às facilidades de uso e manipulação do Ambiente.

VI.2. CONSIDERAÇÕES SOBRE A PASSAGEM DE VARIÁVEIS

Conhecendo o funcionamento dos diferentes modos de passagem de variáveis, o programador pode escolher o que melhor se aplica às suas necessidades. Vamos aqui, analisar alguns aspectos da eficiência desse modos de passagem, considerando o seu uso e o tempo de execução.

Na passagem de variáveis em equivalência forte, o modo REF é bastante eficiente. No caso de equivalência forte com variáveis simples é o Editor de Ligações quem põe o endereço da variável enviada (VE) na variável recebida correspondente (VR), durante a ligação dos módulos; com variáveis componentes o cálculo do endereço é feito uma vez só na entrada do módulo e VR passa a apontar diretamente para VE no seu módulo de origem. Não há necessidade de recalcular esse endereço até o final da execução da configuração, nem da aplicação de ações especiais para atualização de seu valor.

Para equivalência fraca, certamente comum de acontecer na passagem de variáveis, o modo REF é caro por envolver chama-

das freqüentes à rotina de conversão (RC).

O modo VALUE e o modo RESULT se aplicam em situações muito particulares. O modo VALUE se aplica a situações específicas onde o programador precisa somente do valor de VE, atribuído à VR na entrada do módulo. A partir desse momento, VR é tratada como uma variável local e nenhuma atualização ou cópia é feita na VE correspondente, por esse módulo, durante ou após a execução. Em equivalência forte exige-se somente o cálculo do endereço de VE para se ter acesso ou seu valor, uma só vez na entrada do módulo. Em equivalência fraca, além do cálculo do endereço de VE é necessária também a conversão do seu valor para LR, o que também ocorre uma vez só na entrada do módulo.

O modo RESULT é aplicado em uma situação oposta a anterior, ou seja, quando o programador usa VR no seu módulo de origem como uma variável local e necessita, atualizar a VE correspondente, no final da execução desse módulo. Esse valor será usado com outras finalidades, por outros módulos da configuração. O seu uso, tanto em equivalência forte como fraca, exige ações semelhantes ao do modo VALUE, aplicadas à VR na saída do módulo para atualização da VE associada.

O uso desses dois métodos não é dos mais caros em termos de execução. Seus custos são equivalentes.

A passagem por nome é eficaz do ponto de vista de segurança de programação, pois a cada uso da variável, o método acessa o seu valor, dando ao programador a certeza de estar controçando exatamente o que ocorre no seu programa. Esse modo de passagem pode ser definido como um modo de passagem de aplicação geral, ou seja, para todas as necessidades de programação

onde se queira passar variáveis entre módulos. Em termos de tempo de execução, entretanto, esse modo de passagem é caro, por envolver freqüentes chamadas aos subprogramas de acesso à variável, para cálculo do seu endereço.

A passagem por VALUE-RESULT pode ser bastante útil em aplicações onde é necessário que a variável recebida seja inicializada com o valor da variável enviada, e a partir daí seja considerada como uma variável estritamente local ao módulo que a recebe. Vamos analisar agora o seguinte exemplo:

```
INTERFACE M1: ALGOL60
 RECEBE vr1(VALUE-RESULT), pp;
 ...
 ...
 integer vr1;
 procedure pp;
FIM

MODULO M1: ALGOL60
begin
 procedure q1;
 begin
  ...
  vr1:= vr1+1
  pp;
  vr1:= vr1+2
 end q1;
 ...
 q1;
 ...
end
FIM
```

```
INTERFACE M2: PASCAL
 RECEBE vr2(VALUE-RESULT);
 ENVIA p2;
 ...
 var vr2: integer;
 procedure p2;
FIM

MODULO M2: PASCAL
 ...
 procedure p2;
  ...
  vr2:= 10;
  ...
 end; (p2)
FIM
```

```
INTERFACE M3: FORTRAN
 ENVIA V1;
 INTEGER V1;
 ...
FIM

MODULO M3: FORTRAN
    ...
C *** parte de inicialização ***
  COMMON /INTERFACE/ V1
  INTEGER V1
  ...
  V1=2
  ...
  END
FIM
```

Supondo que a configuração seguinte defina o programa PROG1:

```
CONFIG
   Junte M1,M2,M3;
   Associe vr1 de M1 com V1 de M3,
           vr2 de M2 com V1 de M3;
           pp de M1 com p2 de M2;
   Execute M3,M1;
FIM
```

A execução de PROG1 começa por M3, que atribui 2 a V1 (V1=2) e segue com a execução de M1. Na entrada de M1, a vr1 é atribuído o valor da variável enviada associada (V1), ou seja, vr1=2; em seguida seu valor passa a 3 (vr1=3). A chamada de pp

produz uma entrada no módulo M2, onde vai ser executado o código de p2 (associado a pp), que começa com a execução do código de inicialização das variáveis recebidas VALUE-RESULT, fazendo então com que o valor de V1 seja copiado em vr2 (vr2=2). Este valor, em seguida, é alterado pelo subprograma p2, e vr2 passa a valer 10. No fim da execução de p2 (saída do módulo M2), o valor de V1 é atualizado para 10. M1, ao retomar o controle da execução, vai incrementar o valor de vr1 (vr1=5).

O fato de vr1 e vr2 representarem a mesma variável enviada — V1 — não é levado em conta por esse modo de passagem, que as considera duas variáveis independentes. Isso nem sempre traduz exatamente as necessidades do programador, que pode imaginar a chamada do subprograma pp em M1 como se fosse substituída pelo código de p2, e a partir desse raciocínio considerar vr1 e vr2 como uma só variável, com dois nomes diferentes, podendo portanto o valor de uma interferir no cálculo do valor da outra. Para expressar esse raciocínio, o programador não deve utilizar o módulo VALUE-RESULT, mas sim o modo IN-OUT para receber as variáveis vr1 e vr2. A semântica desse modo faz com que na entrada de M1 seja executado o código de inicialização com o valor de V1 atribuído a vr1 (vr1=2). O valor de vr1 é, em seguida, incrementado pelo comando vr1:= vr1+1; e passa a ser igual a 3 (vr1=3). A chamada ao procedimento pp faz com que sejam atualizadas todas as variáveis enviadas associadas a variáveis recebidas no modo IN-OUT. Daí V1 passa a valer 3 (V1=3). O controle de execução, ao passar para o código de p2, executa o código de inicialização das variáveis recebidas por M2. Nesse caso a variável vr2 toma o valor de V1, atualizado com o valor de vr1, portanto vr2=3. Durante a execução de p2, vr2 passa a

valer 10; no final de p2 é executado o código de finalização das variáveis IN-OUT, produzindo a atualização do valor de V1 (V1=10). M1, ao retomar o controle da execução no ponto seguinte a chamada de pp, re-inicializa as variáveis IN-OUT e, portanto, vr1 passa a valer 10 (vr1=10). Em seguida, esse valor é incrementado (pelo comando vr1:= vr1+2), e então, vr1=12.

A semântica do modo IN-OUT é mais clara que a do modo VALUE-RESULT, e portanto, fica mais fácil para o programador, entender o que ocorre com seu programa.

O modo IN-OUT presta-se para as mesmas aplicações do modo NAME, sendo mais eficiente que este, por não necessitar, a cada uso ou definição de VR de recalcular o endereço da variável VE associada, e converter o seu valor, (no caso de equivalência fraca); isso é feito somente a cada entrada e saída externa ao módulo que recebe VR.

Uma sugestão de otimização:

Em alguns casos, seria possível otimizar a passagem de variáveis em modo REF e IN-OUT, por meio da definição de um OTIMIZADOR que analisasse as informações obtidas não só na interface de cada módulo, como também na definição da Configuração, aplicando técnicas semelhantes às de análise de fluxo de dados interprocedural (AHO e HULLMAN [13]). Em situações onde fosse possível assegurar otimizações, os algoritmos para modo REF e IN-OUT poderiam ser alterados. Por exemplo, no caso em que ocorrem VE's compartilhadas por módulos diferentes de uma mesma configuração, onde chamadas de subprogramas recebidos por um dos módulos, altera algumas dessas variáveis enviados, como mostrado no exemplo seguinte:

```
INTERFACE M1: L1
 RECEBE (I, J, K) IN-OUT, P ;
 ...
 ...
FIM

MODULO M1: L1
 ...
 P1;  ** chamada a P2**
 ...
 ...
FIM
```

```
INTERFACE M2: L2
 RECEBE (I1, J1, K1) IN-OUT;
 ENVIA P2;
 ...
FIM

MODULO M2: L2
 ...
 PROCEDURE P2;
 ...
 I1:=...
 ...
 END
FIM
```

```
INTERFACE M3: L3
 ENVIA A, B, C;
 ...
FIM

MODULO M3: L3
 ...
 ...
FIM
```

O seguinte comando Associe define as relações entre os objetos enviados e recebidos:

```
Associe I de M1 com A de M3,
       J de M1 com B de M3,
       K de M1 com C de M3,
       I1 de M2 com A de M3,
       J1 de M2 com B de M3,
       K1 de M2 com C de M3,
       P1 de M1 com P2 de M2;
```

As variáveis A, B e C de M3 são compartilhadas pelos módulos M1 e M2, sendo que o valor de A (recebida em M2 como I1) é alterada pelo procedimento P2 de M2.

Uma otimização para modos REF e IN-OUT é possível, associando, no código objeto, a cada VR de um desses modos, uma variável booleana – TROCA(VR) – que deve ser ligada, cada vez que uma das tais VR's tiver seu valor alterado. Antes de cada saída externa produzida pela chamada de um subprograma recebido pelo módulo, devem ser atualizadas somente as VE's recebidas também pelo módulo onde é definido tal subprograma e por ele usadas.

Tal otimização é sugerida, pois no momento em que o módulo está sendo compilado, nada pode ser verificado sobre o subprograma declarado como recebido em sua interface. Isto porque eles não fazem parte de nenhuma configuração, ainda. Portanto, como uma variável enviada pode ser compartilhada por vários módulos diferentes, não sabemos, nesse caso, se o subprograma em questão pode ou não alterar o valor das mesmas variáveis recebidas que o módulo onde é chamado.

Algoritmo otimizado para modo REF em equivalência fraca:

1 - ** Na entrada do módulo executar o código de inicialização e atribuir os valores convertidos das VE's a temporárias: (conforme III.2.5.2).

2 - ** A cada definição de uma VR, o compilador do módulo liga uma variável booleana – TROCA(VR) – associada a cada VR modo REF, avisando que houve mudança de valor atribuído à sua temporária. Com isto evitamos que, a cada saída externa ao módulo, sejam atualizadas, indiscriminadamente, todas as variáveis modo REF em equivalência fraca, isto porque estamos considerando que a VE vai ter seu valor convertido e atribuído à respectiva temporária, na entrada do módulo.

. TROCA(VR) := TRUE;

3 - ** Antes de ocorrer uma saída para fora do módulo (através de um subprograma recebido, ou no fim propriamente dito do módulo), atualizar os valores das VE's. No caso da saída externa ter sido provocada pela chamada de um subprograma recebido pelo módulo, devem ser atualizadas somente as VE's recebidas também pelo módulo onde é definido (enviado) esse subprograma, cujas VR's foram alteradas.

```
IF TROCA(VR) THEN
 BEGIN
   END: = VR; ** põe valor de VR no endereço conhecido;
   RC (DESC-VR, DESC-VE, LE);
   ENDEREÇO↑:= END; ** põe em VE o valor convertido;
 END
```

4 - No caso da saída externa ter sido produzida por um sub-

programa recebido pelo módulo, o retorno da execução ao ponto de chamada, acarreta a re-inicialização das variáveis REF recebidas, também por tal subprograma.

Algoritmo otimizado para modo IN-OUT em equivalência forte ou fraca:

1 - ** na entrada do módulo

. código de inicialização como do modo VALUE para equivalência forte ou fraca; (conforme III.2.5.4);

2 - ** a cada definição de VR, o compilador do módulo liga uma variável booleana – TROCA(VR) -- associada, também às variáveis IN-OUT, indicando alteração do valor de VR;

3 - ** antes de ocorrer uma saída externa provocada pela chamada de um subprograma recebido, atualizar cada VE, cuja VR correspondente tem sua variável booleana ligada e que seja também recebida pelo subprograma chamado;

4 - ** ao retornar de uma saída externa:

re-inicialização das variáveis IN-OUT recebidas também pelo módulo do subprograma;

. código de inicialização como do modo VALUE para equivalência forte ou fraca; (conforme III.2.5.4);

5 - ** na saída do módulo

. código de finalização como do modo RESULT, para equivalência forte ou fraca; (conforme III.2.5.5.).

A otimização sugerida nos algoritmos anteriores (para mo

do REF e IN-OUT), torna-se cara, em termos de tempo de execução, pois as técnicas sugeridas para aplicação pelo OTIMIZADOR não são simples.

Vamos a seguir, através de um exemplo, mostrar uma situação em que não é prático aplicar uma otimização.

```
INTERFACE M1: L1                INTERFACE M2: L2
 ENVIA X[I], X[J];               RECEBE (Y,Z) REF;
 ...                             ...
 ...                             ...
FIM                             FIM

MODULO M1: L1                   MODULO M2: L2
 ...                             ...
 ...                             Y = 1;
                                 WRITE (Z);
FIM                              ...
                                FIM
```

Suponha que Y e Z são associados com X[I] e X[J] respectivamente, em equivalência fraca. O código de inicialização a ser executado na entrada de M2 fará:

- cálculo do endereço de X[I], atribuído a Y; valor convertido de X[I] atribuído à T1;
- cálculo do endereço de X[J], atribuído à Z; valor convertido de X[J] atribuído à T2.

Se o valor de I for igual ao de J, Y e Z apontam para o mesmo elemento de X. Neste caso, o trecho de código apresentado em M2, alteraria o valor de Z, que passaria também a ser

igual a 1. Como os valores de Y e Z são atribuídos respectivamente às temporárias T1 e T2 a aplicação do algoritmo otimizado alteraria somente a temporária T1 associada à Y e o valor de Z permaneceria inalterado, o que não estaria correto pelo uso do modo REF.

Para casos como esse, o algoritmo otimizado para modo REF não funciona. Uma otimização, não seria prática de ser implementada, pois exigiria que o OTIMIZADOR detectasse a igualdade de I e J, o que nem sempre é possível.

VI.3. A PASSAGEM DE UNIDADES DE ENCAPSULAMENTO

Unidades de encapsulamento podem ser entendidas como mecanismos de estruturação, em que um objeto composto é obtido a partir de uma lista de objetos componentes; assim, seu tratamento tem algumas semelhanças com o tratamento de objetos de tipos *record*. Desse modo, a verificação da compatibilidade entre duas unidades de encapsulamento associadas (recebida e enviada) consiste na verificação da compatibilidade entre os objetos que as compõem. O tratamento da passagem de unidades de encapsulamento recai, portanto, no tratamento dos objetos aí definidos, conforme detalhado no decorrer deste trabalho. Esses objetos devem ser da mesma natureza que aqueles permitidos em interfaces de módulos escritos nessas linguagens.

A visibilidade dos objetos definidos em uma unidade de encapsulamento recebida segue as regras da lingaugem em cuja interface aparece.

Outro aspecto a considerar é o fato de que não se indicam, neste caso, os modos de passagem dos objetos componentes de uma unidade de encapsulamento. A passagem de variáveis e constantes, neste caso, deve ser feita por um modo de passagem *default*, previamente fixado. A escolha desse modo recai no mais geral, que é a passagem por nome, mas sua implementação pode ser simplificada, e semelhante à passagem por referência, uma vez que todos os nomes de objetos passados são nomes simples. Nada impede, entretanto, que através da inclusão de declarações adicionais na interface, o usuário possa especificar outros modos de passagem, para cada objeto em particular.

VI.4. ASSOCIAÇÃO DE OBJETOS DE ESPÉCIES DIFERENTES

VI.4.1. INTRODUÇÃO

De acordo com as linguagens escolhidas para uma implementação do Ambiente multi-linguagem pode se tornar conveniente a associação de objetos de espécies diferentes. Estes casos devem ser tratados adequadamente, como exceções às regras de equivalência entre objetos, seguidas pelo Verificador (ver Cap. II).

Vamos, no que se segue, para ilustrar essa questão, tratar da associação entre variáveis e constantes.

VI.4.2. ASSOCIAÇÃO DE VARIÁVEIS À CONSTANTES

Em uma implementação onde sejam escolhidas para mistura linguagens com ou sem definição de constantes, como por exemplo, Ada, FORTRAN e Algol 60 é conveniente que possam ser associadas variáveis à constantes. Para isso é necessário que os dois objetos tenham tipos compatíveis e o objeto recebido seja recebido por valor, garantindo que o objeto enviado não vai ter seu valor alterado pelo módulo que o recebe.

VI.4.3. ASSOCIAÇÃO DE VARIÁVEIS PROCEDIMENTO À SUBPROGRAMAS

Em implementação onde se misturam linguagens que definem tipo procedimento, como Modula-2 ou Algol-68, com outras que não o possuem é conveniente a associação de variáveis tipo procedimento a subprogramas. Isto ocorre de forma conceitualmente semelhante ao visto em VI.4.2.

O termo procedimento é aqui usado tanto para procedimentos quanto para funções. Naturalmente a associação de uma variável declarada como tipo procedimento com resultado (função) deve ser feita a um subprograma função.

Um subprograma associado a uma variável tipo procedimento deve ser tratado, para fins de implementação, como uma constante ou variável tipo procedimento. A compatibilidade entre esses objetos é verificada em função dos tipos e modos de passagem dos seus parâmetros e do tipo do resultado, se fôr o caso, conforme mencionado no Capítulo IV (Seção IV.1).

Na associação desses objetos há dois casos a se conside-

rar:

1 - A variável tipo procedimento é enviada:

Nesse caso, a variável pode assumir valores diferentes durante a execução da configuração.

O procedimento recebido é tratado, então, para fins de implementação, como uma variável recebida por nome. Isso significa que a cada chamada do subprograma recebido é calculado o endereço da variável procedimento associada, para acesso ao código do subprograma a ser executado. Dessa forma é garantido que se o valor de uma variável procedimento mudar, no caso de uma saída externa ao módulo, uma chamada posterior ao subprograma recebido, refletirá essa mudança de valor.

Exemplo:

```
INTERFACE M1; Pascal;

  RECEBE b, c;
  procedure b(i:integer; l:integer);
  procedure c(y1:integer, y2:integer);
  ......
FIM
```

```
MODULO M1: Pascal;

  procedure pp;
  var
     k, l, z1, z1, x1, x2: integer;
  ......

  begin
   ......
   b (k, l);
   ......
   c (z1, z2);
   ......
   b (x1, x2);
   ......
  end
FIM

INTERFACE M2; Modula-2

  ENVIA a;
  var a: procedure (integer, integer);
  procedure pa (xi: integer, xj: integer);
  ......
FIM
```

```
MODULO M2: Modula-2;

  DEFINITION MODULE mm;
   EXPORT QUALIFIED a, pa, p1;
   var a: procedure (integer, integer);
   procedure pa (xi: integer, xj: integer);
   procedure p1 (i: integer, j: integer);
   ......
  END
  IMPLEMENTATION MODULE mm;
    ......
    PROCEDURE p (x: integer, y: integer);
    ......
    END
  PROCEDURE p1 (i: integer, j: integer);
  ......
  END
 PROCEDURE pa (xi: integer; xj: integer);
 ......
 BEGIN
  a:= p1;
 ......
 END
 BEGIN
  a:= p;
 END
END mm;
FIM
```

Definimos, a seguir, uma configuração reunindo esses módulos:

```
CONFIG
   Junte M1, M2;
   Associe b de M1 com a de M2,
           c de M1 com pa de M2
   Execute M2, M1;
FIM
```

A execução começa pela parte de comandos de M2, que atribui à variável *a* o valor *p*; em seguida é executada a parte de comandos de M1, onde a primeira chamada ao procedimento *b* provoca a execução do procedimento *p* de M2. A chamada do procedimento *c*, no módulo Pascal, acarreta a execução do procedimento *pa* de M2, que altera o valor da variável *a* (em M2). De volta ao módulo M1, a segunda chamada de *b* provoca a execução do procedimento *p1* de M2 (o novo valor da variável *a* associada à *b*).

2 - A variável tipo procedimento é recebida:

Neste caso, à variável recebida é associado um valor constante, ou seja, o endereço do subprograma de acesso ao código do procedimento enviado. Portanto, na implementação, o subprograma enviado é tratado como uma constante. Como disciplina de programação, a variável recebida deve, então, ser declarada do modo VALUE, já que seu valor não vai ser alterado.

```
INTERFACE M1: Pascal;
  ENVIA b;
  procedure b(i:integer, 1:integer);
  ...
FIM
```

```
MODULO M1: Pascal;
  ...
  procedure b(i:integer, l:integer);
   ...
  end
  ...
FIM

INTERFACE M2: Modula-2;
  RECEBE a VALUE;
  var a: procedure (integer, integer);
  ...
FIM

MODULO M2: Modula-2;
  DEFINITION MODULE mm;
  var a: procedure (integer, integer);
   ...
  END
  IMPLEMENTATION MODULE mm;
    FROM INTERFACE
    IMPORT a;
    VAR k, l: integer;
    ...
    BEGIN
     ...
     a (k, l);
     ...
    END;
  END mm;
FIM
```

Uma configuração reunindo os módulos é definida a seguir:

```
CONFIG
   Junte M1, M2;
   Associe a de M2 com b de M1;
   Execute M2;
FIM
```

A execução começa pela parte de comandos de M2, onde a chamada à a(k, l) produz a execução do procedimento b de M1.

VI.5. CONJUNTO IMPLEMENTÁVEL DE OBJETOS EQUIVALENTES NAS LINGUAGENS MISTURÁVEIS

VI.5.1. INTRODUÇÃO

Na prática, fixado o conjunto de linguagens para uma implementação de um Ambiente multi-linguagem, ao determinarmos o conjunto de objetos de cada uma delas que poderão ser passados nas interfaces, devemos obedecer ao princípio de que se um objeto de uma linguagem pode ser considerado (fracamente) equivalente a algum outro de alguma das outras linguagens, então sua passagem na interface deve ser considerada. Em se tratando de objetos de utilização muito restrita, ou se a conversão ou o processamento adicional para sua passagem forem excessivamente caros ou complicados, o princípio anterior não se aplica. Ainda assim, se for de interesse do programador e previsto na implementação do Ambiente por ele usada, esses objetos podem ser

passados entre os módulos, só que sem testes de compatibilidade entre eles (ver opção de desligamento de testes – na seção seguinte).

Com base nessas idéias e levando em conta que outras espécies de objetos passados em interfaces, além de tipos, são normalmente construídos a partir de tipos a eles associados analisaremos, também, os tipos mais comuns à maioria das linguagens para uma implementação de um Ambiente multi-linguagem, que sugerimos sejam considerados como tipos equivalentes.

VI.5.2. OPÇÃO PARA DESLIGAMENTOS DE TESTES NA DEFINIÇÃO DA CONFIGURAÇÃO

O projetista de um Ambiente multi-linguagem pode incluir, em sua implementação, opções que permitam ao programador suprimir alguns testes para maior flexibilidade de associação entre os objetos.

Uma sugestão para implementação de tais opções é a de tratar esses objetos como se existisse equivalência forte entre eles, deixando ao programador a responsabilidade pelas conseqüências do uso dessa facilidade, que lhe cabe considerar.

Uma outra sugestão de implementação é a de tratar os objetos associados como se fossem recebidos por referência em equivalência forte e, portanto, o objeto recebido deverá apontar diretamente para o valor do enviado correspondente.

VI.5.3. OS TIPOS EQUIVALENTES

VI.5.3.1. INTRODUÇÃO

Os objetos em equivalência forte levam à execução mais eficiente do programa com mistura, uma vez que a passagem do objeto para o módulo que o recebe é feita sem necessidade de conversão. Para isso é importante que, por construção do Ambiente, sejam criadas algumas condições necessárias para que a equivalência forte ocorra em muitos casos de associações de objetos. Sugerimos, então, um conjunto de tipos básicos comuns à maioria das linguagens misturáveis que, por construção de seus compiladores, devem ter a mesma implementação, garantindo dessa maneira, a ocorrência de muitos objetos passados entre os módulos em equivalência forte. Além desses, considerando as razões expostas no ítem anterior, sugerimos que sejam implementados como equivalentes, objetos construídos a partir de alguns descritores genéricos de tipos. Vamos fazer algumas observações sobre esses tipos sem entretanto, tentar esgotar o assunto. Consideraremos os descritores de tipos *array*, *record*, de acesso e de enumeração.

Não consideraremos tipos *file*, porque seu uso é restrito, e de qualquer forma a maioria das linguagens permitem (através de uma interface específica com o sistema operacional), acesso geral a todos os arquivos, e a comunicação através dos arquivos não precisa passar pela interface que se define neste trabalho.

Tipos intervalo, como os tipos *subrange* de Pascal, também não serão aqui considerados. Nossa atitude a respeito será

semelhante à da linguagem Ada: intervalos não definem tipos, definem apenas restrições aos valores de certas variáveis, que devem ser testadas em tempo de execução, automaticamente, por código gerado especificamente com esta finalidade, ou, na sua falta, por código acrescentado pelo programador, caso esse teste seja por ele considerado importante. Assim, para nós, o tipo *subrange* de Pascal *1..10* define variáveis do tipo *integer*, cujo valor deve ficar entre 1 e 10.

Tipos *set* também não serão aqui considerados, por se limitarem à Pascal e Modula-2 tendo implementação totalmente voltada às restrições de *hardware* (em função da unidade endereçável de memória), além de uso restrito mesmo nessas linguagens. De qualquer forma, a simulação de um tipo *set* é suficientemente simples, na maioria das linguagens que possuem definição de tipos.

VI.5.3.2. TIPOS BÁSICOS

Nas diversas linguagens que consideramos, sempre encontramos tipos que representam valores inteiros, reais, caracteres, booleanos (ou lógicos), e algum tipo de *array* de caracteres, para a representação de cadeias de caracteres. Consideraremos aqui que os tipos básicos *Integer*, *Real*, *Char*, *Boolean* e *String* são definidos em todas as linguagens (que os possuem) da forma mais apropriada ao *hardware* da máquina utilizada para a implementação do Ambiente. Se haverá ou não nas diversas linguagens outros tipos básicos distintos em implementação destes citados, é um problema a ser resolvido de acordo com as características de projeto de cada linguagem. De qualquer forma, o

princípio visto acima se aplica: estes tipos básicos mencionados serão considerados fracamente equivalentes a outros tipos que visem representar a mesma abstração. Assim, por exemplo, o tipo *long-integer* de Ada que também representa inteiros, será considerado fracamente equivalente a *Integer*.

VI.5.3.3. TIPOS ARRAY

Dois pontos podem compremeter a equivalência entre os tipos *array* das diversas linguagens que estamos considerando:

-1- Em Fortran, a arrumação de matrizes se faz por colunas, de forma que contrasta com a definição de matrizes, em outras linguagens. A presença em Fortran de declarações como COMMON e EQUIVALENCE fazem com que a arrumação na memória seja uma característica essencial da linguagem: na maioria das linguagens considera-se deselegante escrever um programa cujo bom funcionamento depende da organização das variáveis na memória.

-2- Em várias linguagens, os limites dos valores de índices das diversas dimensões de um *array* fazem parte do valor de uma variável daquele tipo, e não da definição do tipo. Ada define com cuidado estes conceitos, falando em "subtipo": ao declarar uma variável indicamos seu tipo e as restrições aplicáveis; entre estas, os intervalos dos valores dos índices. Em alguns casos, definem-se os limites em tempo de criação da variável *array*, e os limites se mantem por toda a vida da variável; em outros, como no caso do *flex* do Algol 68, um comando de atribuição pode alterar o valor da variável, alterando também os seus limites.

O tratamento de *arrays* deve levar os dois pontos acima em consideração. Implementar todos os *arrays* como o *flex* do Algol68, isto é, dinamicamente no *heap* (exceto os de FORTRAN) seria conceitualmente simples, uma vez que a equivalência forte seria a regra e não a exceção, mas seria também bastante ineficiente, uma vez que aumentaria o código gerado a menos de otimizações não muito simples. Propomos então que os *arrays* a serem passados na interface serão sempre de tamanhos fixos, o que imporá ao programador que dispõe de *arrays* variáveis em sua linguagem a necessidade de definir valores máximos para os limites, todas as vezes que quiser passar um *array*. Ainda assim, não teremos equivalência forte em todos os caos, uma vez que tipicamente, a implementação de *arrays* de tamanhos fixos em linguagens que permitem *arrays* variáveis segue o caso mais geral. O caso do FORTRAN, já mencionado, deve também ser tratado.

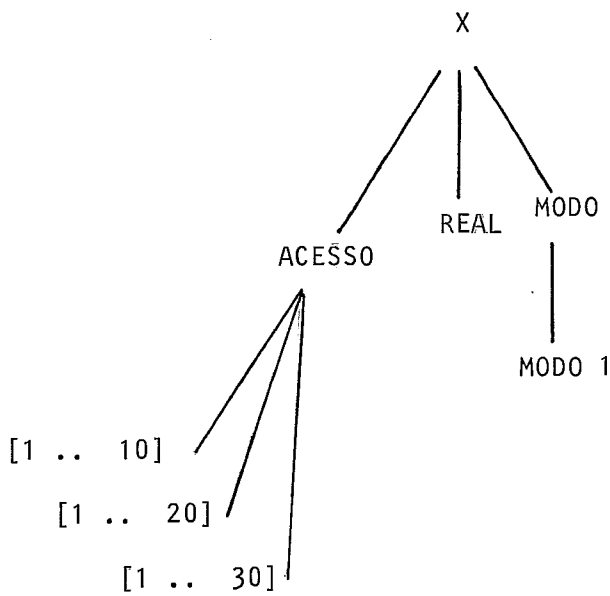
Vamos, apenas para melhor caracterização, considerar nos exemplos três linguagens: FORTRAN, Algol-60 e Pascal e mostrar como poderiam ser os descritores de alguns exemplos típicos, criados a partir do descritor genérico de tipo *array*. Os modos 1, 2, ou 3 que aparecem nos descritores dos exemplos correspondem a forma de implementação do *array* em cada uma das linguagens mencionadas antes. O modo 2 corresponde à implementação mais simples, a usada em Pascal: um valor *array* Pascal é construído pela simples justaposição dos valores das componentes: o modo 1, correspondente à implementação de *array* FORTRAN é semelhante a anterior, exceto pela forma de identificação das componentes: o modo 3 corresponde à implementação de um *array* Algol, feita em duas partes: um descritor que contém os limites dos índices e um ponteiro para a outra parte, que contém os

valores justapostos das componentes, de forma semelhante à vista no primeiro caso.

Assim a declaração FORTRAN

DIMENSION X(10,20,30)

associa a *X* a representação:



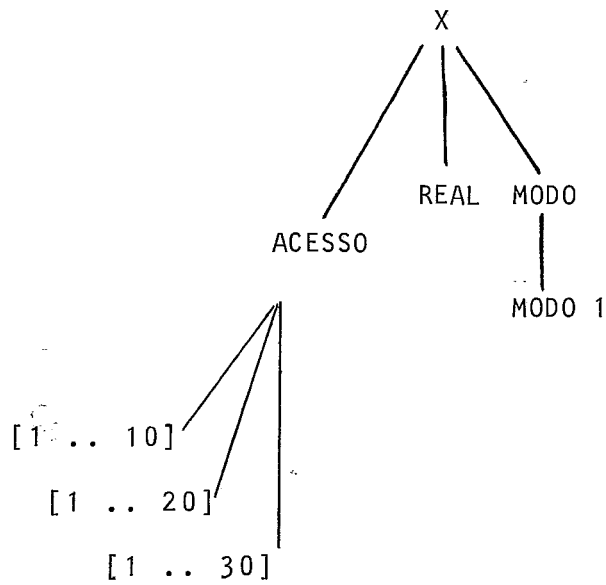
O objeto correspondente a X em Pascal é:

X: *array* [1..10, 1..20, 1..30] *of real*,

ou, equivalentemente,

X: *array* [1..10] *of array* [1..20] *of array* [1..30] *of real*,

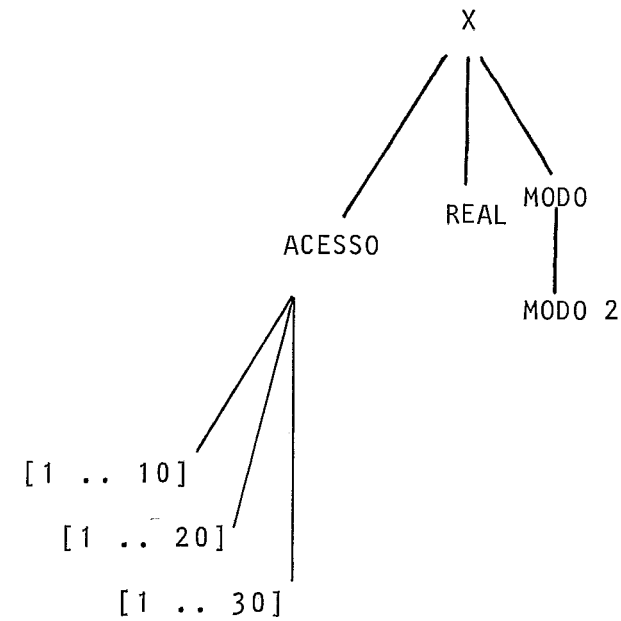
é representado por:



Em Algol-60 teríamos:

INTEGER ARRAY X[1:10,1:20,1:30];

para declarar uma variável de tipo correspondente aos considerados anteriormente. Neste caso a representação seria:

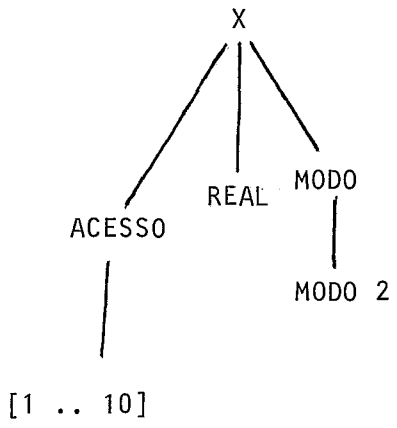


Os objetos X, considerados anteriormente, são fracamente equivalentes, em qualquer combinação das linguagens dois a dois considerada. Lembramos que a indicação dos intervalos é opcional: entendemos todas essas indicações de intervalos como equivalentes a referências ao tipo básico *integer*. Em FORTRAN, este é o único tipo possível para índices, mas em outras linguagens outros tipos (ou intervalos desses tipos) podem ser utilizados.

O único caso de equivalência forte que podemos considerar aqui é o de "vetores", entre FORTRAN e Pascal, como veremos no exemplo a seguir. Em FORTRAN,

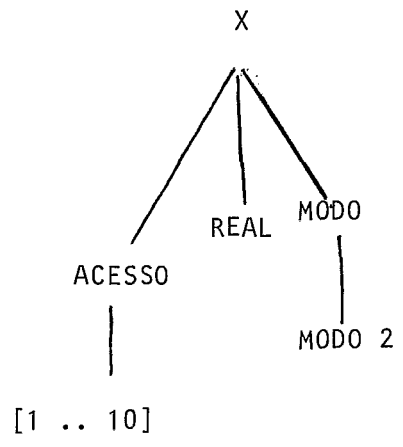
DIMENSION X(10)

tem a representação:



enquanto em Pascal, X tem o tipo

array [1..10] of real



com exatamente a mesma representação, o que caracteriza os dois objetos como fortemente equivalentes, uma vez que para *arrays* em FORTRAN a diferença na arrumação da memória só se torna aparente a partir da segunda dimensão.

O caso de outras linguagens é semelhante e pode ser resolvido da forma vista aqui.

VI.5.3.4. TIPOS RECORD

A implementação de tipos *record* é mais ou menos uniforme em todas as linguagens, no caso de *records* com composição uniforme (sem variantes). Neste caso a implementação habitual é por justaposição simples dos valores dos diversos tipos.

Quando se trata de permitir variação na composição, entretanto, as soluções oferecidas variam muito. Citaremos aqui apenas a união discriminada de Algol-68, (o único mecanismo cla

ramente seguro de tratar o problema), a união não discriminada de tipos em C, os tipos *records* com variantes de Pascal e os tipos *records* com discriminantes de Ada. A cada um destes corresponde uma forma de implementação diferente, mas não residiria aqui o problema. Este aparece pela dificuldade de fazer a correspondência entre as intenções dos programadores nos diversos casos, ou seja, entre as abstrações associadas. Esta correspondência é fundamental, pois ela é que define as formas de conversão de valores e/ou tratamento adicional que deve ser aplicado, sempre que objetos forem passados.

Por essa razão, a proposta é de que apenas *records* com composição fixa, ou seja, sem variantes, sejam considerados na interface. Nestas condições, dois objetos tipo *record* serão equivalentes dependendo da equivalência dos tipos das suas componentes. Os nomes e as formas de seleção de componentes não serão considerados para fins de equivalência.

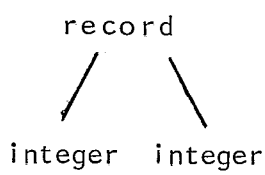
Assim, por exemplo, os objetos

record a,b:integer end;

em Pascal, e

record x:integer; y:integer; end record;

em Ada são fortemente equivalentes, uma vez que ambos correspondem ao descritor:



Por outro lado, nas mesmas linguagens,

record a,b:integer end;

e

record x:integer; y:long-integer; end record;

seriam apenas fracamente equivalentes, sendo suas representações

A equivalência entre os tipos *record* não é forte porque a equivalência entre os tipos das segundas componentes não o é.

VI.5.3.5. TIPOS DE ACESSO

Os tipos de acesso (*pointer*, *ref*, *access*) são utilizados com duas finalidades principais:

-1- para permitir definições recursivas de tipos, e

-2- para compartilhar um grupo de dados, dispensando sua cópia, permitindo várias referências à mesma área de dados.

As mesmas razões fazem com que seja importante associações desses objetos para se programar no sistema aqui descrito. Tais tipos são representados pelo descritos genérico *pointer*.

As regras para definição de equivalência fraca, entretanto, precisam ser bem consideradas. No caso de *arrays* e *records*, o conhecimento de ambos os tipos, e de seus descritores é sem-

pre suficiente para definir a forma de conversão de valores: convertemos os valores das componentes, e novamente arrumamos esses valores na memória. Naturalmente, no caso de *arrays*, deve-se observar nessa arrumação o modo especificado pelo descritor do objeto recebido se se tratar de um descritor diferente daquele usado para o objeto enviado.

No caso de tipos de acesso, há entretanto algumas dificuldades, uma vez que não seria prático converter listas, árvores, ou outras estruturas dinâmicas usadas pelo programador, e às quais a variável de tipo acesso dá entrada. Converter os valores de variáveis apontadas apenas quando esses valores não incluem apontadores para outras estruturas seria obviamente insuficiente para qualquer utilidade prática. Por outro lado, raramente será o caso de termos estruturas complexas e interessantes, em que todos os tipos usados tem equivalentes fortes na outra linguagem, principalmente pela restrição que fizemos aos *records* com variantes. Seria também insuficiente a limitação apenas aos casos em que os tipos apontados são fortemente equivalentes.

A proposta neste caso é simples: aceitar como *fortemente equivalentes* toos os objetos construídos com o auxílio do descritor genérico *pointer*, e deixar por conta do programador de cada módulo a verificação da correção de seus programas através do exame dos detalhes pertinentes de implementação dos tipos utilizados, em ambas as linguagens envolvidas. Como facilidade adicional, uma advertência, no caso em que a equivalência forte não se verifica, seria uma boa idéia, para alertar o programador dos pontos que devem ser observados.

VI.5.3.6. TIPOS PROCEDIMENTO

Como observado anteriormente (seções VI.4 deste capítulo e IV.1 do capítulo IV), para fins de implementação, trataremos unidades de programa tais como subrotinas, funções, procedimentos, fazendo referência a seus tipos, mesmo que a linguagem em que foram programados, ou em que serão ativados não possuam semelhante conceito. As informações que devem distinguir um tipo procedimento do outro são o número, o tipo e o modo de passagem dos parâmetros, além do tipo do resultado, no caso de tipos de funções. O descritor de um tipo procedimento pode ser feito a partir dos descritores genéricos *procedure* ou *function*, e tem como componente uma lista de pares [modo, tipo], para os parâmetros, e adicionalmente, no segundo caso, um tipo adicional, o tipo do resultado.

Definimos a equivalência fraca entre dois tipos *procedure* ou dois tipos *function* dizendo apenas que deve haver equivalência fraca entre os pares de tipos correspondentes. A equivalência forte exige equivalência forte entre os tipos, e que os modos dos parâmetros correspondentes devem ser os mesmos.

Consideramos, é claro, adicionalmente, que a forma de implementação das chamadas e retornos dos procedimentos é sempre a mesma, mas esta é uma das razões pelas quais sugerimos que os compiladores das linguagens misturáveis sejam construídos especificamente para este sistema.

VI.5.3.7. TIPOS DE ENUMERAÇÃO

No caso dos tipos de enumeração, a situação é relativa-

mente simples. Tanto para o programador como para a implementação, os escalares que compõem um tipo de enumeração são constantes cujos nomes substituem códigos numéricos para maior facilidade de programação. A maioria das vezes, a escolha destes códigos numéricos é irrelevante, e pode ser uniformizada para os diversos compiladores das linguagens misturáveis. A exceção parece ser o caso da Ada, uma vez que está prevista a especificação, através de um agregado de valores, dos códigos numéricos a que cada escalar deve corresponder.

Teremos equivalência forte sempre que o número de escalares de um tipo for o mesmo de outro, e a representação for a definida para todos os compiladores. A equivalência será fraca no caso de haver de um tipo para outro, alteração nos códigos numéricos.

VI.6. CARACTERÍSTICAS DOS COMPILADORES DAS LINGUAGENS MISTURÁVEIS

Os compiladores das linguagens escolhidas para uma implementação devem possuir características comuns, como gerar código compatível e dar o mesmo tratamento a objetos enviados e recebidos pelos módulos, conforme especificado nos capítulos anteriores. Além disso, cada compilador deve ser estruturado de maneira que seja possível compilar em separado módulos, interfaces ou unidades de compilação da linguagem correspondente. Essa estrutura (figura VI.1) é vantajosa pois permite o compartilhamento de código do compilador, comum a essas unidades compilá-

veis em separado, além de colaborar para tornar a estrutura de uma implementação de um Ambiente multi-linguagem mais modular, facilitando, até mesmo a inclusão de uma nova linguagem a essa implementação.

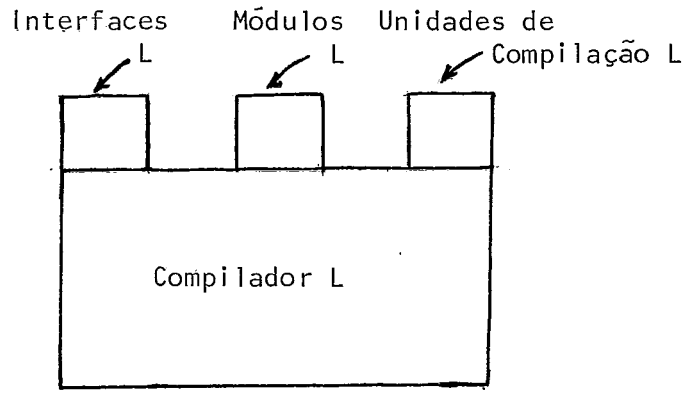


Figura VI.1 - Estrutura geral dos compiladores das linguagens misturáveis

VI.7. DESCRIÇÃO SUMÁRIO DE UMA INTERFACE DO SISTEMA COM O USUÁRIO

Desse caso, denominados por Interface do Ambiente (IA), uma ferramenta que reúne todas as facilidades de uso e manipulação do Ambiente pelo usuário (com acesso a outras ferramentas, bibliotecas, etc.). Esta ferramenta deve ser interativa, dirigida por menus, com acesso não só a algumas facilidades já conhecidas, como o conjunto de compiladores das linguagens misturáveis, a BM, etc., como também a outras facilidades, algumas das quais, até mesmo, voltadas inteiramente para um tipo de

usuário específico.

Em alguns casos pode ser conveniente definir mais de uma IA para uma mesma implementação do Ambiente, cada uma delas com características que levem em consideração o perfil do usuário e/ou as necessidades específicas das aplicações a que se destina essa implementação. Isso significa que para cada IA podem ser definidos conjuntos diferentes de linguagens misturáveis, as facilidades que tornam possível a mistura de linguagens (compiladores, Editor de Ligações, BM, etc.), outras facilidades dirigidas especialmente àquele usuário ou àquela aplicação, além de facilidades de uso geral para manipulação e uso do Ambiente. Nesse caso, as facilidades comuns às diferentes IA's devem permitir o compartilhamento das ferramentas, bibliotecas, etc., correspondentes.

Dentre as facilidades de uso geral destacamos, como sugestão, editores específicos para cada linguagem misturável, (como também para o LC) tornando mais fácil a escrita de unidades de compilação do Ambiente (módulos e suas interfaces) em cada uma dessas linguagens; um sistema para controle de versões: de módulos (e suas respectivas interfaces), programa (reunindo módulos), configuração usada numa certa implementação do Ambiente, ferramentas, etc. Uma outra facilidade útil permitiria ao usuário obter informações sobre módulos e respectivas interfaces, ou programas com mistura compilados e guardados na BM. Isso pode ser implementado como um arquivo onde o programador de uma dessas unidades, coloca as informações sobre elas que considerar importantes. Essas informações estariam disponíveis ao usuário e poderiam ser por ele acessadas através, por exemplo, de um comando Help. Muitas outras facilidades devem

ser incluídas em uma implementação, apresentamos aqui, apenas algumas sugestões.

CONCLUSÕES

Os princípios para programação multi-linguagem aqui propostos se aplicam a linguagens imperativas fortemente tipadas, tais como Ada, Modula-2, CLU. No caso de linguagens com facilidades para paralelismo ou concorrência, consideramos aqui subconjuntos dessas linguagens em que essas características não foram incluídas.

O objetivo principal deste trabalho foi mostrar a viabilidade da programação multi-linguagem em larga escala, produzindo programas que seguem os princípios de qualidade exigidos pela Engenharia de Software. Para isso, examinamos principalmente os aspectos que dizem respeito à comunicação entre os módulos escritos em linguagens diferentes e sua execução sob o controle de uma configuração descrita pelo programador. Esse controle é garantido por meio de testes efetuados com os objetos que passam entre os módulos, produzindo código especial para realizar a passagem desses objetos durante a execução.

Cabe mencionar aqui, entretanto, algumas facilidades, que incluídas à Linguagem de Configuração (LC), dariam a um Ambiente multi-linguagem, de forma integrada, mecanismos de programação em larga escala, que algumas linguagens já possuem, tais como mecanismos para tratamento de exceções, para construção de

software parametrizado ou genérico, e para paralelismo.

A definição de mecanismos de exceção na LC é interessante não só para permitir o controle da execução em presença de falhas eventualmente ocorridas na associação de objetos durante a execução de uma configuração, como também para suprir a falta desses mecanismos nas linguagens que não os possuem. É necessário, entretanto, um estudo mais pormenorizado para que seja possível compatibilizar um mecanismo de exceção que inclua tratadores definidos pelo usuário a nível de configuração, com outros mecanismos já existentes nas linguagens do Ambiente.

A inclusão, na LC, de um mecanismo para construção de *software* genérico (semelhante aos mecanismos existentes em linguagens como Ada e CLU) permitiria a construção de módulos genéricos em qualquer linguagem do Ambiente. Isso exigiria, entretanto, alterações no processo de compilação dos módulos e configurações, aqui descrito.

Seria interessante permitir o uso de mecanismos de programação concorrente/paralela, nas linguagens que já os possuem, ou até estendê-los às linguagens que deles não dispõem, no caso de aplicações onde esses recursos se revelassem de utilidade. A inclusão desses mecanismos exigiria, entretanto, estudo das formas específicas de implementação de um Ambiente multi-linguagem, e de cada linguagem nele incluída.

Extensões a esse trabalho podem ser feitas através do estudo para inclusão de linguagens de diferentes paradigmas, tais como linguagens para programação em lógica, programação funcional e programação voltada à objetos.

Outra extensão possível diz respeito ao estudo dos fatores humanos em Ambientes de programação multi-linguagem. Um usuário típico desse sistema é aquele que apesar de ter tomado consciência que sua linguagem preferida de programação (por exemplo, FORTRAN, para fixar as idéias), pode ser considerada uma linguagem obsoleta quando comparada a linguagens mais modernas, reluta em abandoná-la. O Ambiente multi-linguagem oferece a ela uma oportunidade aparente de continuar programando em sua linguagem, dispondo de facilidades adicionais, como se um novo dialeto mais poderoso da linguagem se tornasse disponível. Com certeza, alguns desses usuários nunca chegariam a considerar por si mesmos, a possibilidade de utilização de novas linguagens, mas através do Ambiente, tomarão conhecimento das facilidades que elas oferecem, e eventualmente passarão a dar preferência às novas linguagens para seu trabalho. O ambiente multi-linguagem proporcionará, assim, para tais usuários, a possibilidade de uma fase de transição mais suave entre uma linguagem antiga e outra mais moderna.

A direção natural de continuação deste trabalho é a definição e a implementação de um Ambiente multi-linguagem completo. O esforço a ser dispendido em tal implementação supera o escopo de uma tese de doutorado.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] WOLBAG, R. John, "Comparing the cost of software conversion to the cost of reprogramming". Sigplan Notices, vol. 16(4), 1981.

[2] VOUK, A. Mladen, "On the cost of mixed language programming". Sigplan Notices, vol. 19(12), 1984.

[3] GENTLEMAN, M.W., TRAUB, F.J., "The Bell Laboratories numerical mathematics program library project". Proceedings of the ACM 23rd National Conference, 1968.

[4] RIS, F., "The relationship between numerical computation and programming languages". Discussion in J.K. Reid (Ed.), North-Holland Publishing Company, Amsterdan, 1982.

[5] KERNIGHAN, W. B., MASHEY, R.J., "The unix programming environment". Software - Practice and Experience, 9, 1979; [6] X3, "Intra language compatibility guideline", SPARC/81-842A. Sigplan Notices, vol. 17(7), 1982.

[6] X3, "Intra Language Compatibility Guideline", SPARC/81-842A. Sigplan Notices, vol. 17(7), 1982.

[7] DARONDEAU, H.P., GUERNIC, P., RAYNAL, M., "Types in a mixed language system". Bit, 21(1981).

[8] EINARSSON, B., "Mixed language programming". Software - Practice and Experience, 4, 1984.

[9] MITCHELL, G.J., MAYBURY, W., SERRT, R., MESA language manual. Xerox Research Center, Palo Alto, Cal., CLS-79-3, 1979.

[10] WIRTH, N., Programming in Modula-2. Springer-Verlag, 1982.

[11] "Ada Programming Language", Military Standard ANSI/MIL-STD-1815A, 1983.

[12] HOARE, C., "Data Structures", Currents Trends in Programming Languages Methodology, vol. IV, Prentice-Hall, 1978.

[13] AHO, A., SETHI, R., Compilers, principles, techniques, and tools, Addison-Wesley Publishing Co., 1986.

[12] "Programming Language Fortran", American National Standart ANS X 3.9, 1966.

[15] WIRTH, N., JENSEN, K., Pascal user manual and report, Springer-Verlag, 1975.

[16] KERNINGHAM, B., RITCHIE, D., The C Programming Language, Prentice-Hall, 1978,

[17] LISKOV, B et all, CLU Reference Manual, Springer-Verlag, 1981.

[18] NAUR, P., "Revised Report on the Algorithmic Language Algol-60", Comm ACM 6, 1 pp 1-17, 1963.

[19] WIJNGAARDEN, A., "Report on the Programming Language Algol-68", Num. Math. 14, pp 29-218, 1969.

[20] GHEZZI, C., JAZAYERI, M., Programming Language Concepts, John Wiley & Sons, 2/E, 1987.

[21] RANGEL, J.L., Projeto de linguagem de Programação, Edição EBAI, 1988.

[22] PEREIRA, R., CARVALHO, S., "Descrição de um Ambiente de Programação com Mistura de Linguagens e seu Funcionamento", Sétimo Congresso de Metodologias en Enginieira de Sistemas, Santiago, Chile, 1987.

[23] _____, "Ambiente de Programação com Mistura de Linguagens: Comunicação entre módulos de um programa", Anais do Primeiro Simpósio Brasileiro de Engenharia de Software, pp 22-31, SBC, COPPE/UFRJ, 1987.

[24] _____, "An Environment for Mixed Language Programming", Conferência International sobre Informática, Havana, Cuba, fevereiro 1988.

[25] MITCHELL, C., "Engineering VAX Ada for Multi-Language Programming Environment", In Proceedings of the 1986 ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, 1987.

[26] REPPY, J., GANSNER, E., "A Foundation for Programming Environments", In Proceedings of the 1986 ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, 1987.

[27] ANCONA, M.; "Integrating Library Modules into Pascal Programs", Software-Practice and Experience, vol. 14(5), pp 401-412, May 1984.

[28] HABERMANN, N.A., "The Gandalf Project", German Chapter of the ACM Berichte, 18, Programmier-Umgebungen und Compiler, BG Teubner, Stuttgart, pp 281-284, 1984.

[29] WOLFE, I.M., et all., "The Ada Language System", IEEE Computer Society/Computer, pp 37-45, June 1981.

[30] SHAW, M., ALMES, G., WULF, W., "A Comparison of Programming Languages for Software Engineering", Software-Practice and Experience, vol. 11, pp 1-52, 1981.

[31] BOOM, J., DE JONG, E., "A Critical Comparison of Several Programming Language Implementations", Software-Practice and Experience, vol. 10, pp 435-473, 1980.

[32] FEUER, A., GEHANI, N., "A Methodology for Comparing Programming Languages", Comparing & Assessing Programming Languages, Prentice-Hall, Inc., pp 197-200, 1984.

[33] ZEIGLER, S., ALLEGRE, N., MORRIS, J., "Ada for the Intel 432 Microcomputer", IEEE Computer Society/Computer, pp 47-64, June 1981.

[34] KAMRAD, M., Runtime Organization for the Ada "Language System Programs", ACM AdaTec, vol. 3, Nov/Dez 83.

[35] STANDISH, T., "Interactive Ada in the Arcturus Environment", Ada Letters, pp 23-34, Jul/Aug 83.

[36] Jr. HOUGHTON, R., "A Toxonomy of Tool Features for the Ada Programming Support Environment", Ada Letters, pp 69-78, Nov/Dec 83.

[37] LAMB, A.L., "IDL: Sharing Intermediate Representations", ACM Transactions on Programming Languages and Systems, vol. 9, No. 3, pp 297-318, 1987.

[38] PRATT, T., Programming Languages Design and Implementation, Prentice-Hall, 1984.

APÊNDICE A

DESCRITORES GENÉRICOS DE TIPO

Representamos aqui, os descritores genéricos dos tipos mais comuns às linguagens que podem fazer parte de uma implementação do Ambiente.

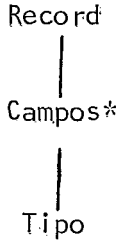
Notação:

* - indica que o nó se repete zero ou mais vezes;

v - indica uma alternativa de escolha entre os filhos daquele nó;

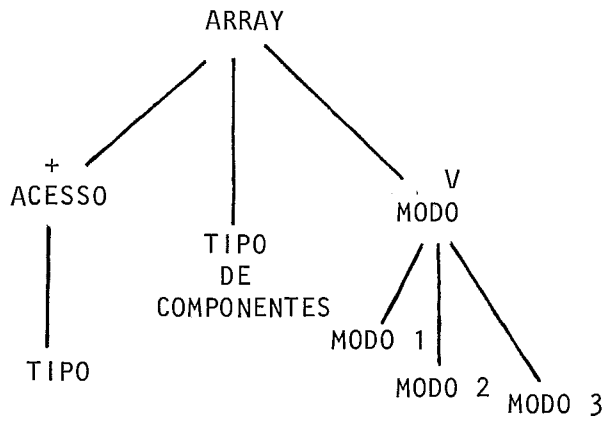
+ - indica que o nó se repete pelo menos uma vez.

1 - Descritor Genérico de Tipos Record

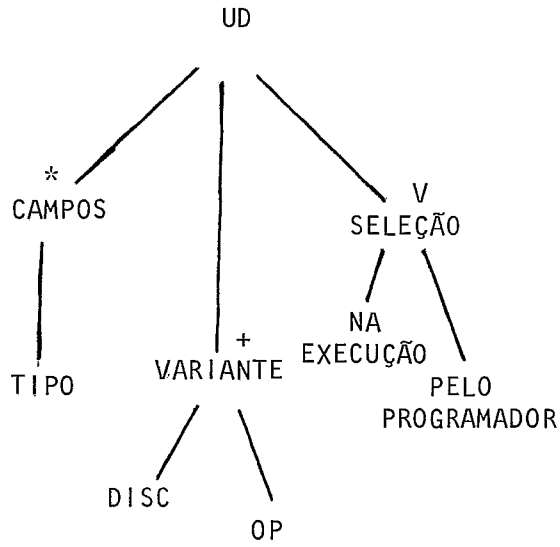


tipos *record*: Ada, Pascal, Modula-2, *struct*: C, etc.

2 - Descritor Genérico de Tipos Array



3 - Descritor Genérico de Tipos União Descriminada



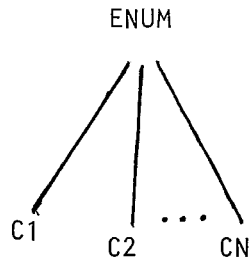
discr - discrimimante

op - opção

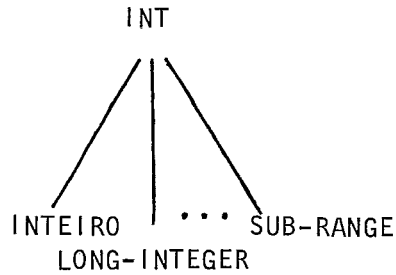
Tipos: *union*: C, Algol-68; *variant record*: Pascal, Ada, Modula-2; etc.

4 - Descritor Genérico de Tipos de Acesso

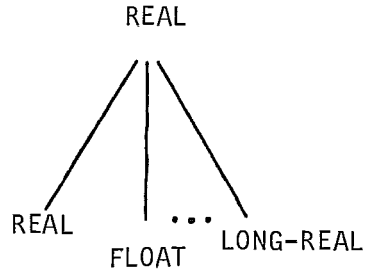
5 - Enumeração



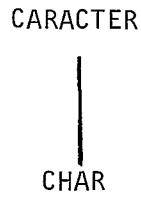
6 - Descritor Genérico de Tipos Inteiros



7 - Descritor Genérico de Tipos Reais

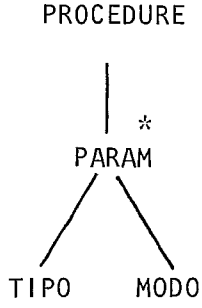


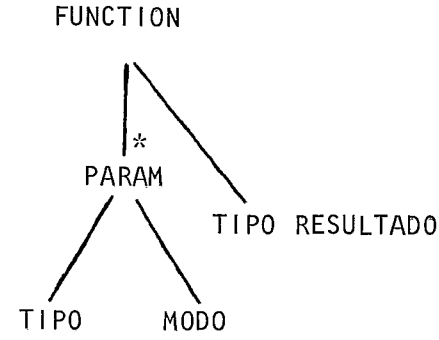
8 - Descritor Genérico de Tipos Caracteres



9 - Descritor Genérico de Tipos Booleano

10 - Descritor Genérico de Tipos Procedimento e Função





* PARAM - parâmetros

APÊNDICE B

UM EXEMPLO COMPLETO DE UMA CONFIGURAÇÃO

Apresentamos aqui um exemplo completo de uma configuração que reune dois módulos escritos, respectivamente, em Ada e Modula-2.

O módulo Ada define um tipo abstrato, denominado pilhas, que é enviado para o módulo USA-PILHA (em Modula-2), que o recebe com o nome *stack*.

```
INTERFACE PILHA1: Ada
  ENVIA pilhas;
  package pilhas is
    type pilha is private;
    function push(p:pilha;i:integer) returns pilha;
    function pop(p:pilha) returns pilha;
    function empty(p:pilha) returns boolean;
    function top(p:pilha) returns integer;
    procedure init(p:out pilha);
  private
    type no is record
      val:integer;
      prox:pilha;
    end;
    type pilha is access no;
  end pilhas;
FIM
```

```
MODULO PILHA1; Ada
  package pilhas is
    type pilha is private;
    function push(p:pilha;i:integer) returns pilha;
    function pop(p:pilha) returns pilha;
    function empty(p:pilha) returns boolean;
    function top(p:pilha) returns integer:
    procedure init(p:out pilha);
  private
    type no is record
      val: integer
      prox:pilha
    end;
    type pilha is access no;
  end pilhas;

  package body pilhas is
    function push(p:pilha;i:integer) returns pilha is
    begin
      return new pilha(i,p);
    end push;
    function pop(p:pilha) returns pilha is
    begin
      return p.prox;
    end pop;
    function empty(p:pilha) returns boolean is
    begin
      return p:=null;
    end empty;
    function top(p:pilha) returns integer is
    begin
      return p.val;
    end top;
    procedure init(p:out pilha) is
    begin
      p:null;
    end init;
  end pilhas;
FIM
```

```
INTERFACE USA-PILHA: Modula-2
  RECEBE stack
  DEFINITION MODULE stack;
    EXPORT QUALIFIED pilha, push, pop, empty, top, init;
    PROCEDURE push (p:pilha, i:integer): pilha.
    PROCEDURE pop (p:pilha): pilha;
    PROCEDURE empty (p:pilha): boolean;
    PROCEDURE top (p:pilha): integer;
    PROCEDURE init (VAR p:pilha);
  END stack.
FIM
```

MODULO USA-PILHA: Modula-2

```
MODULE usa-stack;
FROM stack IMPORT pilha, push, pop, empty, top, init;
CONST
  abrepar  = 1;
  fechapar = 2;
  abrech   = 3;
  fechach  = 4;
  fim      = 0;
VAR s: ARRAY [1...100] OF integer;
  j, n, x: integer;
PROCEDURE check: boolean;
VAR i: integer;
  p: pilha;
  ok: boolean
BEGIN
  i:0;
  int (p);
  ok:= true;
  REPEAT
    i:= i+1;
    x:= s[i];
    CASE x OF
      abrepar, abrech   : push (x);
      fechapar, fechach :
      IF (NOT empty(p)) AND (top(p) = x-1) THEN
        POP(p);
      ELSE
        ok:= false;
```

```
        END;
        fim :
        END;
      UNTIL (NOT ok) OR (x= fim)
      RETURN ok;
    END;
  BEGIN
    Write('>');
    ReadInt(n);
    WriteLn;
    FOR j:=1 TO n DO
      Write('>');
     ReadInt(s[j]);
    END;
    IF check THEN
      WriteString('Ok!');
    ELSE
      WriteString('Erro!');
    END;
  END.
FIM
```

Definição da Configuração:

```
CONFIG
   Junte PILHA1, USA-PILHA;
   Associe stack de USA-PILHA com pilhas de PILHA1;
   Execute USA-PILHA;
FIM
```