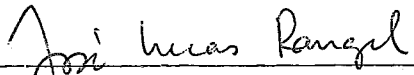


ADA: ESPECIFICAÇÃO DE UM MÓDULO
DE ANÁLISE SEMÂNTICA ESTÁTICA

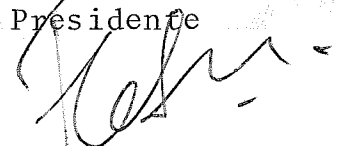
Miguel de Teive e Argollo Júnior

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


Aprovada por:



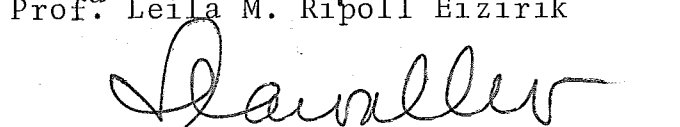
Prof. José Lucas M. Rangel Netto
Presidente



Prof. Estevam Gilberto De Simone



Prof.^a Leila M. Ripoll Eizirik



Prof. Sérgio E. Rodrigues Carvalho

RIO DE JANEIRO, RJ - BRASIL

ABRIL DE 1984

ARGOLLO JUNIOR, MIGUEL DE TEIVE E

ADA: Especificação de um Módulo de Análise Semântica Estática (Rio de Janeiro), 1984.

VII , 111 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1984).

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1. Análise Semântica I. COPPE/UFRJ II. Título (série).

À meus pais
Miguel e Mariinha

AGRADECIMENTOS

A José Lucas Rangel pela amizade, paciência e orientação.

A Estevam De Simone, pela amizade constante desde 1978.

A Valéria, Tarso e Regina, pela amizade e incentivo ao longo de nosso projeto.

Aos amigos da COPPE: Zancanella, Olinto, Lèila, Sueli, Lídia, Gerhard, Antônio, Schneider, Jorge, Manoel, Marta, Betty, Vera, Felipe, Nelson, Edu, Adilson, Michel, Maria, John e Ludmila e a todos que me incentivaram durante esse trabalho.

A Denise, pelo excelente trabalho (ecológico) de datilografia.

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

ADA: ESPECIFICAÇÃO DE UM MÓDULO
DE ANÁLISE SEMÂNTICA ESTÁTICA

Miguel de Teive e Argollo Júnior

Abril, 1984

Orientador: José Lucas M. Rangel Netto

Programa: Engenharia de Sistemas e Computação

Esse trabalho descreve a especificação de um módulo de análise semântica estática para a linguagem ADA que faz parte de um projeto em desenvolvimento no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ.

O módulo recebe como entrada a representação intermediária de uma unidade de compilação de um programa ADA e efetua os procedimentos necessários para a resolução de nomes e expressões, elaboração de declarações, detecção de irregularidades semânticas da unidade de compilação e avaliação dos atributos necessários.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.).

ADA: A MODULE ESPECIFICATION
FOR STATIC SEMANTIC ANALYSIS

Miguel de Teive e Argollo Júnior

April, 1984

Chairman: José Lucas M. Rangel Netto
Department: Engenharia de Sistemas e Computação

This thesis describes a proposed specification for a static semantic analysis module for ADA as part of a project for a compiler under development at the Programa de Engenharia de Sistemas e Computação at COPPE/UFRJ.

The module takes as input an intermediate representation of an ADA compilation unit and executes the procedures for name and expression resolution, declaration elaboration, semantics error detection and semantic attributes evaluation.

ÍNDICE

CAPÍTULO I

INTRODUÇÃO	1
------------------	---

CAPÍTULO II

ALGUMAS CARACTERÍSTICAS DE ADA E DE SEU AMBIENTE DE PROGRAMAÇÃO	6
2.1. TIPOS	7
2.2. COMPILAÇÃO EM SEPARADO	8
2.3. SUBPROGRAMAS	9
2.4. PACKAGES	11
2.5. PROCESSAMENTO PARALELO	15
2.6. TRATAMENTO DE EXCEÇÕES	16
2.7. UNIDADES GENÉRICAS	19
2.8. ASPECTOS DEPENDENTES DE MÁQUINA	22
2.9. ENTRADA E SAÍDA	25
2.10. AMBIENTE DE PROGRAMAÇÃO ADA	26

CAPÍTULO III

DIANA	30
3.1. CONSIDERAÇÕES GERAIS	30
3.2. PRINCÍPIOS GERAIS	33
3.3. NOTAÇÃO	34
3.4. ALGUMAS CARACTERÍSTICAS DE DIANA	41
3.4.1. AMBIGUIDADES NA GRAMÁTICA DE ADA	42
3.4.2. ASPECTOS DE COMPILAÇÃO EM SEPARADO	43
3.4.3. MANIPULAÇÃO DE NOMES	44
3.4.4. DEFINIÇÃO MÚLTIPLAS DE IDENTIFICADORES	47
3.4.5. ESPECIFICAÇÃO DE TIPOS	49
3.4.6. DUPLICAÇÃO DE SUB-ÁRVORES	53
3.4.7. TRATAMENTO DE INSTANCIACIONES	55
3.5. OPÇÕES DE IMPLEMENTAÇÃO	58

CAPÍTULO IV

RESOLUÇÃO DE NOMES E EXPRESSÕES	62
4.1. CONCEITUAÇÃO	62
4.2. ALGORITMOS	66
4.3. DIANA	72
4.4. ESPECIFICAÇÃO DO ALGORITMO	76

CAPÍTULO V

ANALISADOR SEMANTICO ESTÁTICO	80
5.1. ASPECTOS INICIAIS	80
5.2. ESTRUTURA DO ANALISADOR SEMÂNTICO	82
5.3. TRATAMENTO DAS DECLARAÇÕES E TIPOS	86
5.4. NOMES E EXPRESSÕES	87
5.5. COMANDOS	93
5.6. SUBPROGRAMAS	95
5.7. PACKAGES	97
5.8. REGRAS DE VISIBILIDADE	99
5.9. PROCESSAMENTO PARALELO	100
5.10. ESTRUTURA DE PROGRAMAS	101
5.11. TRATAMENTO DE EXCEÇÕES	102
5.12. UNIDADES GENÉRICAS	103
5.13. CLÁUSULAS DE REPRESENTAÇÃO E ASPECTOS DEPENDEN- TES DE MÁQUINA	104

CAPÍTULO VI

CONCLUSÕES	106
------------------	-----

CAPÍTULO I

INTRODUÇÃO

O Departamento de Defesa Americano identificou, em meados da década de setenta, a grande proliferação de linguagens e correspondentes compiladores como um problema a ser solucionado com urgência. Em 1975 foi formado um grupo de trabalho com o objetivo de apontar um número pequeno das linguagens usadas que representasse um conjunto suficientemente poderoso para fazer frente às necessidades do DoD. Esse grupo de trabalho redigiu um documento inicial - STRAWMAN - que continha um conjunto de requisitos a serem obedecidos pelas linguagens selecionadas; esse documento foi amplamente circulado pela comunidade e deu origem a novos documentos, que continham um conjunto de requisitos mais poderosos - WOODENMAN, TINNAN. Nenhuma das linguagens estudadas satisfazia plenamente aos documentos, em parte porque enquanto a tecnologia se desenvolvia os profissionais de software passavam a esperar ferramentas mais poderosas e em parte porque as linguagens de uso geral existentes não tratavam satisfatoriamente as áreas de paralelismo, tratamento de exceções e entrada e saída de dispositivos que operam em tempo real, entre outros aspectos importantes. Um conjunto mais restrito de requisitos foi desenvolvido junto com um novo documento - IRONMAN - que novamente circulou entre a comunidade científica e de usuários. Em agosto de 1977 quatro grupos foram contratados para participar de um concurso visando a especificação de uma nova linguagem de programação que satisfizesse os requisitos existentes no IRONMAN e que seria adotada como padrão pelo DoD; paralelamente, esse documento sofreu algumas modificações dando origem ao conjunto final de especificações que a nova linguagem deveria obedecer, criando o documento conhecido como STEELMAN. Ao mesmo tempo, verificou-se que dificilmente uma linguagem sozinha seria capaz de atender os requisitos levantados; dessa forma uma nova série de documentos - SANDMAN, PEBBLEMAN e STONEMAN - foi produzido e circulou pela comunidade levantando as características

de um ambiente de suporte à programação que respondesse às necessidades crescentes de desenvolvimento e manutenção de software.

Finalmente em maio de 1979 a linguagem definida pelo grupo francês da CII Honeywell_Bull foi a escolhida e a divulgação inicial de seu manual de referência, bem como de seu "rationale", foi realizada em junho de 1979 [DoD (01)]. O nome escolhido para batizá-la foi ADA, uma homenagem à condessa Augusta Ada Lovelace considerada como 1^a programadora da história [Morris e James (5)]. A partir desse ponto uma grande polêmica se levantou na comunidade de informática: enquanto os impiedosos críticos da nova linguagem questionavam seu tamanho, sua complexidade e a falta de um sub-conjunto oficial, seus ardorosos defensores respondiam com sua potencialidade e com o cuidado com que foi definida. WEGNER (06) apontou em janeiro de 1982 que "a literatura de defeitos de projeto de ADA já é (era) volumosa". De qualquer forma, é inegável o interesse provocado pela nova linguagem: em junho de 1983 foi publicada por ROMANOWSKY (07) uma bibliografia com mais de 570 artigos sobre ADA!

No início de 1982 o grupo de linguagens de programação do Programa de Engenharia de Sistemas e Computação da COPPE, preocupado não só com a influência que ADA poderá vir a ter em um futuro próximo mas também com os problemas técnicos que a implementação de uma linguagem desse porte necessariamente provoca, começou a estudar a definição e implementação de um compilador ADA para um computador nacional. A idéia inicial era oferecer temas de tese relacionados com módulos desse compilador, de forma que cada tese deveria constar de um estudo de seu módulo correspondente e a definição, em um nível global, do mesmo; os algoritmos necessários deveriam ser definidos na própria linguagem ADA.

A figura I.1 apresenta a estrutura inicialmente proposta para nosso projeto.

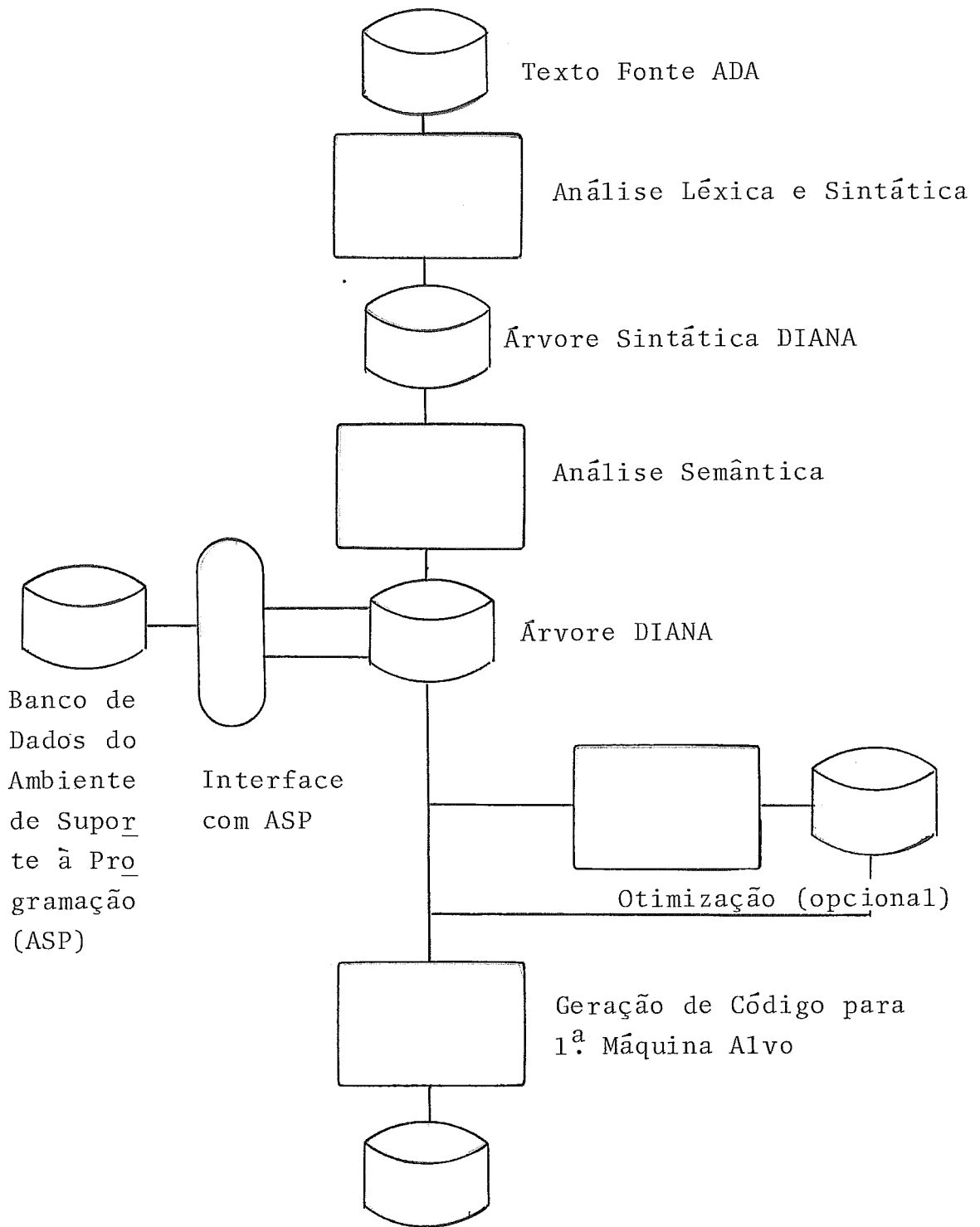


Figura I.1 - Estrutura proposta para o projeto do compilador inicial.

Como pode ser observado, foi proposta uma separação distinta entre uma parte dependente da linguagem - FRONT-END - e uma parte dependente da máquina alvo - BACK-END - , de forma a aumentar o grau de portabilidade do compilador.

O front-end é composto por um analisador léxico formado por um automato finito que percorre o texto de entrada e fornece como saída uma sequência de símbolos codificados que o representa. O analisador sintático deve ser baseado em uma tabela LALR(1) compactada e gerar uma forma intermediária chamada "*Árvore DIANA Sintática*", definida no capítulo 3 dessa tese. Esses módulos, juntamente com a interface com o ASP, formam a 1ª tese desse projeto [CHAVES (08)].

O analisador semântico é responsável pela análise de nomes (resolução de "*overloading*"), verificação de tipo e verificação de contexto. Algumas das informações obtidas pelo analisador semântico são necessárias pelo otimizador e/ou gerador de código, ao passo que outras informações podem ser necessárias para a análise de outras unidades de programas compiladas separadamente. Essas informações ficam agrupadas na árvore sintática DIANA, que passa a ser conhecida por "*Árvore DIANA*". Esse módulo forma a 2ª tese desse projeto e foi o objetivo do trabalho que está sendo apresentado.

Espera-se que os compiladores de ADA realizem otimizações clássicas, tais como eliminação de sub-expressões comuns, movimentação de código e eliminação de código redundante ("*dead code*"); essas transformações dependem da semântica da linguagem sendo analisada. Por outro lado, ADA possui uma série de características, tais como tratamento de exceções, paralelismo, etc ..., que tendem a tornar completo o trabalho de um otimizador. De forma geral, espera-se que durante a fase de definição da linguagem os aspectos relevantes à otimização de um programa ADA tenham sido levados em consideração. Nosso projeto apresenta um módulo de otimização opcional que recebe uma árvore DIANA e aplica os algoritmos necessários para otimizá-la. O estudo dos algoritmos de otimização e a definição desse módulo serão descritos por MELLO (09), na 3ª tese de nosso projeto.

O módulo final de nosso projeto propõe a definição de

um gerador de código para uma máquina alvo específica. Embora a maior parte dos artigos genéricos sobre compiladores ADA proponham o emprego de vários geradores de códigos, produzidos de preferência por um gerador de geradores de código, essa opção não é a mais usual entre os projetos descritos na literatura: na referência (11), publicada em outubro de 1983, a maior parte dos projetos apresentados tinha uma máquina alvo específica. O módulo de geração de código de nosso projeto, que deverá usar algoritmos bastante gerais que permitam o emprego de máquinas alvos de concepções distintas com um pequeno trabalho de adaptação, compõe a 4.^a tese em desenvolvimento, descrita por TRINDADE (10).

Espera-se que os quatro módulos apresentados sirvam como base de um compilador ADA a ser implementado no futuro com tecnologia nacional. Os demais módulos necessários para a implementação do compilador deverão ser definidos durante o próprio projeto. As teses em desenvolvimento, além dos objetivos acadêmicos, devem servir como uma especificação inicial do compilador.

O próximo capítulo apresenta os principais conceitos da linguagem ADA e um resumo das características mais marcantes que um ambiente de suporte à programação deve possuir; o 3.^o capítulo apresenta a forma intermediária usada por todos os módulos do compilador-DIANA. No 4.^o capítulo é apresentado o algoritmo utilizado para resolução de nomes e expressões, um dos mais críticos da fase de análise semântica estática; o 5.^o capítulo mostra, de maneira geral, as soluções utilizadas para a definição do analisador semântico. O 6.^o capítulo contém as conclusões e algumas propostas para continuação do trabalho.

Durante o desenvolvimento dessa tese serão feitas várias referências aos manuais de ADA [DoD (02)] e de DIANA [EVANS (04)]; elas serão marcadas pela indicação do manual referenciado ("ARM" e "DRM" respectivamente) e da seção citada.

CAPÍTULO IIALGUMAS CARACTERÍSTICAS DE ADA
E DE SEU AMBIENTE DE PROGRAMAÇÃO

Esse capítulo discute alguns tópicos da linguagem considerados interessantes e, em linhas gerais, alguns aspectos de seu ambiente de suporte à programação. O material dessa seção foi baseado no manual de referência da linguagem e no rationale da mesma, dos quais vários exemplos foram retirados.

2.1. TIPOS

O conceito atual de tipos em linguagens de programação é tal que não define somente o conjunto de valores que seus objetos podem assumir, mas também especifica as operações que podem ser aplicadas aos objetos. Dessa forma, as propriedades e características comuns a vários objetos devem ser fornecidas em um único lugar; a declaração de tipos serve a esse propósito. Por outro lado, objetos com propriedades distintas devem ser distinguíveis em um programa, devendo ser esse fato verificado automaticamente pelos tradutores da linguagem em questão.

A linguagem ADA foi definida de forma que cada definição de tipo introduz um novo tipo. Dessa forma, a equivalência é nominal e não estrutural: "duas definições de tipo sempre definem 2 tipos distintos, mesmo que sejam estruturalmente iguais (ARM Sc. 3.3.1-8)".

Paralelamente, ADA utiliza também a noção de subtipo: um tipo caracteriza um conjunto de valores e de propriedades estáticas dos objetos desse tipo; restrições podem ser aplicadas sobre alguns tipos (por exemplo, um tipo escalar pode ter seu intervalo diminuído). A declaração de sub-tipos serve para associar restrições a tipos já definidos, não introduzindo um novo tipo.

Por outro lado, a declaração de um tipo derivado introduz um novo tipo que possui suas características herdadas de seu tipo pai; particularmente, o conjunto de valores de um tipo derivado é uma cópia dos valores de seu tipo pai (ARM. Sc. 3.4-4).

A linguagem suporta tipos escalares (enumeração, inteiro e real), o tipo ARRAY, o tipo RECORD e o tipo de ACESSO. Tipos genéricos serão apresentados em uma sub-seção própria; tipos "task" serão apresentados na sub-seção de paralelismo. Os tipos privado e privado limitado podem ser declarados em pacotes, e fazem parte dos mecanismos de abstração de dados da linguagem.

2.2. COMPILAÇÃO EM SEPARADO

A compilação em separado é uma característica desejável numa linguagem de programação, pois permite a separação de tarefas, programação estruturada e a construção de bibliotecas de programas. Deve ficar bem claro a diferença existente entre compilação em separado e compilação independente. Esta vem sendo usada há bastante tempo em linguagens tais como Fortran, PL/I, Assembly, etc, porém apresenta como deficiência o não compartilhamento das informações definidas nas diversas unidades de compilação - UC's - , diminuindo bastante o grau de confiabilidade de um programa composto por várias UC's. A compilação em separado permite uma maior troca de informações entre as diversas UC's de um programa. Quando uma UC é submetida ao compilador, este tem acesso não só ao texto fonte correspondente, mas também às informações de outras UC's. É essa característica que permite um nível maior de testes, principalmente testes de tipos, entre as diversas UC's de um programa, aumentando a confiabilidade deste. O compartilhamento de informações entre as diversas UC's é conseguido através do uso de um arquivo de biblioteca, que contém informações sobre as UC's já compiladas, tais como tabelas de símbolos, data de compilação, relação unidade/subunidade, lista de visibilidade, etc.

Ao ser ativado, o compilador é informado das entidades externas visíveis pela unidade a ser compilada pela cláusula "WITH". Com essa informação, o ambiente necessário à análise semântica (tabela de símbolos) dessa unidade pode ser montado; como outras unidades de compilação podem querer acessar essa unidade, sua tabela de símbolos deve ser salva na biblioteca de programas. Um sistema do nível do proposto por RANGEL et al (18) e especificado em CHAVES (8) deve ser capaz de fornecer os mecanismos necessários para implementação de compilação em separado em nosso projeto.

2.3. SUBPROGRAMAS

Em ADA, essa construção apresenta os conceitos clássicos de linguagens do tipo Algol e incorpora algumas novidades relevantes.

Em primeiro lugar, a linguagem permite que a especificação de um subprograma seja separada de seu corpo. Essa característica pode aumentar a legibilidade de um programa, permitindo que as especificações de todos subprogramas usados sejam fornecidas em uma pequena região de seu texto. Por outro lado, essa característica é fundamental em "packages", pois permite que os detalhes da implementação de subprogramas fiquem invisíveis para os usuários do "package". Finalmente, essa característica torna possível que o corpo de uma subrotina seja compilado em uma unidade diferente da de sua especificação.

A linguagem admite 3 modos para passagem de parâmetros. No 1º - IN - o parâmetro formal representa uma constante e só permite a leitura do valor do parâmetro real associado; o programador pode indicar valores default para os parâmetros desse modo, ou seja, valores que devem ser usados se os parâmetros reais correspondentes não forem especificados em alguma chamada do subprograma. No 2º modo - IN OUT - o parâmetro formal é uma variável, permitindo a leitura e atualização do valor do parâmetro real associado. No 3º modo - OUT - o parâmetro formal também representa uma variável, porém só é permitida a atualização do valor do parâmetro real correspondente. A linguagem não especifica que mecanismo deve ser adotado para passagem de parâmetros; um programa é errôneo se depender do mecanismo utilizado por determinada implementação.

Além da clássica associação de parâmetros por posição para chamadas de subprogramas, existe a possibilidade da associação ser nominal, na qual o programador especifica o nome do parâmetro formal e seu parâmetro atual correspondente. Essa possibilidade é bastante interessante para chamadas de subprogramas que tenham uma longa lista de parâmetros. Na realidade a linguagem também aceita que os 2 tipos de associação sejam usados em uma mesma chamada, desde que a partir da 1.^a associa

ção por nome as demais associações sejam desse tipo.

Certamente uma das principais características de ADA é a possibilidade de, em qualquer ponto de um programa, vários subprogramas declarados com o mesmo designador serem visíveis, sem que um esconda os demais, como acontece com variáveis. Essa característica é chamada de *sobreposição* ("overloading"), e também ocorre com literais. O principal motivo para incorporação da sobreposição em uma linguagem de programação é o aumento no grau de liberdade fornecida aos programadores para escolha de nomes de subprogramas. Ela também permite que se concentre a atenção mais na função dos mesmos. Dessa forma, pode-se ter várias rotinas de impressão com o mesmo nome (por exemplo, "PUT") tal que um imprima um inteiro quando chamada com um parâmetro desse tipo, outra imprima uma cadeia de caracteres quando seu parâmetro for desse tipo, etc ...

2.4. PACKAGES

Uma das características mais marcantes da linguagem ADA foi o emprego de pacotes ("packages"), que permitem o agrupamento de entidades logicamente relacionadas bem como de operações sobre as mesmas e a delimitação da quantidade de informação definida que será acessível pelo resto do programa. Isso permite que informações internas ao pacote sejam protegidas do uso acidental ou incorreto por entidades externas ao mesmo.

Dependendo da estrutura do pacote pode-se obter 3 finalidades distintas para o agrupamento de suas entidades: relacionamento de declarações, relacionamento de subprogramas ou obtenção de tipos de dados encapsulados, que serão descritos a seguir.

O 1º tipo de agrupamento é bastante útil para o relacionamento de entidades que sirvam de interface entre diversos módulos de um programa, tal como apresentado na figura II.1, retirada do "rationale" da linguagem.

```
PACKAGE WORK_DATA IS
  TYPE DAY IS (MON,TUE,WED,THU,FRI,SAT,SUN);
  TYPE DURATION IS DELTA 0.01 RANGE 0.0 .. 24.0;
  TYPE TIME_TABLE IS ARRAY (MON..SUN) OF DURATION;
  WORK_HOURS: TIME_TABLE;
  NORMAL_HOURS: CONSTANT TIME_TABLE:=
    (MON..THU => 8.25,FRI => 7.0,SAT|SUN => 0.0);
END WORK_DATA;
```

Figura II.1 - Exemplo de um pacote que agrupa um conjunto de declarações.

Qualquer módulo que use o um pacote desse tipo tem acesso à todas entidades declaradas no mesmo.

O 2º tipo de agrupamento fornecido por pacotes, permite a definição de operações que sejam acessíveis por qualquer módulo que o utilize. Esse tipo de pacote é dividido em 2 partes: na 1ª, a especificação, estão as declarações das entidades

des que podem ser acessadas fora do pacote; na 2^a, o corpo, estão as entidades acessíveis somente pelo próprio pacote. Note que a especificação e o corpo do pacote não precisam ser fornecidos em uma única unidade de compilação. A figura II.2 apresenta o exemplo (já clássico) da especificação de números racionais.

```

PACKAGE RATIONAL_NUMBERS IS
  TYPE RATIONAL IS
    RECORD
      NUMERATOR    : INTEGER;
      DENOMINATOR  : INTEGER;
    END RECORD;

  FUNCTION "=" (X,Y : RATIONAL) RETURN BOOLEAN;
  FUNCTION "+" (X,Y : RATIONAL) RETURN RATIONAL;
  FUNCTION "*" (X,Y : RATIONAL) RETURN RATIONAL;
END;

PACKAGE BODY RATIONAL_NUMBERS IS
  PROCEDURE SAME_DENOMINATOR (X,Y : IN OUT RATIONAL) IS
  BEGIN
    -- REDUZ X E Y PARA O MESMO DENOMINADOR
  END;

  FUNCTION "=" (X,Y : RATIONAL) RETURN BOOLEAN IS
    U,V : RATIONAL;
  BEGIN
    U := X;
    V := Y;
    SAME_DENOMINATOR (U,V);
    RETURN (U.NUMERATOR = V.NUMERATOR);
  END "=";

  FUNCTION "+" (X,Y : RATIONAL) RETURN RATIONAL IS ...END "+";
  FUNCTION "*" (X,Y : RATIONAL) RETURN RATIONAL IS ...END "*";

END RATIONAL_NUMBERS;

```

Figura II.2 - Exemplo de um pacote que especifica números racionais.

Um usuário desse pacote pode empregar as operações "=", "+" e "*" definidas na especificação do mesmo, porém não tem acesso à rotina "SAME_DENOMINATOR" declarada em seu corpo. Além disso ele pode declarar variáveis do tipo racional e empregar o fato dessas variáveis terem sido definidas como um record, para, por exemplo, definir o valor de uma dessas variáveis através de um agregado.

Resumindo, um pacote desse tipo permite que um usuário tenha acesso a todas entidades definidas em sua especificação (subprogramas, declarações de tipos e variáveis, etc) mas proíbe o acesso às entidades definidas em seu corpo.

Ora, a liberdade de manipular um elemento do tipo racional como um record dada a um usuário do pacote mostrado na figura II.2 pode levá-lo a operações inválidas sob o ponto de vista matemático, tais como atribuir a um objeto do tipo racional um valor que tenha o denominador nulo, colocando em uma situação incorreta o programa utilizado. Para evitar essa situação o autor de um pacote pode sub-dividir sua especificação em 2 partes. Na 1.^a parte pode ser definido um tipo sem que sua estrutura seja fornecida, através de declarações de tipos privados e limitados; na 2.^a parte, conhecida como parte privada do pacote, a estrutura desses tipos é então definida, porém usuários do pacote não tem acesso à mesma; dessa forma, o conhecimento da estrutura desses tipos só pode ser utilizado por subprogramas definidos no corpo do pacote. A figura II.3 procura exemplificar esse conceito.

```

PACKAGE SIMPLE_INPUT_OUTPUT IS

  TYPE FILE_NAME IS PRIVATE;
  NO_FILE : CONSTANT FILE_NAME;
  PROCEDURE CREATE RETURN FILE_NAME;
  PROCEDURE READ (ELEM : OUT INTEGER; F : IN FILE_NAME);
  PROCEDURE WRITE(ELEM : IN INTEGER; F : IN FILE_NAME);

PRIVATE
  TYPE FILE_NAME IS NEW INTEGER 0 .. 50;
  NO_FILE : CONSTANT FILE_NAME := 0;
END SIMPLE_INPUT_OUTPUT;

```

Figura II.3 - Exemplo de um pacote que empregue um tipo privado.

Dessa forma um usuário desse pacote pode simplesmente aplicar as operações definidas para tipos privados (basicamente atribuição, testes de pertinência ("*membership*") e de igualdade) sobre o tipo "FILE_NAME", ficando, porém, proibido de utilizar o fato dele estar definido como um tipo inteiro na parte privada, ou de que a constante NO_FILE ser representada pela constante inteira zero.

2.5. PROCESSAMENTO PARALELO

Devido ao grande número de sistemas em tempo real utilizados pelo Departamento de Defesa Americano, mecanismos de controle de atividades paralelas foram incorporados na linguagem ADA. Basicamente a linguagem apresenta estruturas do tipo task sincronizadas através de *rendez-vous*, caracterizando a comunicação de processos de uma forma síncrona e assimétrica. Além dos comandos básicos para comunicação entre tasks ("*entry-call*" e comando "*accept*"), a sincronização das mesmas também pode ser obtida pelos comandos "*select*", "*conditionall entry call*" ou "*timed entry call*". Tal como discutido para pacotes, os conceitos de modularidade e, de certa forma, de abstração, também se encontram presentes em tasks. A especificação de uma task possui a declaração das entradas ("*entries*") que definem os pontos para comunicação com outras tasks. O corpo de uma task define seu processamento.

2.6. TRATAMENTO DE EXCEÇÕES

Uma característica fundamental em uma linguagem que pretenda ser usada para obtenção de software confiável para aplicações em tempo real é o tratamento de situações de erro que, embora raras, podem acontecer em um sistema normal. De maneira geral, existem 2 possibilidades para o tratamento dessas situações em linguagens de programação. A 1.^a possibilidade considera esse tratamento como uma técnica de programação normal para situações que não caracterizem um erro, mas que sejam raras na prática. Dessa forma, após uma exceção ter ocorrido e sido tratada, o fluxo do programa deve retornar ao ponto seguinte de sua ocorrência. Um exemplo típico seria o tratamento de final de arquivo. A 2.^a possibilidade considera uma exceção como uma situação que de alguma forma caracterize uma situação de erro. Assim, quando uma exceção ocorrer em determinada subrotina o processamento da mesma deve ser encerrado, não importando as ações efetuadas pelo tratamento da exceção. Essa solução foi especificada pelo relatório STEELMAN, e a linguagem ADA a utiliza.

Um "*exception-handler*" é composto por uma sequência de comandos opcional que pode ocorrer em um comando bloco, em um corpo de subprograma, task ou pacote, no qual ficam especificadas as exceções tratadas com suas respectivas ações, como mostrado na figura II.4. Note que o comando "RAISE" é usado para provocar a ocorrência da exceção "SINGULAR" nas rotinas "Q" e "R".


```

PROCEDURE P IS
  SINGULAR : EXCEPTION;

  PROCEDURE Q IS
  BEGIN
    ...
    RAISE SINGULAR;
    ...
  END Q;

  PROCEDURE R IS
  BEGIN
    ... Q ...
    RAISE SINGULAR;
  EXCEPTION
    WHEN SINGULAR =>
      -- HANDLER # 1
  END R;

BEGIN -- P
  ... R ..; ... Q ...
EXCEPTION
  WHEN SINGULAR =>
    -- HANDLER # 2
END P;

```

Figura II.4 - Exemplo do emprego de exceções.

Uma questão bastante pertinente é a associação de exceções com seus respectivos "handlers". Note que um programa pode tratar uma exceção em mais de um lugar, como é o caso da figura II.4, na qual a exceção "singular" é tratada tanto pela rotina "R" quanto pela rotina "P". Por outro lado, uma exceção pode ser declarada e ocorrer sem que exista nenhum tratamento para a mesma. O tratamento também é diferenciado caso a exceção tenha ocorrido em um subprograma ou em uma task, em uma parte declarativa ou em uma sequência de comandos. Abaixo será apresentada a situação de uma exceção que tenha ocorrido na sequência de comandos de um subprograma, como mostrado no exemplo dado.

Duas situações podem existir. Na primeira, o corpo do subprograma no qual a exceção ocorreu possui um handler para a mesma. Nesse caso o fluxo do processamento é desviado para o handler e após o final de seu processamento a execução do subprograma é encerrada. Dessa forma, os comandos compreendidos entre o ponto no qual a exceção ocorreu e o final do subprograma não são executados. É essa situação no caso da exceção da figura II.4 ocorrer durante a execução da rotina "R".

Por outro lado, se o corpo do subprograma não possuir um handler para a exceção sua execução é abandonada e a exceção é automaticamente levantada no ponto de sua chamada. Esse processo é chamado de *propagação* de exceções. No exemplo dado, se a exceção ocorrer na rotina "Q", ela é tratada pelo handler número 1 caso a rotina tenha sido chamada por "R" e pelo handler número 2 se tiver sido chamada diretamente por "P".

O tratamento de exceções em tasks é bastante mais complexo devido à própria natureza de ambas. Para finalizar, deve ser acrescentado que as exceções pre-definidas da linguagem ("*constraint-error*", "*numeric-error*", "*program-error*", "*storage-error*" e "*tasking-error*") são tratadas da mesma forma apresentada.

2.7. UNIDADES GENÉRICAS

O conceito de unidades genéricas incorporado em ADA pode ser considerado como uma expansão do mecanismo de parametrização oferecido por subprogramas de linguagens tradicionais. Porém, embora a parametrização oferecida por subprogramas se restrinja a variáveis, unidades genéricas permitem a parametrização de tipos e mesmo de subprogramas. Uma implementação em ADA do tipo *pilha* pode ser obtida através de um pacote que forneça o tipo de elementos da pilha e as operações usuais que atuam sobre essa estrutura. Porém, se um determinado programa precisar manipular pilhas de elementos de tipos distintos, as regras da linguagem obrigam a definição e implementação de todas as operações necessárias para cada tipo empregado. Note que, entretanto, essas operações são intrinsecamente independentes desses tipos. A linguagem ADA permite a solução desse problema, por exemplo, pela definição de um pacote genérico no qual seriam especificados um tipo (genérico) e as operações desejadas, que seriam definidas em termos do tipo formal fornecido. A figura II.5 ilustra uma especificação possível para um pacote desse tipo.

```

GENERIC
  SIZE : POSITIVE;
  TYPE ITEM IS PRIVATE;
PACKAGE STACK IS
  PROCEDURE PUSH (E : IN ITEM);
  PROCEDURE POP  (E : OUT ITEM);
  OVERFLOW, UNDERFLOW : EXCEPTION;
END STACK;
```

Figura II.5 - Exemplo da especificação de um pacote genérico para definição do tipo "pilha".

A definição do pacote poderia ser a fornecida na próxima figura.

```

PACKAGE BODY STACK IS

  TYPE TABLE IS ARRAY (POSITIVE RANGE <>) OF ITEM;
  SPACE : TABLE (1 .. SIZE);
  INDEX : NATURAL := 0;

  PROCEDURE PUSH (E : IN ITEM) IS
  BEGIN
    IF INDEX >= SIZE THEN
      RAISE OVERFLOW;
    END IF;
    INDEX := INDEX + 1;
    SPACE (INDEX) := E;
  END PUSH;

  PROCEDURE POP (E : OUT ITEM) IS
  BEGIN
    IF INDEX = 0 THEN
      RAISE UNDERFLOW;
    END IF;
    E := SPACE (INDEX);
    INDEX := INDEX - 1;
  END POP;

END STACK;

```

Figura II.6 - Definição de um pacote genérico que implemente o tipo "pilha".

Um usuário poderia criar instâncias desse pacote e utilizar suas operações da forma definida na figura II.7.

```

PACKAGE STACK_INT IS NEW STACK(SIZE => 200, ITEM => INTEGER);
PACKAGE STACK_BOOL IS NEW STACK (100, BOOLEAN);

STACK_INT . PUSH (N);
STACK_BOOL . POP (TRUE);

```

Figura II.7 - Exemplo da instanciação do pacote fornecido.

A expectativa do emprego de unidades genéricas pelo grupo que definiu a linguagem é tal que uma pequena equipe de programadores experientes e conhecedores dos mecanismos (complexos) para definição de unidades genéricas seja responsável pela definição e implementação das mesmas, sempre tendo em vista as necessidades dos usuários. Estes, por sua vez, só precisam conhecer a operação de instanciação de unidades genéricas (uma tarefa relativamente simples se comparada com a de suas definições), além, é claro, das especificações das unidades genéricas que pretendem utilizar na instanciação.

2.8. ASPECTOS DEPENDENTES DE MÁQUINA

Como já foi apresentado, ADA é uma linguagem de alto nível sendo que dois dos principais objetivos de sua definição foram a portabilidade da linguagem e a existência de mecanismos que permitam a obtenção de software altamente confiável. Uma maneira de se atender esses objetivos seria mantendo a linguagem afastada das características das máquinas nas quais programadas escritos em ADA rodariam. Porém, isso seria inaceitável para uma linguagem voltada para programas de sistemas em tempo real. Nessas aplicações não somente a eficiência é um aspecto importante (e na realidade não existe nada que indique que programas ADA sejam necessariamente ineficientes), porém a capacidade de acessar características de hardware, tal como endereço de portas de entrada e saída, níveis de interrupções e endereço das rotinas correspondentes, etc..., também é fundamental. ADA procura resolver essa aparente contradição através de definição de características dependentes de máquina em pontos específicos do programa.

Dessa forma a linguagem permite que cláusulas de representação sejam definidas para os tipos de enumeração e record, conforme mostrado na figura II.8, retirada do manual de referência da linguagem.

```

TYPE MIX_CODE IS (ADD, SUB, MUL, LDA, STA, STZ);

FOR MIX_CODE USE
  (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24,
   STZ => 33);

WORD : CONSTANT := 4  -- 4 BYTES POR PALAVRA

TYPE PROGRAM_STATUS_WORD IS
  RECORD
    SYSTEM_MASK      : BYTE_MASK;
    PROTECTION_KEY   : INTEGER RANGE 0 ... 3;
    MACHINE_STATE    : STATE_MASK;
    INTERRUPT_CAUSE  : INTERRUPTION_CODE;
    ILC              : INTEGER RANGE 0 .. 3;
    CC               : INTEGER RANGE 0 .. 3;
    PROGRAM_MASK     : MODE_MASK;
    INST_ADDRESS     : ADDRESS;
  END RECORD;

FOR PROGRAM_STATUS_WORD USE
  RECORD AT MOD 8;
    SYSTEM_MASK      at 0*WORD RANGE 0  .. 7;
    PROTECTION_KEY   at 0*WORD RANGE 10 .. 11;
    MACHINE_STATE    at 0*WORD RANGE 12 .. 15;
    INTERRUPT_CAUSE  at 0*WORD RANGE 16 .. 31;
    ILC              at 1*WORD RANGE 0  .. 1;
    CC               at 1*WORD RANGE 2  .. 3;
    PROGRAM_MASK     at 1*WORD RANGE 4  .. 7;
    INST_ADDRESS     at 1*WORD RANGE 8  .. 31;
  END RECORD;

```

Figura II.8 - Exemplos de cláusulas de representação dos tipos de enumeração e record.

A cláusula de representação de um tipo de enumeração especifica os códigos internos dos literais que ocorrem na mesma; a cláusula de representação de um tipo record indica a representação interna desse tipo, ou seja, a ordem, posição e tamanho de seus componentes. Note que essas cláusulas dizem respeito a tipos, e não a objetos específicos.

Uma cláusula de endereço permite que algumas entidades-objetos, subprogramas, pacotes, task, ou entradas simples ("*single entry*") tenham seus endereços especificados em um programa. De particular importância é a especificação de endereços de entradas, pois permite que essas sejam associadas a dispositivos que podem provocar uma interrupção.

A linguagem também permite inserção de código de máquina através do comando "CODE", só permitidos em rotinas; além disso, se uma rotina possuir um comando code, ela só pode ter comandos desse tipo. Por outro lado, também é permitido que programas escritos em outras linguagens sejam chamados por programas ADA.

2.9. ENTRADA E SAÍDA

ADA prove operações de entrada e saída através de pacotes predefinidos. Os pacotes genéricos "SEQUENTIAL_IO" e "DIRECT_IO" definem operações de entrada e saída aplicáveis a arquivos contendo elementos de determinado tipo. Operações adicionais para entrada e saída de texto são fornecidas no pacote "TEXT_IO". O pacote "IO_EXCEPTION" define as exceções necessárias pelos pacotes citados acima. Por último, o pacote "LOW_LEVEL_IO" fornece o controle necessário para tratamento de periféricos.

2.10. AMBIENTE DE PROGRAMAÇÃO ADA

As principais características do ambiente de suporte à programação proposta por STONEMAN são as seguintes:

- . Suporte durante o ciclo de vida completo do software;
- . Ambiente do tipo aberto;
- . Suporte para linguagem inteira;
- . Controle de configuração;
- . Suporte para trabalho em equipe;
- . Portabilidade;
- . Uso de um banco de dados para gerenciar projetos.

A seguir serão apresentados alguns aspectos de cada característica.

O ambiente de programação deve ser capaz de fornecer informações e mecanismos que permitam o acompanhamento de um projeto durante todo seu ciclo de vida, desde as especificações iniciais até a manutenção do mesmo. O gerenciamento dessas informações deve ser feito por um banco de dados incorporado ao ambiente, e devem existir ferramentas apropriadas para manipulação dessas informações para cada fase do projeto. De particular importância nesse aspecto é a questão de configuração de controle.

A qualquer momento um usuário deve ser capaz de obter todos os dados relativos a um determinado programa, tais como conjunto dos módulos que o compõe, texto fonte desses módulos, documentação correspondente, conjunto de testes usados na fase de depuração do programa, etc. Por outro lado se houver necessidade de se modificar algum módulo, o ambiente deve ser capaz de fornecer o respectivo texto fonte e uma lista com os demais módulos que utilizam o módulo modificado. Dessa forma espera-se automatizar o processo de manutenção de software, diminuir o tempo envolvido no mesmo e minimizar a possibilidade de se inserir erros espúrios durante este processo; o banco de dados do sistema deve ser o gerenciador de todas essas informações.

Existem 2 maneira básicas de se definir um ambiente

de programação: a primeira é através da definição de um número fixo de ferramentas que o ambiente suporta; esse número deve ser suficiente para oferecer as condições mínimas de utilização pela maior parte da comunidade de usuários do sistema, exigindo um cuidado muito grande durante a fase de definição dessas ferramentas. Isso caracteriza o chamado ambiente de programação fechado, que não permite que o usuário defina/implemente novas ferramentas de acordo com suas necessidades. Por outro lado, a proposta contida no STONEMAN é no sentido que o ambiente tenha um mínimo de ferramentas necessárias à comunidade de usuários, porém que forneça condições para que cada usuário possa definir e implementar novas ferramentas de acordo com a evolução de suas necessidades, caracterizando o chamado ambiente de programação aberto. Em um ambiente desse tipo, tão importante quanto o projeto das novas ferramentas é o projeto das interfaces das mesmas, que deve ser bem feito de modo a manter o maior grau de portabilidade possível. Por outro lado, um projeto cuidadoso das interfaces permite que uma nova ferramenta possa interagir com as demais ferramentas existentes no ambiente (por exemplo, um formatador de programas fonte pode ser definido de modo a utilizar a saída do front-end do compilador).

Um conceito fundamental levantado pelo STONEMAN é a portabilidade que o ambiente deve apresentar e oferecer a seus usuários. Essa portabilidade se reflete na portabilidade de ferramenta (a capacidade de uma ferramenta ser transportada para um ambiente compatível ao ambiente em que ela foi desenvolvida), portabilidade de projetos (a capacidade de um projeto ser transportado de um hospedeiro para outro), a portabilidade de programadores (capacidade de um programador trabalhar em outros projetos ou com outro ambiente compatível sem necessidade de treinamento), a capacidade do ambiente suportar uma nova máquina alvo é a possibilidade de se mover o próprio ambiente para nova máquina hospedeira.

De forma a facilitar a obtenção desse grau de portabilidade, é sugerida a divisão hierárquica do ambiente em 3 níveis.

O 1º nível, chamado KAPSE - KERNEL ADA PROGRAMMING SUPPORT ENVIROMENT - implementa as funções do sistema dependen

tes de máquina; ele compõe o núcleo básico necessário aos programas, sendo fundamental para se obter o grau de portabilidade desejável. Dessa forma, qualquer programa que faça referências externas somente ao KAPSE pode ser transportado para qualquer sistema que forneça o mesmo KAPSE. Por outro lado, se todas as ferramentas forem escritas em ADA e não fizerem uso de aspectos dependentes de máquina que a linguagem oferece, (tais como inserção de código de máquina) o próprio ambiente pode migrar para outro hospedeiro sem grandes alterações. Para facilitar o transporte do KAPSE para outras máquinas, ele deve ser constituído por um número pequeno de módulos. O KAPSE que está sendo desenvolvido para o ALS (ADA Language System) pela Softech com o financiamento da US ARMY CECOM, contém 43 rotinas.

O 2º nível, chamado MAPSE - Minimal ADA Programming Support Environment - contém o conjunto de ferramentas mínimo que permite à comunidade de usuários trabalhar confortavelmente durante todo o ciclo de vida de um sistema. O mesmo projeto citado anteriormente contém 17 módulos para o MAPSE.

Como o ambiente deve ser do tipo aberto, novos módulos podem ser acrescentados a qualquer momento no mesmo, ampliando dessa forma o 3º nível do ambiente, chamado APSE. No caso de nenhum módulo ser acrescentado ao APSE este se confunde com o próprio MAPSE.

Por tudo que já foi mencionado, fica clara a importância do banco de dados no funcionamento do ambiente, uma vez que ele é responsável pelo gerenciamento de todas as informações necessárias durante todo o ciclo de vida de um projeto. Por exemplo, ele deve manter as informações que garantam a integridade dos diversos módulos do sistema. Dessa forma, o banco de dados é de particular importância para qualquer ferramenta de controle de configuração. Por outro lado, o banco de dados deve ser capaz de permitir a compilação em separado de módulos escritos em ADA (subrotinas, "packages", etc), possibilitando a recriação de contextos necessários para o nível de controle de tipo definido pela linguagem.

Como um usuário pode incorporar novas ferramentas no APSE, o banco de dados deve ser capaz de identificar os novos tipos de entidades criadas e de tratar esses tipos normalmente.

Paralelamente, o banco de dados deve fornecer um mecanismo efi
caz de controle de acesso aos módulos criados.

CAPÍTULO III

DIANA

3.1. CONSIDERAÇÕES GERAIS

DIANA é uma forma intermediária projetada para servir fundamentalmente como comunicação entre o front-end e o back-end de compiladores ADA. Paralelamente, em um ambiente de programação como o esperado para ADA, deve existir um bom número de ferramentas (formatadores, depuradores simbólicos, editores orientados pela sintaxe, etc...) que podem se beneficiar pelo emprego de uma entrada mais elaborada do que o texto fonte do programa em análise; DIANA pode servir como entrada para essas ferramentas.

DIANA está baseada em duas propostas de forma intermediária de programas ADA: AIDA [DAUSMANN et al (12)] desenvolvida pelo grupo da Universidade de Karlsruhe e TCOL [BROSGOL et al. (13)], desenvolvida pelo grupo da Universidade de Carnegie-Mellon. Em sua primeira especificação, publicada como um manual de referência por GOOS e WULF (3), DIANA representa a definição da linguagem conhecida como ADA-80 [DOD (01)]. Devido à sua potencialidade, vários grupos se interessaram em usá-la em seus projetos, ficando patentes as vantagens de uma forma intermediária padrão e, portanto, portátil, em vários ambientes de programação ADA. Devido a pequenas falhas em sua definição inicial e à nova versão da linguagem-ADA-82 [DOD (02)] o departamento de defesa americano contratou "TARTAN LABORATORIES" para efetuar as modificações necessárias e centralizar o controle e divulgação da nova versão de DIANA [EVANS E BUTLER (4)]. Devido ao grande poderio de DIANA, bem como à sua grande aceitação por vários implementadores da linguagem, decidimos usá-la em nosso projeto.

Para definição de DIANA foi usado o modelo de árvores de atributos [McKEEMAN (24)]. Uma grande preocupação na definição de DIANA foi apresenta-la como um tipo abstrato de dados,

de modo a não impor nenhuma restrição à sua implementação. Porém, devido às características do objeto sendo modelado, a interpretação de DIANA como uma árvore é bastante natural, e longe de desestimulá-la, será incentivada nesse trabalho. DIANA pode ser considerada em 2 níveis de abstração distintos: o primeiro representa a árvore sintática abstrata construída antes da análise semântica; essa árvore será conhecida como "*Árvore DIANA Sintática*" e possui somente atributos léxicos e sintáticos. O segundo nível de abstração representa o resultado da incorporação das informações provenientes da análise semântica estática na árvore DIANA sintática, na qual ficam incorporados atributos semânticos e que passa a ser chamada de "*Árvore DIANA*". Devido aos atributos semânticos a árvore DIANA pode ser considerada mais propriamente como um grafo acíclico.

Note que, da forma que foi definida, DIANA não contém atributos resultantes da análise semântica dinâmica e da otimização ou geração de código. Porém, fica aberta a possibilidade de se estender a linguagem para permitir a inclusão de informações provenientes de outros tipos de processamento. Isso obviamente representa uma grande vantagem pois, por exemplo, diferentes algoritmos de otimização podem necessitar ou produzir diferentes tipos de informações. Assim sendo, determinada implementação pode acrescentar novos atributos de acordo com suas necessidades. Como será visto ao longo desse trabalho, foram definidos novos atributos semânticos para utilização pelo analisador semântico de modo a tornar seus algoritmos mais simples. Como esses atributos não são necessários às outras ferramentas do ambiente de programação, espera-se que o módulo que guarde a árvore DIANA na biblioteca consiga filtrá-los. É claro que a criação e o emprego de novos atributos pode comprometer a padronização e portabilidade de DIANA. Tentando minimizar esse perigo o manual de referência procura caracterizar o que seja uma aplicação que produza DIANA e uma aplicação que consuma DIANA: a primeira deve produzir uma saída que inclua toda informação tal como definida no manual de referência, ficando livre para adicionar novos atributos de acordo com seus objetivos ou finalidades. Um consumidor de DIANA não deve depender de outros atributos além dos definidos no manual de referência, podendo

aproveitá-los, caso os atributos desejados existam. Uma ferramenta do ambiente de programação ADA projetada para processar árvores geradas por vários sistemas deve observar essas definições. Note que essas características dizem respeito somente às interfaces das ferramentas. Por exemplo, o analisador semântico não pode prescindir do uso de alguns dos novos atributos definidos para o seu funcionamento; porém esses atributos são criados pelo próprio analisador semântico e não precisam estar presentes na árvore que é guardada na biblioteca, de modo que sua interface com outros módulos fica representada pela própria definição de DIANA.

3.2. PRINCÍPIOS GERAIS

Nessa seção serão listados, resumidamente, os princípios básicos que nortearam o projeto de DIANA.

- I) DIANA é independente de representação. Como foi visto, DIANA foi apresentada como um tipo abstrato de dados, de modo a não impor nenhuma restrição à sua implementação. Além disso, nos pontos onde são necessárias informações específicas de cada implementação (como representação de valores na máquina alvo) é usado outro tipo abstrato (ou privado, no sentido definido por ADA).
- II) DIANA é baseada na definição formal de ADA [INRIA (14)], especialmente no que diz respeito à árvore sintática DIANA. Note que como a definição formal não foi atualizada, a versão de DIANA teve que se desviar um pouco da mesma.
- III) Regularidade é uma das principais características de DIANA. Nota-se esse princípio especialmente na descrição e notação da linguagem, o que facilita bastante seu entendimento.
- IV) DIANA deve ser eficientemente implementável. Infelizmente os resultados iniciais não são muito animadores, principalmente no que diz respeito ao uso de memória [PAYTON (15), QUINN (16)]. De modo geral, essas referências tratam de implementações piloto, nas quais a eficiência não era o objetivo principal, mas sim o estudo de técnicas de implementação da linguagem.
- V) A estrutura inicial do programa fonte não é destruída; os atributos léxicos incorporados em DIANA permitem a reconstrução do fonte a qualquer momento.

3.3. NOTAÇÃO

A notação utilizada no manual de referência para descrição de DIANA é conhecida como IDL, descrita por Nestor (17), tendo sido desenvolvida pela Universidade de Carnegie-Mellon em seu projeto de desenvolvimento de métodos para produção automática de compiladores, onde foi utilizada na descrição de TCOL. Serão apresentadas aqui somente as principais características dessa notação.

O conjunto de árvores usado para se modelar DIANA pode ser considerado como uma linguagem, na qual as sentenças terminais representam árvores em vez de cadeias de caracteres, como demonstra o exemplo da figura III.1.

```
TYPE FRUTAS IS (BANANA, MAÇA, LARANJA);
```

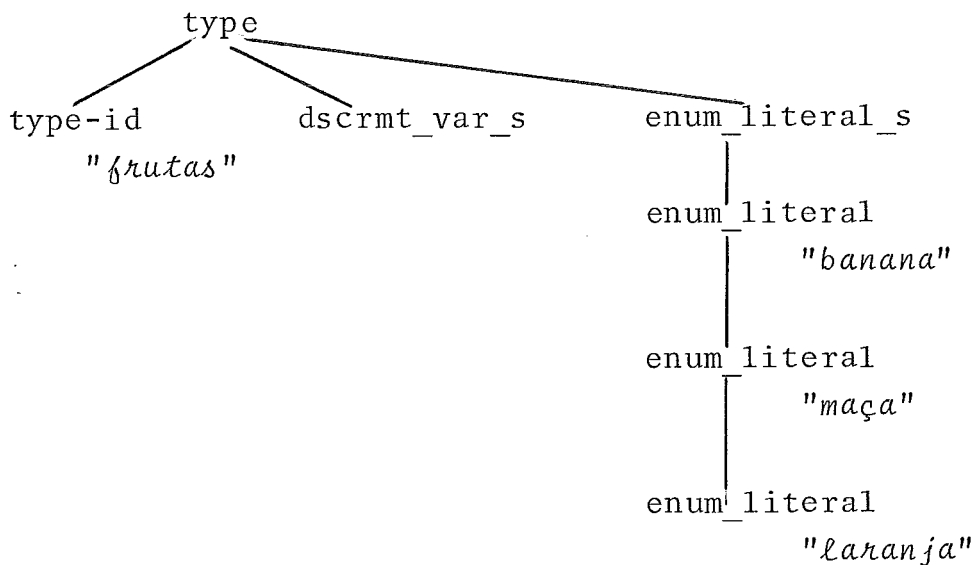


Figura III.1 - Trecho de um programa ADA e representação DIANA correspondente (só são apresentados os atributos léxicos e sintáticos).

Para definição dessa linguagem pode-se usar uma notação similar à BNF, como mostrado na próxima figura.

```
EXP:= folha | árvore;
```

Figura III.2 - Uma produção em IDL

Tal como em BNF, essa produção pode ser lida da seguinte forma: "A entidade "EXP" é definida pela entidade "folha" ou pela entidade "ÁRVORE". A entidade "EXP", que representa um não-terminal na comparação com BNF, é chamada de *classe*; como convenção para esse trabalho, classes serão sempre representadas por letras maiúsculas. As entidades "folha" e "árvore" representam terminais em BNF e são chamadas de *nós* em IDL, sendo sempre representadas por minúsculas nesse trabalho. Tal como em BNF, uma sentença terminal em IDL deve conter somente símbolos terminais (nós).

Como um nó pode conter informações não sintáticas (no caso de DIANA, informações léxicas e semânticas) deve haver uma maneira de se relacionar essas informações aos respectivos nós. Esse mecanismo é implementado por meio de atributos; dessa forma, a definição de um nó deve representar os atributos presentes nele, bem como seus tipos. Novamente é usada uma notação semelhante à BNF para descrição dos atributos. A próxima figura apresenta os atributos do nó "árvore" do exemplo anterior:

```

árvore => op: OPERADOR,
          esq:EXP,
          dir:EXP;

```

Figura III.3 - Atributos do nó "ÁRVORE"

Note que a notação é ligeiramente diferente da notação usada para definição de classes de modo a evitar confusões (o símbolo "::<=" é trocado pelo símbolo "=>"). Nesse ponto, duas diferenças são marcantes em relação à notação BNF: em primeiro lugar a ordem dos atributos não é relevante; em segundo lugar, múltiplas definições para um único nó simplesmente acrescentam novos atributos ao nó. A próxima figura apresenta os mesmos atributos do nó "árvore" definidos de uma forma diferente mas com o mesmo significado.

```

    árvore => op: OPERADOR;
    árvore => dir: EXP,
              esq: EXP;

```

Figura III.4 - Outra definição para os atributos do nó "árvore"

Em IDL o atributo de um nó pode ter um tipo básico ou um tipo privado. Os tipos básicos são os tipos booleano, inteiro, real e cadeia de caracteres. Além desses, uma classe também pode ser considerada como tipo base podendo, portanto, representar o tipo de um atributo. Um tipo privado é tal que sua estrutura pode (ou deve) depender da implementação, ficando sua especificação com um nível maior de detalhes. Por exemplo, em DIANA vários nós possuem o atributo "*source-position*", que indica em que ponto a estrutura léxica associada ao nó ocorreu no texto fonte. De forma a fornecer maior liberdade ao implementador para guardar essa informação, o tipo desse atributo foi deixado como privado. O mesmo acontece com o atributo que recebe o valor de expressões estáticas.

A notação também contém um construtor de sequência de classe, definido informalmente no manual de referência como um tipo. Dessa forma, a notação "*seq of T*" representa uma sequência de objetos do tipo T.

Algumas convenções adicionais foram adotadas para definição de DIANA. Uma classe ou um nó cujos nomes terminem pelos sufixos "*_S*" ou "*_s*" representam uma sequência do que vem antes do sufixo, conforme a figura III.5.

```

    ID_S ::= id_s;
    id_s => as_list: seq of ID;

```

Figura III.5 - Exemplo do construtor seq.

Uma classe que termine pelo sufixo '*_VOID*' sempre tem uma definição do tipo:

```

EXP_VOID ::= EXP |
           void;

```

Figura III.6 - Exemplo do nó "void"

O nó "void" não possui nenhum atributo.

Finalmente, existem 4 tipos de atributos apresentados pelo manual de DIANA: *estruturais*, representados pelo prefixo "as_"; *lêxicos*, representados pelo prefixo "lx_"; *semânticos*, representados pelo prefixo "sm_" e atributos de código, representados pelo prefixo "cd_". De forma semelhante, os atributos que foram criados nesse trabalho para serem usados pela análise semântica estática são sempre representados pelo prefixo "se_". Os atributos léxicos representam as informações necessárias para reconstrução do texto fonte, tais como representação de identificadores, comentários, posição no texto fonte, etc.. Os atributos estruturais definem a árvore sintática abstrata de um programa ADA, ao passo que os atributos semânticos contêm as informações obtidas pela análise semântica estática. Finalmente, os atributos de código representam informações provenientes de cláusulas de especificação de representação e são usados somente pelo gerador de código.

Para finalizar essa seção de notação serão apresentadas, como um exemplo, as produções em IDL de DIANA que representam a declaração de um tipo de enumeração. O nó 'type' será usado como nó inicial.

```

type => as_id: ID, --sempre 'type_id'
      as_dscrmt_var_s: DSCRMT_VAR_S,
      as_type_spec: TYPE_SPEC;
type => lx_srcpos: source_position,
      lx_comments: comments;

ID ::= DEF_ID;
DEF_ID ::= type_id;

type_id => sm_type_spec: TYPE_SPEC,
        sm_first: DEF_OCCURRENCE;
type_id => lx_srcpos: source_position,
        lx_comments: comments,
        lx_symrep: symbol_rep;

TYPE_SPEC ::= enum_literal_s;
enum_literal_s => as_list: seq of ENUM_LITERAL;
enum_literal_s => lx_srcpos: source_position,
                lx_comments: comments;
enum_literal_s => sm_size: exp_void;
enum_literal_s => cd_impl_size: integer;
ENUM_LITERAL ::= enum_id;
enum_id => lx_srcpos: source_position,
          lx_comments: comments,
          lx_symrep: symbol_rep;
enum_id => sm_obj_type: TYPE_SPEC, --referencia o nó
          --enum_literal_s
          sm_pos: integer, --posição consecutiva
          sm_rep: integer; --valor fornecido pelo usuário

DEF_OCCURRENCE ::= DEF_ID;

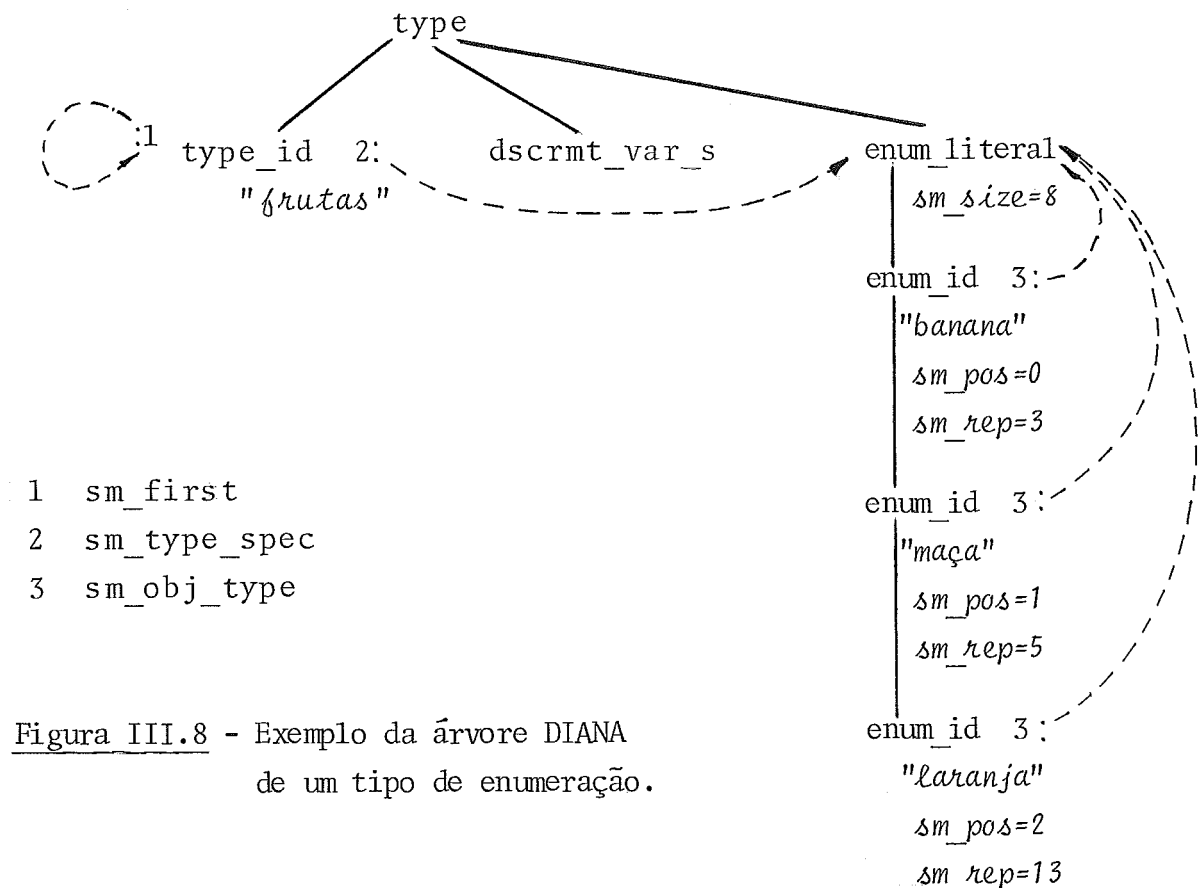
```

Figura III.7 - Exemplo de produção DIANA

Os comentários que aparecem no exemplo (texto após os caracteres '--') foram extraídos do manual; eles são úteis para dar algumas pistas sobre o significado dos atributos semânticos empregados (normalmente, junto com o bom senso, as uni

cas pistas disponíveis). Todos os atributos léxicos são privados, podendo variar entre as implementações. A classe "DSCRMT_VAR_S" serve para representar a parte discriminante de um tipo, sempre inexistente em um tipo de enumeração. O atributo "sm_first" do nó "type_id" serve para referenciar a primeira ocorrência de um identificador com mais de uma definição como, por exemplo, a declaração de um tipo incompleto. O atributo "sm_size" do nó "enum_literal_s" denota a expressão fornecida na especificação de representação, se essa existir; caso contrário, aponta para um nó "void". O atributo "sm_pos" dos nós "enum_id" representa a posição do nó em relação aos outros literais de enumeração; o atributo "sm_rep" contém o valor fornecido pelo usuário em uma cláusula de representação para o nó em questão. A próxima figura procura explicitar mais o exemplo dado.

```
TYPE FRUTAS IS (BANANA, MAÇA, LARANJA);
FOR FRUTAS'SIZE USE 8;
FOR FRUTAS USE (BANANA => 3, MAÇA => 5, LARANJA => 13);
```



Note que na figura anterior os atributos semânticos que se referenciam a outros nós da árvore foram representados por linhas pontilhadas e tiveram seus nomes explícitos. Os atributos estruturais foram representados por linhas cheias e não tiveram seus nomes explicitamente colocados na figura. Dos atributos léxicos, somente o que representa identificadores foi apresentado. Essas convenções serão obedecidas até o final do trabalho. O apêndice 1 apresenta um sumário de DIANA, com a especificação de todas as classes e nós em IDL.

3.4. ALGUMAS CARACTERÍSTICAS DE DIANA

O capítulo 3 do manual de referência de DIANA fornece uma explicação para os pontos mais obscuros de seu projeto e procura tornar mais clara as soluções adotadas. Embora não caiba aqui um levantamento completo desses pontos, a apresentação de alguns deles é importante para se entender algumas sutilezas necessárias na definição do analisador semântico.

3.4.1. AMBIGUIDADES NA GRAMÁTICA DE ADA

Algumas ambiguidades na gramática de ADA não podem ser resolvidas simplesmente no nível sintático, sendo necessárias informações semânticas para tal. O exemplo mais típico é o de construções indexadas: componentes indexados, chamadas de subprogramas, conversões de tipo e "slices". Nesse caso específico é gerado um nó genérico pelo analisador sintático; a sub-árvore correspondente deve ser modificada durante a análise semântica, de acordo com as informações que vão sendo obtidas. A interface completa entre o analisador sintático e semântico do nosso projeto, com as demais ambiguidades relacionadas, será definida em CHAVES (8). Não esperamos muita dificuldade para se alterar o módulo de análise semântica de forma que possa operar corretamente com a interface.

3.4.2. ASPECTOS DE COMPILAÇÃO EM SEPARADO

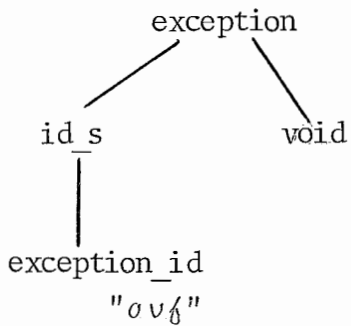
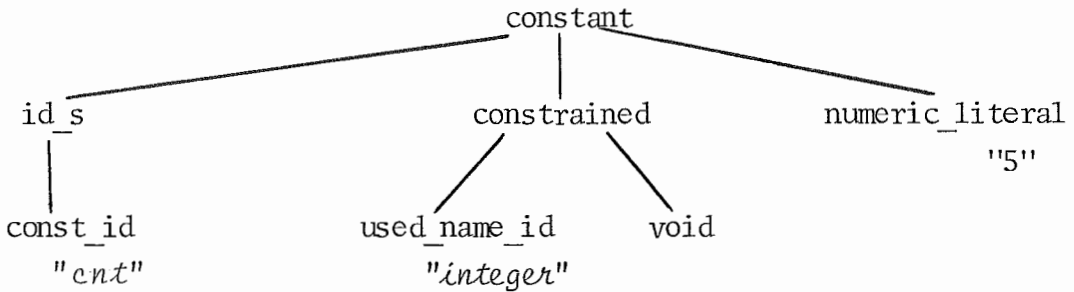
A incorporação do conceito de compilação em separado deve influenciar o projeto de qualquer linguagem intermediária projetada para ADA, uma vez que o front-end do compilador deve ser capaz de acessar a representação de outras unidades de compilação. A definição de DIANA não impõe nenhuma dificuldade nesse aspecto, e um sistema do nível do definido em RANGEL et al. (18) e CHAVES (8), deve implementar facilmente os mecanismos necessários.

3.4.3. MANIPULAÇÃO DE NOMES

Toda entidade em ADA é introduzida através de uma declaração (explícita ou não) que associa um identificador com a entidade. As sub-árvores DIANA que representam essas declarações fazem o papel da tabela de símbolos nos compiladores convencionais. Dessa forma, DIANA diferencia a definição do uso de identificadores. A sub-árvore de uma declaração possui um filho que representa a sequência de identificadores das novas entidades declaradas. Para cada tipo de entidade definida existe um tipo de nó específico com um conjunto de atributos necessários para o nó em questão. Esses nós são definidos pela classe "DEF_ID". A figura III.9 apresenta um exemplo de declaração de constantes e exceções.

CNT: CONSTANT INTEGER:= 5;

OVF: EXCEPTION



```

const_id => sm_address: EXP_VOID,
           sm_obj_type: TYPE_SPEC,
           sm_obj_def: OBJECT_DEF,
           sm_first: DEF_OCCURRENCE;
  
```

```

exception_id => sm_exception_def: EXCEPTION_DEF;
  
```

Figura III.9 - Exemplo de declarações de constantes e exceções. Note que o nó "*const_id*" possui 4 atributos semânticos, ao passo que "*exception_id*" somente 1.

Por outro lado, toda vez que um identificador aparece fora de uma declaração ele representa o uso de uma entidade já declarada. DIANA diferencia essas ocorrências em 3 tipos: se o identificador se referenciar a um objeto ele é representado por um nó do tipo `"used_object_id"`; caso contrário ele é representado por um nó do tipo `"used_name_id"` ou `"used_blt_n_id"`. Esses 3 nós possuem atributos semânticos `"sm_defn"` ou `"sm_operator"` que se referenciam à definição da entidade, na qual todas as informações sobre a mesma podem ser obtidas, conforme apresentado na próxima figura.

```
TYPE FRUTAS IS (BANANA, MAÇA, LARANJA);
FRT: FRUTAS := BANANA;
```

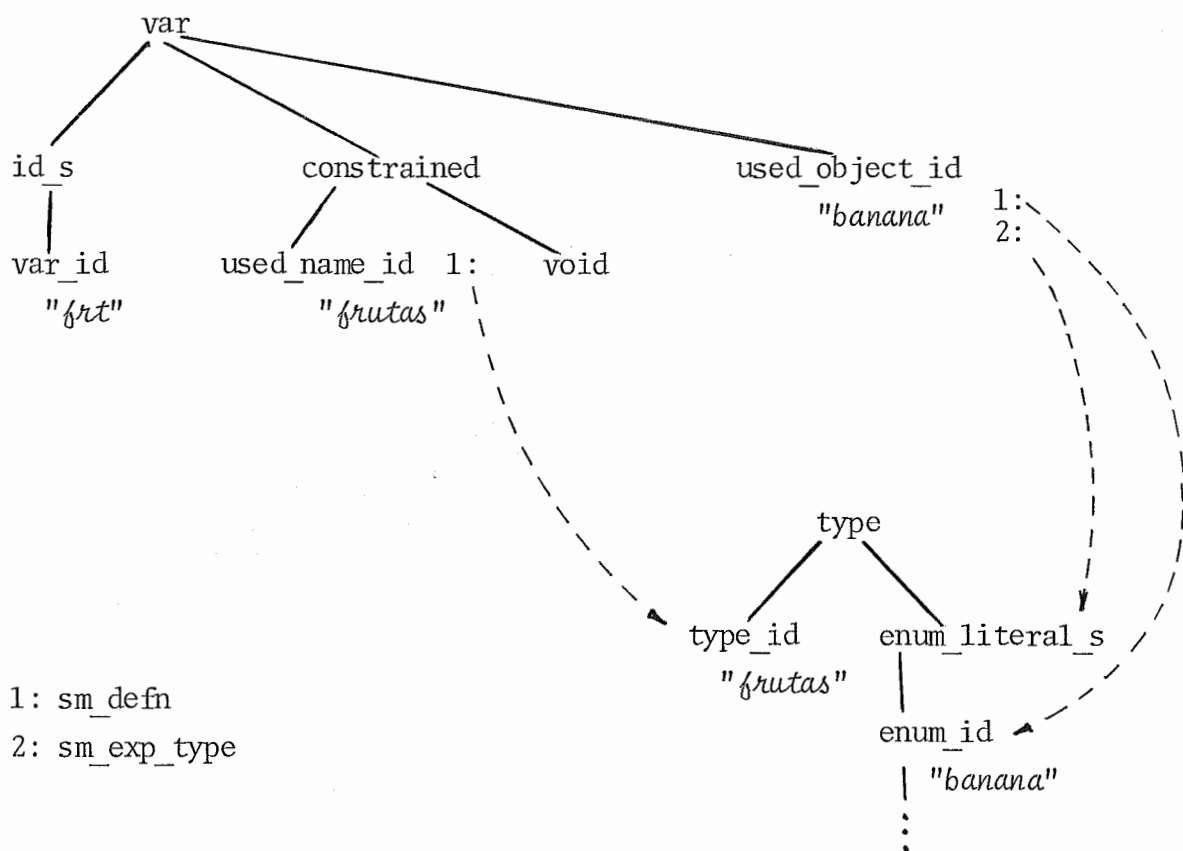


Figura III.10 - O exemplo de uso dos nós `"used_name_id"` e `"used_object_id"`. Note que somente os atributos semânticos desses nós foram representados para se simplificar a figura.

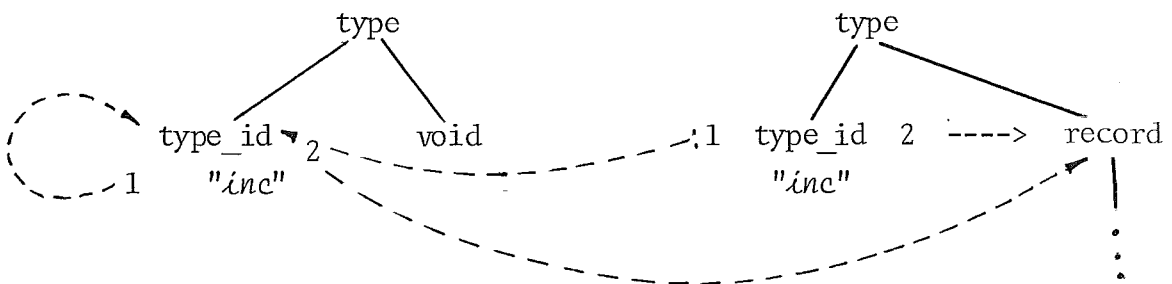
3.4.4. DEFINIÇÃO MÚLTIPLAS DE IDENTIFICADORES

Em ADA, um identificador pode ter sua definição fornecida em vários pontos (por exemplo, tipos incompletos ou constantes postergadas) ou mesmo duplicadas (especificação de subprogramas). DIANA resolve isso criando um nó da classe "DEF_ID" para cada definição da entidade; esses nós possuem um atributo ("sm_first") que se referencia à primeira ocorrência do identificador. Dessa forma, a primeira definição da entidade é considerada como a definição da entidade e todas as referências à entidade são consideradas como referências à sua primeira definição. A figura III.11 ilustra essa solução para declaração incompleta de tipo.

```

TYPE INC;
---
TYPE INC IS
  RECORD
    :
  :
  :

```



1: sm_first
2: sm_type_spec

Figura III.11 - Exemplo da declaração de tipos incompletos.

Note que essa solução permite que a declaração de tipos seja tratada de uma maneira uniforme, sem que se tenha que diferenciar entre declarações de tipos incompletos e completos.

O tratamento de subprogramas é um pouco mais complicado, porque a declaração e o corpo de subprogramas podem ocorrer em unidades de compilação diferentes. Como em princípio a compilação em separado não permite a atualização de unidades já compiladas não é possível modificar os atributos semânticos da especificação que se referenciam ao corpo do subprograma correspondente. Dessa forma, em todos os casos nos quais o corpo do subprograma estiver em uma unidade separada, o valor do atributo "sm_body" do nó "proc_id" ou "function_id" aponta para a sub-árvore do corpo do subprograma. A figura III.12 ilustra esse caso. Uma solução semelhante foi utilizada para o tratamento de "tasks" e "packages".

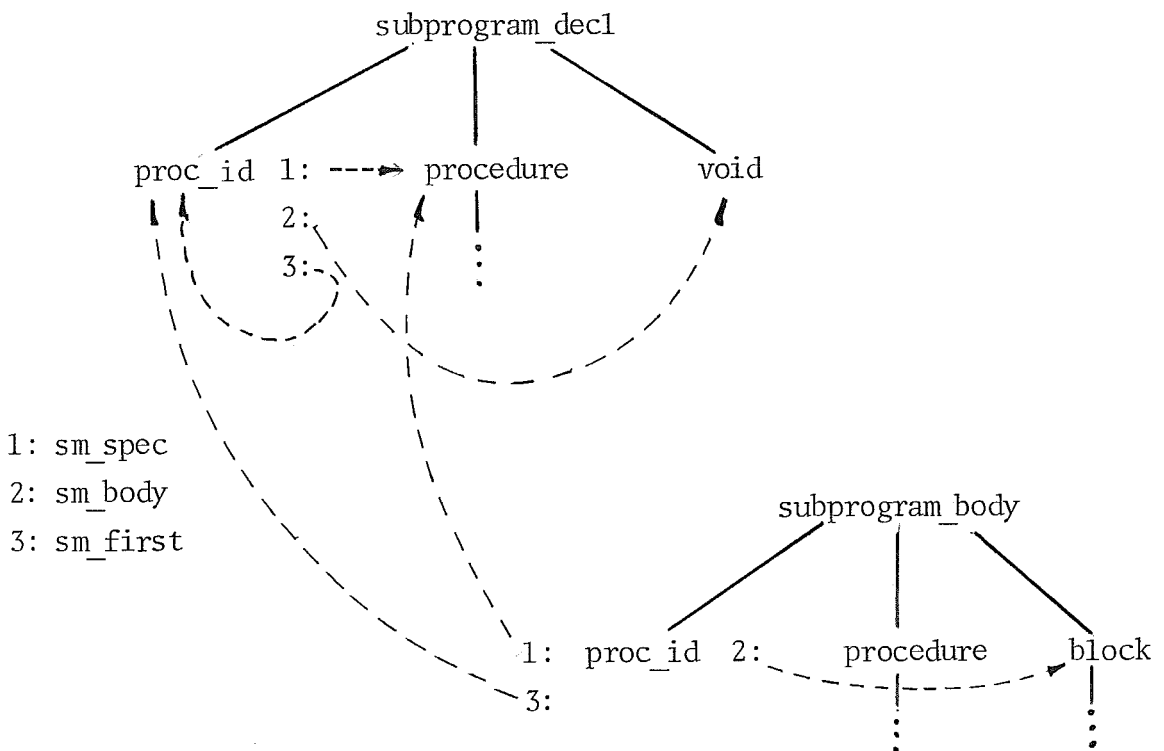


Figura III.12 - Exemplo de declaração e corpo de um sub-programa em unidades de compilação diferentes.

3.4.5. ESPECIFICAÇÃO DE TIPOS

Existem 8 nós em DIANA usados para se guardar informações provenientes de declarações de tipo: "*integer*", "*fixed*", "*float*", "*enum_literal_s*", "*record*", "*array*", "*access*" e "*task_spec*". Embora as informações guardadas nas sub-árvores desses nós sejam importantes para a compreensão da especificação do analisador semântico, seria longo e penoso o detalhamento das mesmas nesse ponto, de forma que se optou por apresentar as sub-árvores de indicação de subtipo ("*subtype_indication*") e de tipos derivados.

Toda indicação de subtipo é representada por um nó "*constrained*", que possui a seguinte definição:

```
constrained => as_name: NAME,
               as_constraint: CONSTRAINT,
               cd_impl_size: integer,
               lx_srcpos: source_position,
               lx_comments: comments,
               sm_type_struct: TYPE_SPEC,
               sm_base_type: TYPE_SPEC,
               sm_constraint: CONSTRAINT;
```

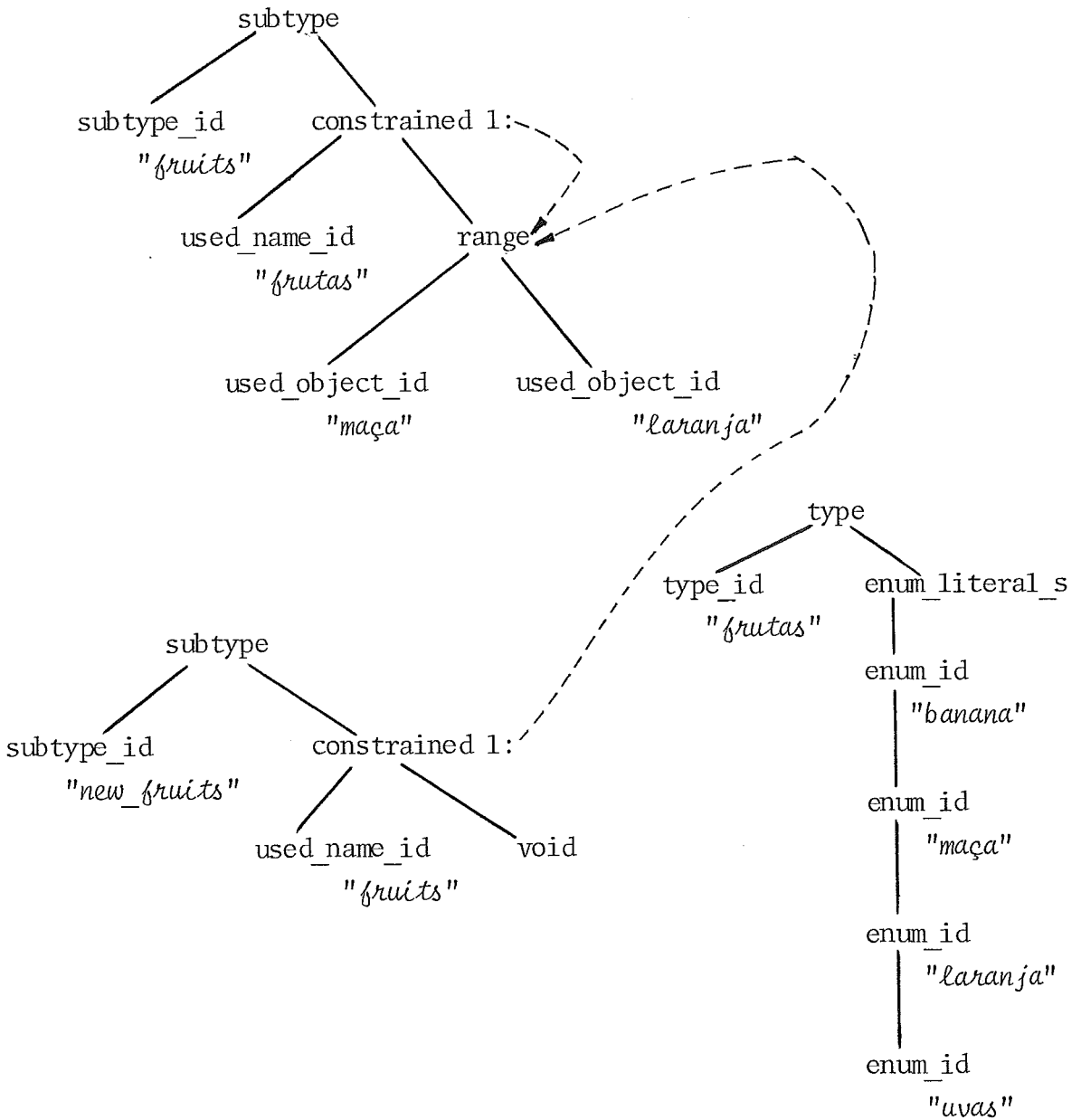
Figura III.13 - Definição do nó "*constrained*".

Uma indicação de subtipo pode ser usada simplesmente para se renomear um tipo, sem que se aplique nenhuma restrição ("*constraint*") ao tipo base. Porém, o gerador de código precisa conhecer a última restrição aplicada ao tipo; dessa forma o atributo "*sm_constraint*" do nó "*constrained*" aponta diretamente para essa restrição, facilitando o trabalho do gerador de código. A figura III.14 ilustra o uso desse atributo.

```

TYPE FRUTAS IS (BANANA, MAÇA, LARANJA, UVAS);
SUBTYPE FRUITS IS FRUTAS RANGE MAÇA .. LARANJA;
SUBTYPE NEW_FRUITS IS FRUITS;

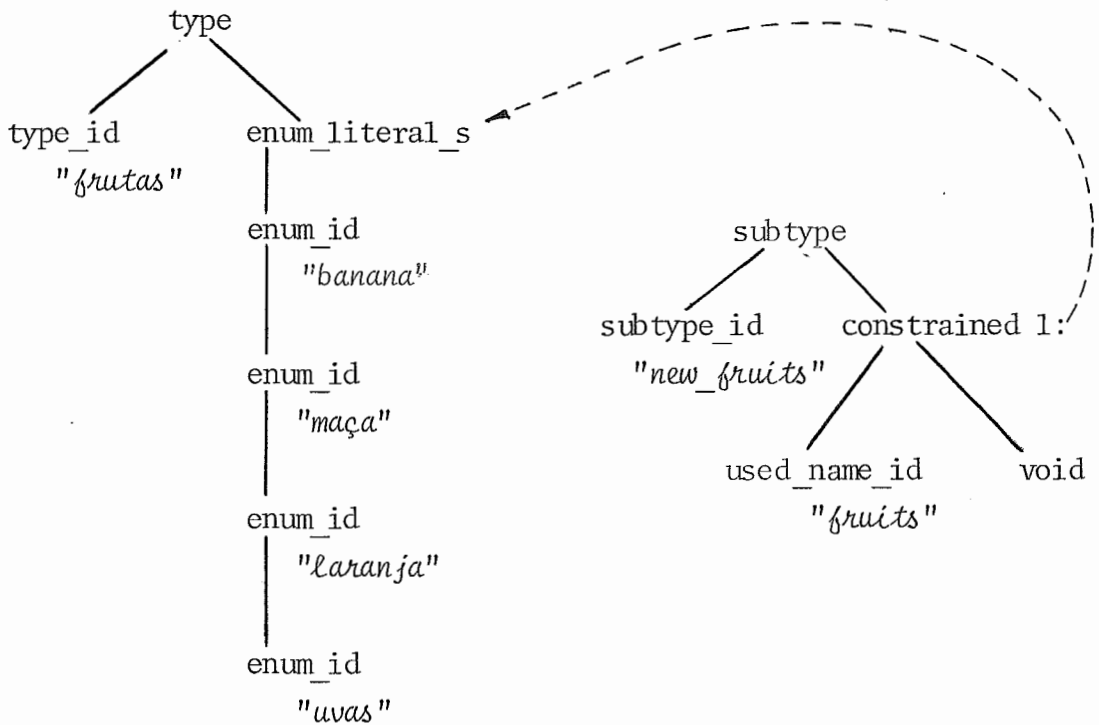
```



1: sm_constraint

Figura III.14 - Exemplo do uso do atributo "sm_constraint"

O gerador de código também necessita de informações sobre a estrutura do tipo que é obtida a partir da definição do tipo original. O atributo `"sm_type_struct"` é usado para guardar essa informação, conforme apresentado na próxima figura.



1: `sm_type_struct`

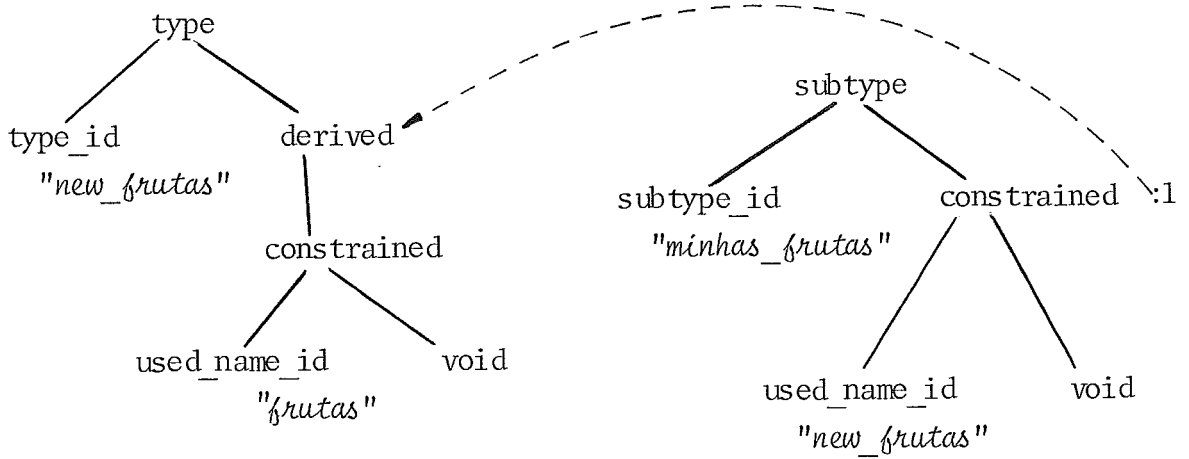
Figura III.15 - Exemplo do uso do atributo `"sm_type_struct"`

Em uma cadeia de especificações de tipos, um programador pode adicionar atributos para cada tipo através de especificações de representação; o tipo do qual um subtipo é construído chamado seu tipo base. O atributo `"sm_base_type"` aponta para a especificação do tipo correspondente, ou seja, para uma sub-árvore na qual toda informação de representação pode ser encontrada. A figura III.16 mostra um exemplo do uso desse atributo.

```

TYPE FRUTAS IS(BANANA, MAÇA, LARANJA, UVA);
TYPE NEW_FRUTAS IS NEW FRUTAS;
SUBTYPE MINHAS_FRUTAS IS NEW_FRUTAS;

```



1: sm_base_type

Figura III.16 - Exemplo do uso do atributo "sm_base_type".

3.4.6. DUPLICAÇÃO DE SUB-ÁRVORES

O exemplo anterior apresenta uma característica importante de DIANA. Não foi representado na figura, mas o nó "constrained" possui um atributo do tipo "sm_type_struct" que, a princípio, apontaria para o nó "enum_literal_s" da definição do tipo "FRUTAS", conforme a próxima figura.

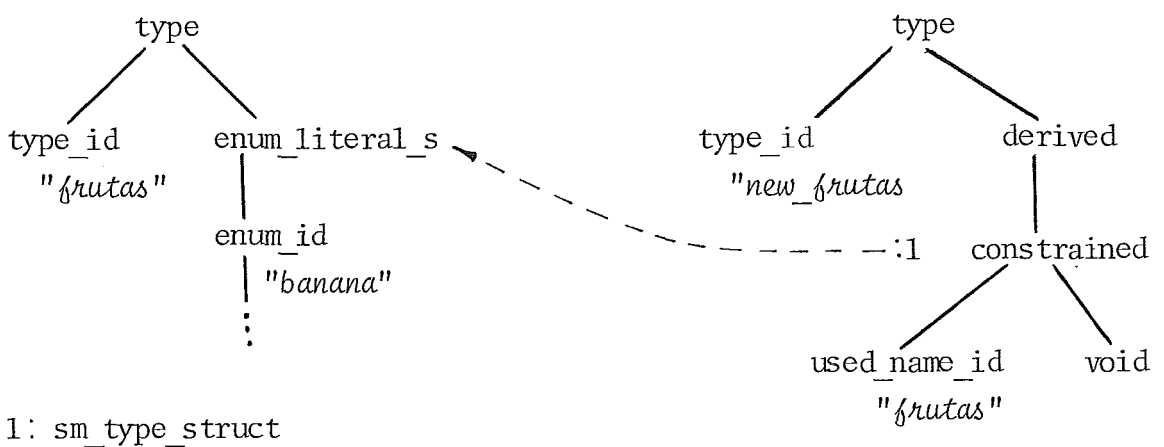


Figura III.17 - Exemplo do uso do atributo "sm_type_struct".

Porém, ADA permite que um programador forneça, por exemplo, as seguintes especificações de representação:

```

TYPE FRUTAS IS (BANANA, MAÇA, LARANJA, UVA);
FOR FRUTAS USE
  (BANANA => 3, MAÇA => 8, LARANJA => 15, UVA => 22);
TYPE NEW_FRUTAS IS NEW FRUTAS;
FOR NEW_FRUTAS USE
  (BANANA => 8, MAÇA => 10, LARANJA => 17, UVA => 20);

```

Os valores fornecidos na 1ª especificação ficam armazenados nos atributos "sm_rep" dos nós "enum_id" da sub-árvore de especificação do tipo enumeração. Fica evidente que os valores fornecidos na 2ª especificação de representação não podem ficar sobrepostos aos valores previamente fornecidos. A solução para

essa situação é a duplicação da sub-árvore da estrutura do tipo, conforme demonstrado na figura III.18.

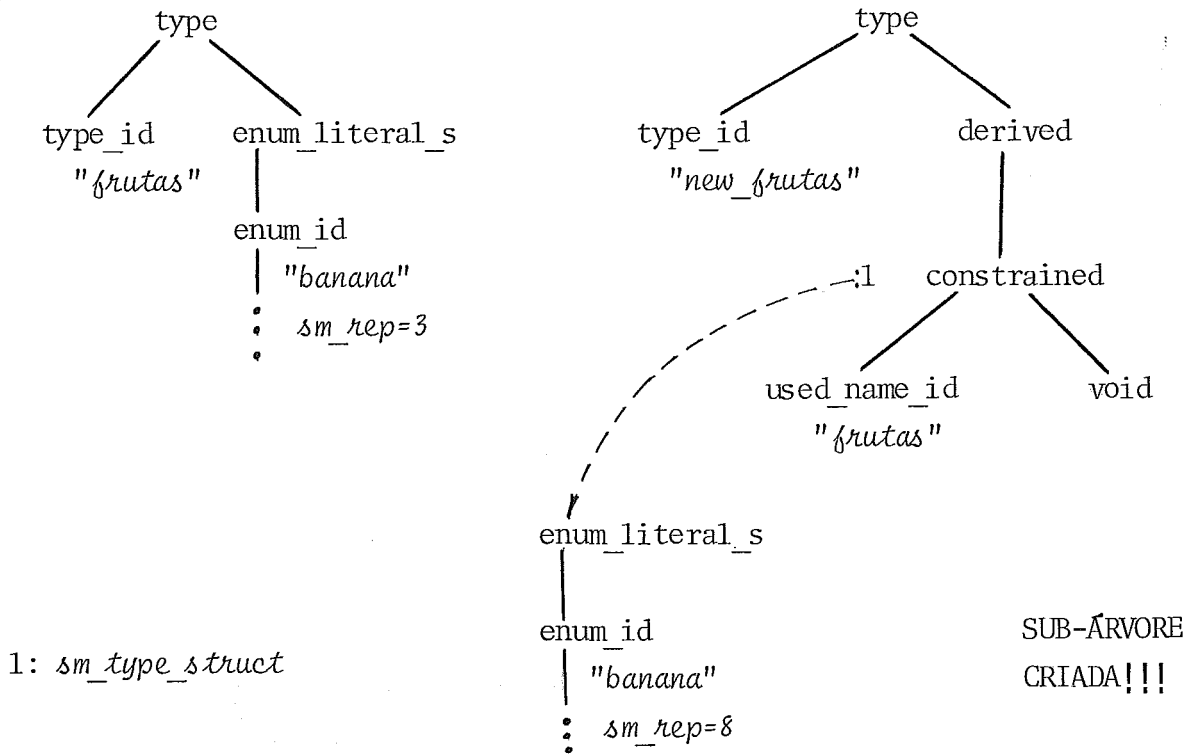


Figura III.18 - Exemplo da duplicação de uma sub-árvore pelo analisador semântico.

A duplicação da sub-árvore só é necessária se uma cláusula de representação for fornecida pelo programador para o tipo derivado. Uma implementação pode decidir se duplica a sub-árvore automaticamente ou não. O mesmo problema ocorre com a representação de uma estrutura do tipo record.

3.4.7. TRATAMENTO DE INSTANCIACÕES

Como já foi visto, uma unidade genérica é uma unidade de programa ADA considerada como um gabarito ("*template*") que pode ser parametrizada ou não e da qual subprogramas ou pacotes não genéricos podem ser obtidos; a unidade resultante é chamada uma *instância* da unidade original. Uma unidade genérica possui 2 partes: a 1.^a representa a unidade propriamente dita e a 2.^a seus parâmetros formais. Uma instância de uma unidade genérica é obtida através de uma *instanciação*, na qual se fornecem os parâmetros genéricos reais.

À primeira vista, uma instanciação pode ser facilmente obtida pela cópia da unidade genérica e correspondente substituição dos parâmetros formais pelos parâmetros reais. Entretanto, essa solução pode não ser possível se o corpo da unidade genérica tiver sido compilado separadamente. Por outro lado, uma implementação poderia tentar otimizar várias instanciações de uma mesma unidade pelo compartilhamento de código entre elas. Dessa forma, a definição de DIANA não obriga a duplicação do corpo de uma unidade genérica no ponto da declaração de sua instanciação de maneira a se ampliar a liberdade das diversas implementações. Assim, uma instanciação é representada em DIANA simplesmente pela cópia da especificação de sua unidade genérica, na qual toda ocorrência de um parâmetro formal é substituída por seu parâmetro real correspondente. Essa cópia é realizada em 2 passos: no 1.^o, uma lista normalizada com os parâmetros genéricos é criada, usando-se o atributo "*sm_decl_s*" do nó "*instantiation*"; no 2.^o, uma cópia da especificação da unidade é criada, na qual as referências aos parâmetros formais são substituídas por referências às novas entidades criadas (parâmetros reais). A próxima figura procura explicitar melhor esse fato.

GENERIC

```

LENGTH: INTEGER := 200;
TYPE ELEM IS PRIVATE;
PROCEDURE EXCHANGE (U: IN OUT ELEM);
:
PROCEDURE SWAP IS NEW EXCHANGE
      (ELEM => INTEGER);

```

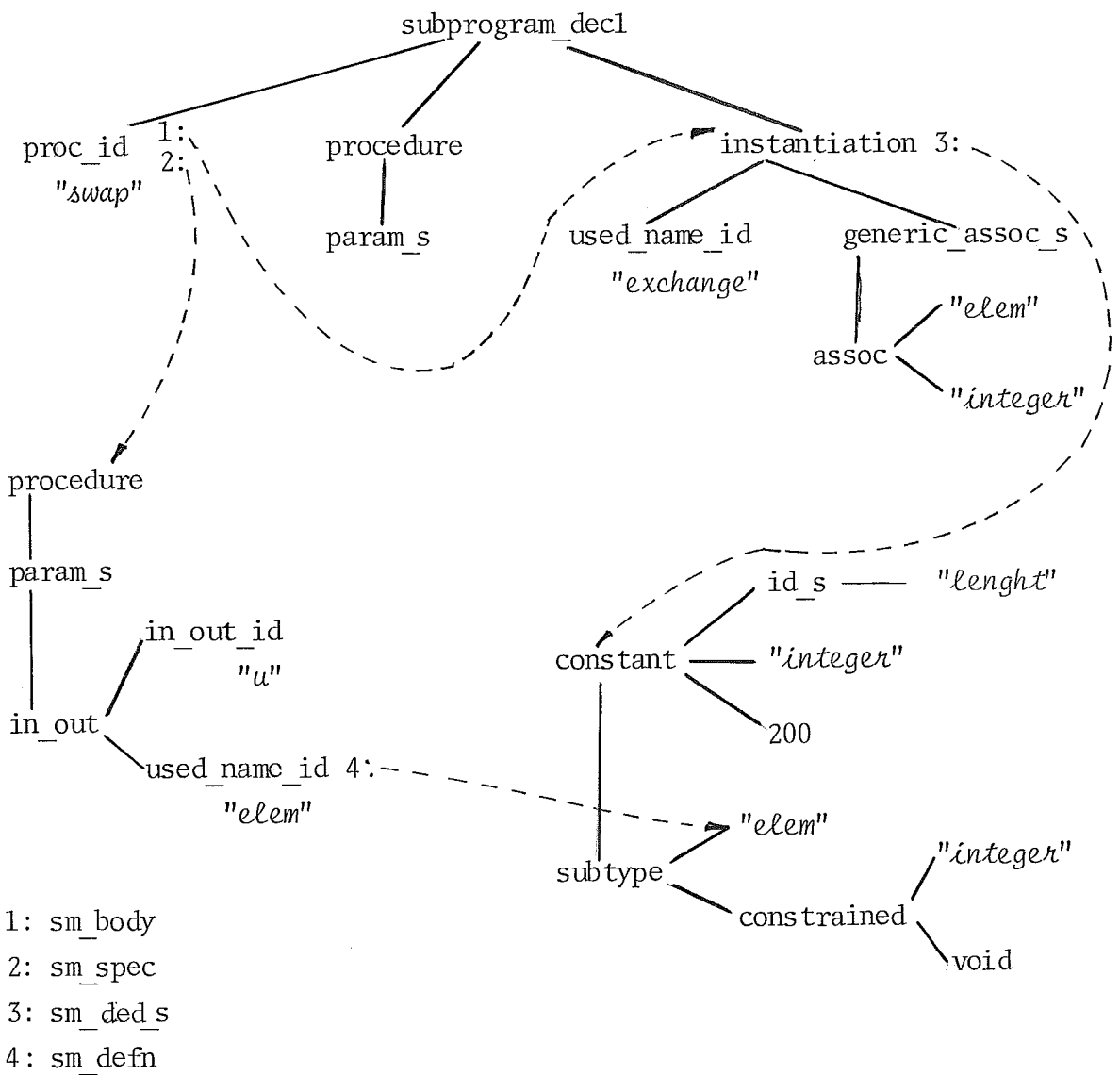


Figura III.19 - Exemplo de instanciação de um sub-programa.

Note que no exemplo dado, as subárvores dos nós "constant" e "procedure", apontadas pelos atributos "sm_exp_s" e "sm_spec" não fazem parte da árvore sintática abstrata, sendo, portanto, explicitamente criadas pelo próprio analisador semântico.

Seguindo-se esta filosofia, a expansão da unidade instanciada deve ficar a cargo do "back-end" do compilador, já que a otimização da mesma deve levar em conta aspectos tais como tamanho da palavra e representação das entidades na máquina alvo, etc... As regras da linguagem são tais que, se uma unidade genérica tiver sido corretamente analisada e sua instanciação for válida, não poderá haver erros semânticos durante sua expansão, o que facilita bastante esta tarefa por outro módulo do compilador que não o analisador semântico.

3.5. OPÇÕES DE IMPLEMENTAÇÃO

Como já foi visto, a definição de DIANA não especifica nenhuma estrutura de dados ou estratégia para sua implementação. Isso causa um enorme impacto em sua utilização, pois cada implementador possui a liberdade de escolher a estrutura que melhor se adapte aos seus objetivos. Obviamente, a máquina na qual DIANA vai ser implementada bem como seu software correspondente (linguagens disponíveis e sistema operacional) e o objetivo da implementação são fatores que devem influenciar fortemente a estrutura escolhida. Um exemplo drástico, porém ilustrativo: a estrutura definida para uma implementação em Fortran (!) certamente não será idêntica a uma implementação em PASCAL. Por outro lado, diferentes compiladores de PASCAL tratam um record variável de uma forma também variável: enquanto alguns compiladores alocam no "heap" a menor porção de memória necessária para um objeto determinado, outros alocam sempre a quantidade necessária para guardar o maior objeto possível. Por outro lado, como o consumo de memória necessário para se guardar a árvore DIANA é considerável, a existência e eficiência dos algoritmos de paginação devem ser levados em consideração, principalmente se for usada alocação dinâmica de objetos em uma área de "heap". Outros dados relevantes para a escolha são o tamanho da memória principal disponível e o tamanho e velocidade de transferência da memória auxiliar. Finalmente não se deve esquecer que o próprio objetivo da implementação deve ser levado em conta: se estiver se trabalhando em um sistema piloto, cuja meta principal seja obter um compilador inicial em pouco tempo para se estudar a utilização da linguagem ou técnicas específicas na implementação da mesma (por exemplo, tratamento de "overloading" ou de compilação em separado) pode-se suportar um certo grau de ineficiência da implementação de DIANA; por outro lado, se o objetivo for um compilador de produção será necessário um projeto mais cuidadoso.

Do que foi apresentado e segundo dois objetivos de nosso projeto já citados (definição de módulos em um nível lógico e não físico e indeterminação da máquina na qual ele vai ser implementado) decidimos não definir uma estrutura física concreta

para implementação de DIANA; isso permitiu concentrar nossas atenções em detalhes do analisador semântico.

Abaixo será apresentado um resumo do modelo que foi usado para a manipulação de DIANA na definição das rotinas semânticas.

Uma idéia que surge ao se imaginar uma estrutura para implementação de DIANA é a utilização de um record para representação dos nós com uma parte variante que contenha os atributos específicos de cada tipo de nó; dessa forma, os atributos estruturais e uma boa parte dos atributos semânticos seriam implementados como um ponteiro ou um valor de acesso para outros nós da estrutura. Esse modelo, que certamente pode ser bastante otimizado, foi escolhido para ser usado na definição de nosso analisador semântico devido à sua grande simplicidade. Note que, entretanto, nossa definição continua independente da estrutura escolhida para implementação física de DIANA pois, como foi ressaltado no capítulo inicial, ela deve servir principalmente como um guia para uma especificação mais completa de um analisador semântico para linguagem ADA, e nessa especificação, os compromissos citados anteriormente devem ser levados em consideração.

A seguir será apresentado um resumo, em ADA, para definição da estrutura utilizada. Primeiramente foi definido o tipo de enumeração "nome-do-nó" que contém um elemento para cada nó de DIANA. Como os nomes de alguns nós de DIANA são conflitantes com palavras reservadas de ADA, todos os elementos do tipo possuem o prefixo "DN", seguido pelo nome do próprio nó.

```

TYPE NO_DIANA IS (
  DN_ABORT,
  DN_ACCEPT
  --
  -- Seguem-se definições de tantos elementos
  -- quantos nós em DIANA.
  --
  DN_WITH);

```

Finalmente são definidos um tipo de record que representa os nós da árvore DIANA e um tipo de acesso correspondente:

```

TYPE TREE (TIPO: NO_DIANA);
TYPE PT_TREE IS ACCESS TREE;
TYPE TREE (TIPO: NO_DIANA) IS
RECORD
  CASE TIPO IS
    WHEN DN_ABORT =>
      AS_NAME: PT_TREE;
      LX_SRCPOS: SOURCE_POSITION;
      LX_COMMENTS: COMMENTS;
    WHEN DN_ACCEPT ==>
      AS_NAME: PT_TREE;
      AS_PARAM_S: PT_TREE;
      AS_STM_S: PT_TREE;
      LX_SRCPOS: SOURCE_POSITION;
      LX_COMMENTS: COMMENTS;
  --
  -- seguem-se definições das partes variantes para
  -- cada um dos nós DIANA restantes
  --
    WHEN DN_WITH =>
      AS_LIST: PT_TREE;
      LX_SRCPOS: SOURCE_POSITION;
      LX_COMMENTS: COMMENTS;
  END CASE;
END RECORD;

```

Foi definida também uma função - "APANHA_NOME_NO" - tal que, dado um ponteiro para determinado nó, é devolvido o elemento do tipo "NO_DIANA" correspondente; esse valor pode ser facilmente obtido pelo discriminante do nó. Os tipos "SOURCE_POSITION" e "COMMENTS" são tipos privados fortemente dependentes da implementação. Os tipos e as funções definidas devem compor um "package" com a especificação de todas as entidades importantes para a definição de DIANA.

Para finalizar deve-se acrescentar que o manual de DIANA indica algumas opções de representação para implementação de DIANA. Como já foi observado, embora nosso trabalho tenha procurado focalizar os tipos acima apresentados, ele é bastante flexível para poder ser implementado com qualquer estrutura que seja definida para DIANA.

CAPÍTULO IV

RESOLUÇÃO DE NOMES E EXPRESSÕES

4.1. CONCEITUAÇÃO

Uma das características mais marcantes da linguagem ADA é a possibilidade de, em qualquer ponto de um programa, vários subprogramas declarados com o mesmo identificador ou operador serem visíveis sem que um esconda os demais (desde que sejam distinguíveis pelo número, tipo e nome de seus parâmetros e, no caso de funções, pelo seu próprio tipo). Tal característica é chamada de sobreposição ("*Overloading*") e é também apresentada por literais de enumeração.

ICHBIAH et al. (19) apresentam no rationale da linguagem as principais vantagens obtidas pelo uso criterioso da sobreposição, uma vez que esta aumenta a liberdade dos usuários para a escolha dos nomes usados em seus programas. Essa característica é ainda mais apreciada quando se leva em consideração a possibilidade de um programador implementar um pacote com declarações de novos tipos e de operações sobre esses tipos para ser usado por uma comunidade de usuários. Se por um lado o programador do pacote não tem condições de saber em que contextos o mesmo será usado de modo a escolher nomes que não interfiram no contexto, por outro lado seria bastante incômodo para os usuários a preocupação da escolha de nomes não definidos no pacote. Deve-se lembrar que a maioria das linguagens de programação já faz uso, de forma reduzida, desse conceito quando, por exemplo, o operador "+" é usado tanto para soma de valores inteiros quanto para a soma de valores reais.

Por outro lado, ADA permite que um tipo seja derivado de outro já existente. O novo tipo, embora distinto do tipo base, herda suas propriedades e características. O manual de referência da linguagem define as circunstâncias nas quais subprogramas que usem o tipo base são implicitamente derivados para operar com o novo tipo derivado. Embora não caiba discu

tir essas circunstâncias aqui, deve-se notar que a redefinição implícita de subprogramas provoca o sobreposicionamento dos mesmos, como mostrado na figura IV.1.

```
TYPE NOVO_INT IS NEW INTEGER;
```

Figura IV.1 - Derivação do tipo "integer"

O tipo "NOVO_INT" é um novo tipo, distinto do tipo "INTEGER" predefinido. O operador "+" é automaticamente derivado para o tipo "NOVO_INT", ficando sobreposto aos operadores "+" predefinidos no pacote "STANDARD".

O manual de referência da linguagem lista as condições que duas ou mais especificações de subprogramas devem possuir para que uma não esconda as demais; "duas partes formais possuem o mesmo perfil de tipo parâmetro (*"parameter type profile"*) se e só se elas possuírem o mesmo número de parâmetros e os parâmetros das mesmas posições possuírem o mesmo tipo base. Um subprograma ou entrada (*"entry"*) possui o mesmo perfil de tipo de parâmetro e resultado (*"parameter and result type profile"*) que outro subprograma se e somente se ambos tem o mesmo perfil de tipo de parâmetro e ou ambos são funções que tenham o mesmo tipo base ou nenhum é função". (ARM.Sc 6.6). Usando-se as informações fornecidas acima, está definido que um subprograma somente pode esconder outro que tenha o mesmo designador se ambos tiverem o mesmo perfil de tipo de parâmetro e resultado.

A identificação da chamada de um subprograma sobreposto deve levar em consideração, além de seu designador, o número de parâmetros usados na chamada, o tipo e a ordem dos parâmetros reais, os nomes dos parâmetros formais (se associação por nome for usada) e o tipo do valor devolvido, no caso do subprograma ser uma função. Se essas informações não forem suficientes para se determinar qual subprograma está sendo usado, trata-se de uma chamada ambígua e, portanto, ilegal.

De forma geral, o uso de uma entidade sobreposta é le

gal se houver somente uma interpretação para cada constituinte do menor contexto completo que a envolve, definido como uma declaração, um comando ou uma cláusula de representação. Além das regras normais de sintaxe, de escopo e de visibilidade, as únicas regras que podem ser usadas para solução de tipos são as seguintes (ARM. Sc 8.7).

- a) Qualquer regra que requeira que um nome ou expressão tenha um certo tipo, ou tenha o mesmo tipo que outro nome ou expressão.

Por exemplo: As expressões que indicam as condições em um comando "IF" devem ter o tipo booleano (ARM. Sc. 5.3); em um comando de atribuição, a variável e a expressão do lado direito devem ter o mesmo tipo (ARM. Sc. 5.2).

- b) Qualquer regra que requeira que o tipo de um nome ou expressão pertença a uma certa classe; paralelamente, qualquer regra que requeira que determinado tipo seja discreto, inteiro, real, universal, caracter, booleano ou não-limitado.

Por exemplo: A expressão de um comando "CASE" deve ser de um tipo discreto determinável independentemente do contexto no qual a expressão ocorre, mas usando-se o fato da expressão ter um tipo discreto. (ARM. Sc. 5.4.3).

- c) Qualquer regra que requeira que um prefixo seja apropriado para um determinado tipo.

Por exemplo: Em um componente selecionado que denote um componente de um record, o seletor deve ser um nome simples que denote um componente de um record, e o prefixo deve ser apropriado para o seletor. (ARM. S.c 4.1.3).

- d) Qualquer regra que especifique um certo tipo como o resultado de uma operação básica, e qualquer regra que especifique que esse tipo seja de uma certa classe.

Por exemplo: o resultado de um teste de pertinência ("membership") deve ser do tipo predefinido booleano (ARM. Sc. 4.5.2).

- e) As regras que requeiram que o tipo de um agregado ou de um literal de cadeias de caracteres seja determinado somente pelo contexto externo. Paralelamente, as regras que requeiram que o tipo do prefixo de um atributo, da expressão de um comando "CASE" ou do operando de uma conversão de tipo seja determinado independentemente do contexto.
- f) As regras especificadas na seção 6.6 para solução de chamadas de subprogramas sobrepostos; na seção 4.6 para a conversão implícita de uma expressão universal; na seção 3.6.1 para interpretação de um intervalo discreto ("*discrete range*") com limites que tenham um tipo universal; na seção 4.1.3 para a interpretação de um nome expandido cujo prefixo denote um subprograma ou comando "ACCEPT".

4.2. ALGORITMOS

Conceituado o problema e apresentadas as regras gerais que podem ser usadas para resolvê-lo, resta-nos somente apresentar soluções para o mesmo. O rationale da linguagem apresenta um algoritmo para solução de tipos baseado em vários passos ascendentes e descendentes sobre a árvore de derivação da expressão. Um percurso ascendente sobre a árvore procura restringir os tipos possíveis da raiz de cada sub-árvore a partir das informações de seus filhos, ao passo que um percurso descendente procura restringir os tipos possíveis de cada nó das diversas sub-árvores de acordo com os tipos possíveis de seu nó pai. O algoritmo termina quando não for efetuada nenhuma modificação nos tipos dos diversos nós da árvore; nesse momento a expressão é inválida se algum nó apresentar mais que um tipo possível ou não apresentar nenhum tipo possível.

PERSCH et al. (20) demonstram que um percurso ascendente seguido de um percurso descendente sobre uma árvore "decorada" (ou seja, uma árvore na qual cada nó possua uma lista com todos os tipos possíveis para o mesmo) é suficiente para a resolução do tipo de uma expressão ou um nome.

O exemplo mostrado nas próximas figuras procura ilustrar o método proposto.

```

TYPE INT1 IS NEW INTEGER;
TYPE INT2 IS NEW INTEGER;
FUNCTION "+" (X: INT1; Y: INT) RETURN INT1;
FUNCTION "+" (X: INT1; Y: INT) RETURN INT2;
I: INTEGER;
N1: INT1;
N2: INT2;

```

Figura IV.2 - Exemplo de sobreposição da função "+".

seja a seguinte expressão:

$$((3 * N1) + (I/5)) - N2$$

com a respectiva árvore de derivação abstrata:

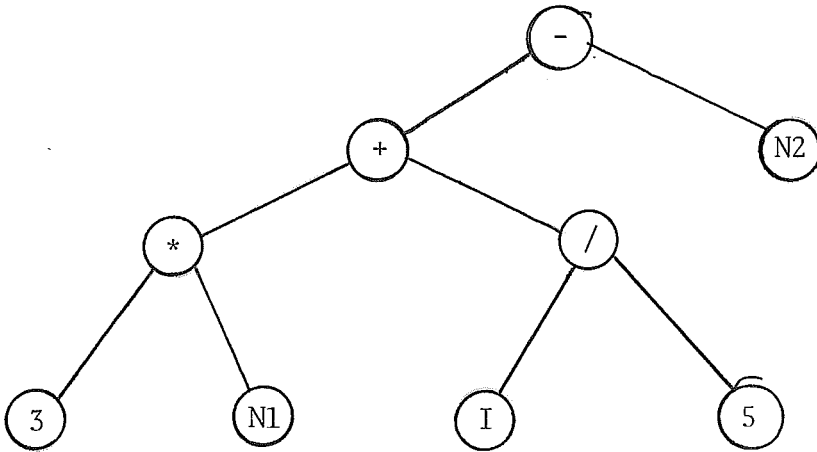


Figura IV.3 - Árvore de derivação abstrada da expressão fornecida.

e a clássica pergunta: qual o tipo da expressão?

O passo inicial do algoritmo deve "decorar" a árvore, ou seja, percorre-la colocando em cada nó as informações de tipo pertinentes ao mesmo. No exemplo dado, cada folha que representar um identificador terá o tipo do mesmo. Os literais numéricos usados no exemplo possuem o tipo predefinido "UNIVERSAL INTEGER", que pode ser implicitamente convertido para qualquer um dos tipos "INTEGER", "INT1" ou "INT2". Cada operador (que é definido em ADA como sendo uma função) terá uma lista com os tipos de seus parâmetros e o tipo do resultado devolvido. (Por exemplo, os operadores definidos nesse exemplo serão representados respectivamente por "INT1, INT→INT1" e "INT1, INT→INT2" ; "INT" é usado no lugar de "INTEGER" por simplicidade). A árvore decorada tem o seguinte aspecto:

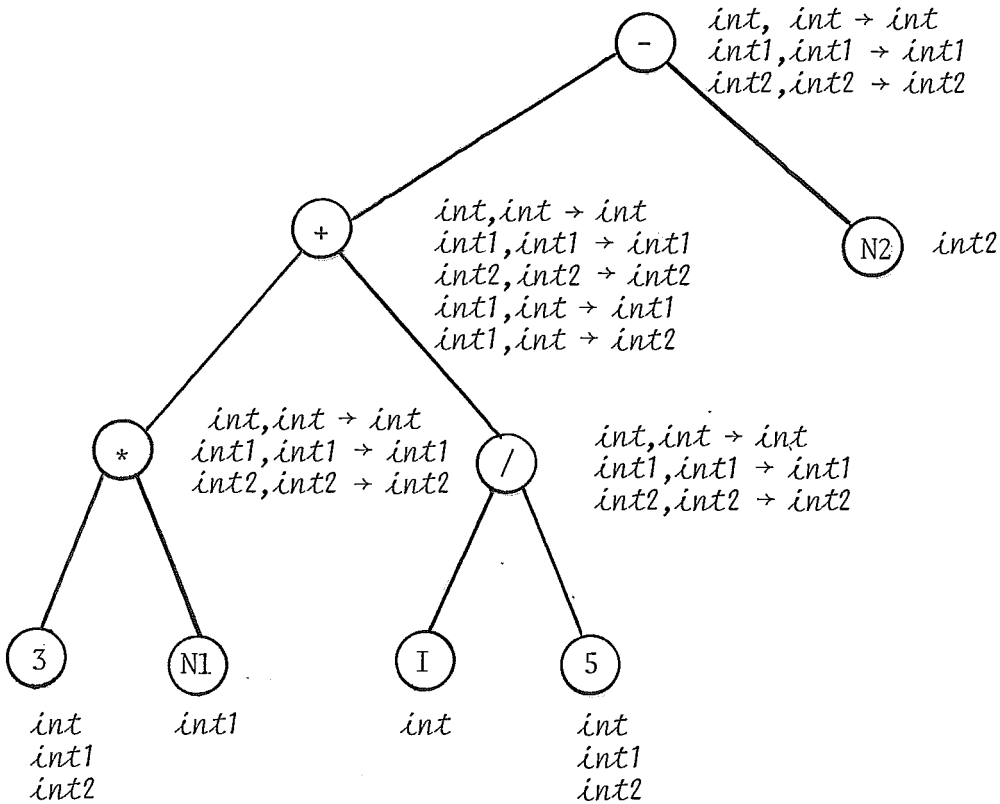


Figura IV.4. - Árvore de derivação abstrata decorada

Note que, para efeito de apresentação, estão representadas as funções implicitamente derivadas para os novos tipos "INT1" e "INT2"; por outro lado, são colocados todos os tipos que os literais numéricos "3" e "5" podem assumir. Uma possibilidade a ser considerada por ocasião da implementação é a de se usar nomes padronizados como abreviatura de listas de tipos frequentemente utilizados.

O 1º percurso na árvore é ascendente e procura limitar o tipo das raízes das sub-árvores de acordo com o tipo de seus filhos. Seja, por exemplo, a sub-árvore do nó "*":

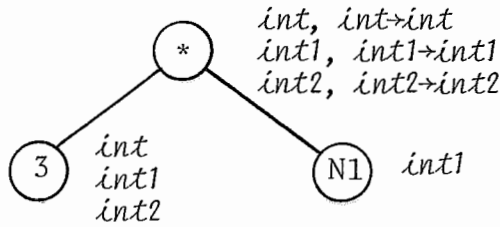


Figura IV.5 - Sub-árvore do operador "*".

Como pode-se observar, na sub-árvore estão representadas tanto a função predefinida para o tipo inteiro quanto as 2. funções implicitamente derivadas para os tipos "INT1" e "INT2". Porém, como seu 2º operando é do tipo "INT1", restringe-se o nó "*" a operar somente com esse tipo de operando, resultando na seguinte sub-árvore:

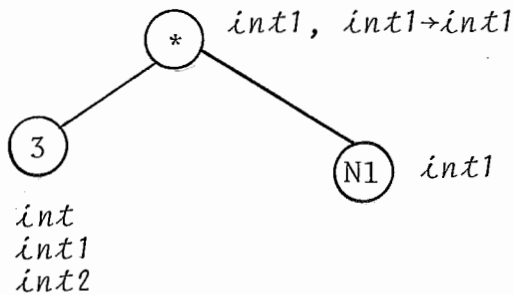


Figura IV.6 - Sub-árvore após 1º passo do algoritmo.

O resultado final é apresentado na figura IV.7.

O 2º percurso na árvore é descendente, quando procura-se limitar o tipo dos nós de acordo com o tipo de seu pai. Seja novamente o exemplo da sub-árvore do nó "*", apresentado na figura IV.8.

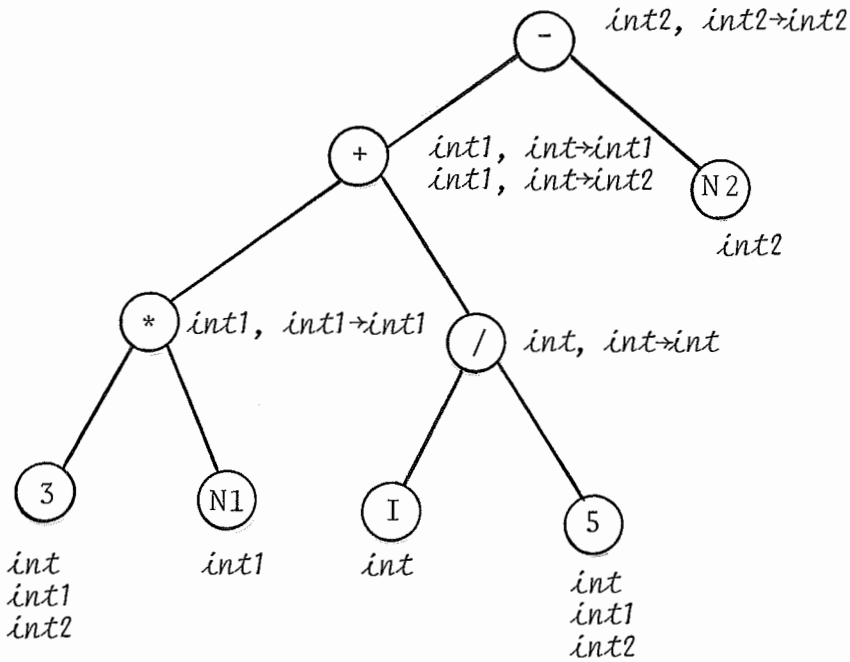


Figura IV.7 - Árvore de derivação abstrata após 1º passo do algoritmo

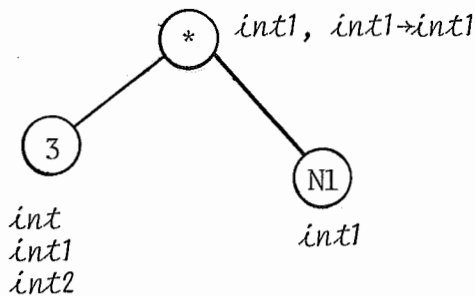


Figura IV.8 - Sub-árvore do nó "*" antes do 2º passo do algoritmo.

Como o operando esquerdo do operador de multiplicação deve ter o tipo "INT1", o tipo da literal numérica "3" fica definido. Note que 2 situações de erro podem ocorrer quando o fluxo do processamento atinge um nó: ou não existe nenhum tipo possível para o nó, (e nesse caso a expressão fica indefinida) ou existe mais que um tipo possível para o nó (e nesse caso a ex

pressão é ambígua).

O resultado final do algoritmo é dado a seguir:

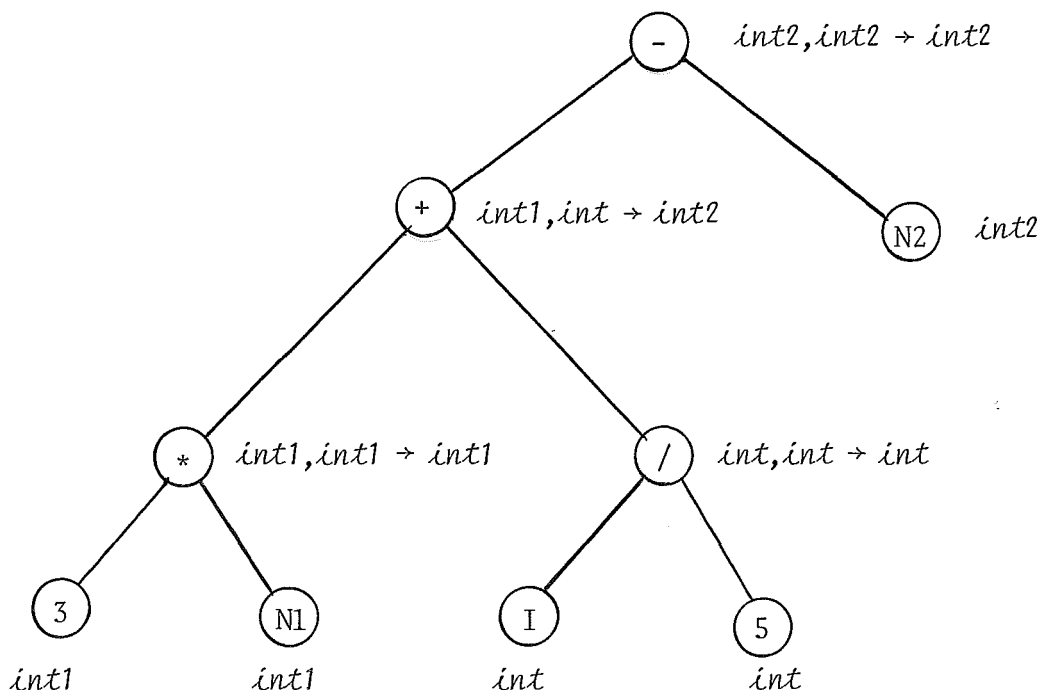


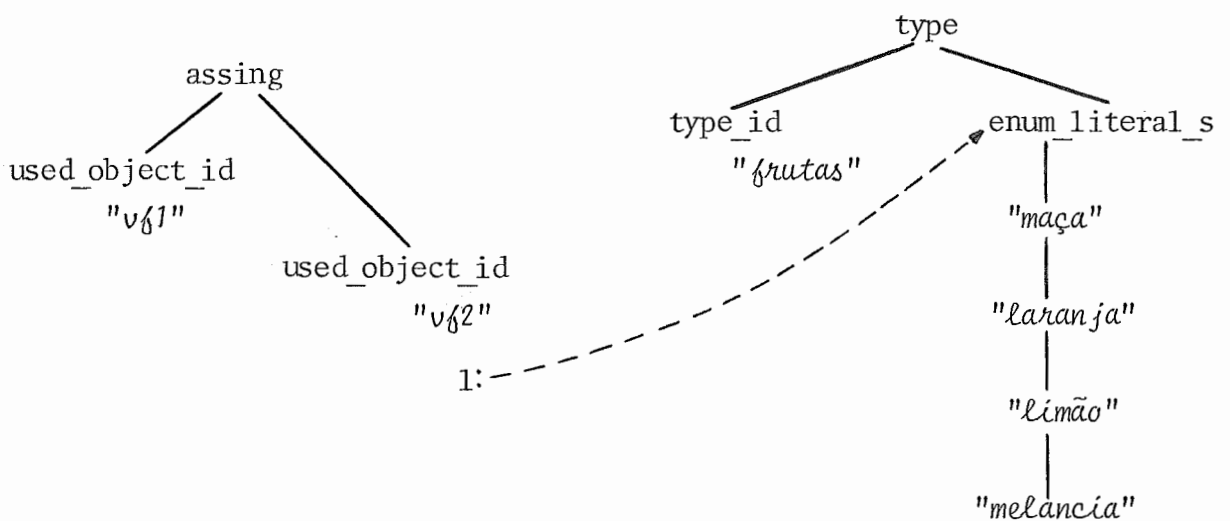
Figura IV.9 - Árvore sintática de derivação com tipos definidos.

BACKER (21) propõe um algoritmo que efetua um único percurso ascendente na árvore, durante o qual um conjunto de grafos acíclicos direcionados com múltiplas entradas é produzido, cada entrada correspondendo a uma interpretação diferente da expressão. A expressão está correta se, ao se atingir sua raiz, houver um e só um grafo válido, ou seja, que represente uma expressão semanticamente correta. O algoritmo apresentado anteriormente foi o escolhido para a especificação do analisador semântico por ser bastante simples de ser implementado e, de certa forma, por ser mais natural. Deve-se ressaltar que a bibliografia apresenta outros algoritmos para resolução de tipos (GANZINGER et al. (22), PENELLO et al. (23)), mas que não serão considerados por serem ineficientes (a maior parte dos algoritmos efetua mais que 2 passos sobre a árvore sintática abstrata da expressão).

4.3. DIANA

Como foi apresentado no capítulo anterior, DIANA guarda o resultado da resolução de tipos de uma expressão (ou sub-expressão) em cada um dos seus nós através do atributo "*sm_exp_type*" (DRM. Sc. 3.4.3). No que diz respeito à resolução de tipo, somente o tipo base de uma expressão é relevante, exceto para expressões que denotem valores que devam satisfazer alguma restrição. Por exemplo, a conversão de uma expressão para um tipo derivado de seu tipo base consiste na conversão propriamente dita seguida de um teste que verifica se o valor da expressão pertence ao tipo derivado. Dessa forma, nesses casos especiais o atributo "*sm_exp_type*" aponta para um nó do tipo "*constrained*", que representa a restrição, ao passo que no caso geral o atributo aponta para a sub-árvore que representa a especificação do tipo base da expressão, conforme os seguintes exemplos:

```
TYPE FRUTAS IS (MAÇA, LARANJA, LIMÃO, MELANCIA);
VF1, VF2: FRUTAS;
VF1:= VF2;
```



1: *sm_exp_type*

Figura IV.10 - Exemplo de uso do atributo "*sm_exp_type*".


```

TYPE CÍTRICAS IS NEW FRUTAS RANGE LARANJA .. LIMÃO;
VC: CÍTRICAS;
VC:= CÍTRICAS (VF1);

```

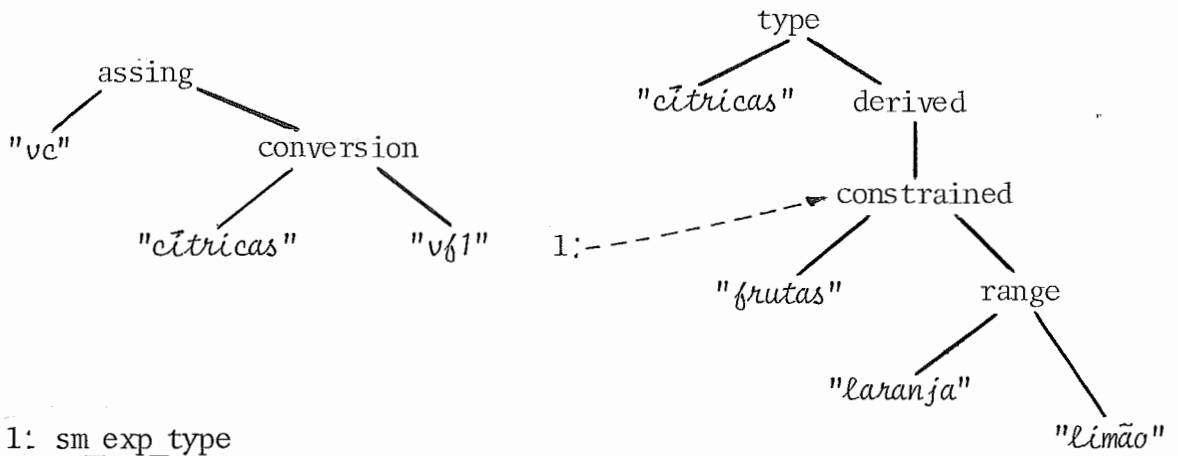


Figura IV.11 - Exemplo de uso do atributo "sm_exp_type"

Na definição de DIANA não existe um atributo que sirva para relacionar um nó com sua lista de tipos. Na realidade não existe nenhum motivo para que tal atributo exista, uma vez que ele só é usado durante a fase de análise semântica, não sendo necessário para as outras fases do compilador nem para outras ferramentas que utilizem a árvore DIANA gerada. Dessa forma houve a necessidade de se definir novos atributos, novos nós e novas classes de nós que são usados nas rotinas de resolução de tipo. Note que esses nós podem ser criados ou filtrados pelo módulo responsável pela leitura e impressão da árvore DIANA na biblioteca.

As novas entidades usadas pelas rotinas de resolução de tipo são listadas abaixo, com a mesma notação usada no manual de DIANA:

```

1. se_lista_de_tipos: TIPO_RETORNO_S;
   TIPO_RETORNO_S ::= tp_tipo_retorno_s;
   tp_tipo_retorno_s => se_lista: seq of TIPO_RETORNO;
   TIPO_RETORNO ::= tp_tipo_retorno;
   tp_tipo_retorno => se_tipo: TYPE_SPEC
                   se_classe: tipo_classe_void

```

Essas entidades foram criadas para representar a lista de tipos ou a classe do tipo que um determinado nó pode ter. A estrutura é agregada em qualquer nó que possa aparecer na sub-árvore de uma expressão.

```

2. tp_lista_de_parametro_s=>se_lista: seq of LISTA_DE_PARAMETROS;
LISTA_DE_PARAMETROS::=tp_lista_de_parametros;
tp_lista_de_parametros => se_lista: PARAMETRO_S;
PARAMETRO_S::= tp_parametro_s;
tp_parametro_s => se_lista: seq of PARAMETRO;
PARAMETRO::= tp_parametro
tp_parametro => se_def_par: DEF_ID; -- in_id out_id
                    se_e_default: boolean;
                    se_ja_usado: boolean;
                    se_tipo: TYPE_SPEC;

```

Essa estrutura é usada para a resolução de chamadas de rotinas e funções e do comando "*entry_call*", de modo a formar uma lista com informações dos parâmetros de cada identificador sobreposto.

```

FUNCTION F (P1: INTEGER; P2: BOOLEAN) RETURN BOOLEAN;
FUNCTION F (P3: BOOLEAN) RETURN INTEGER;

```

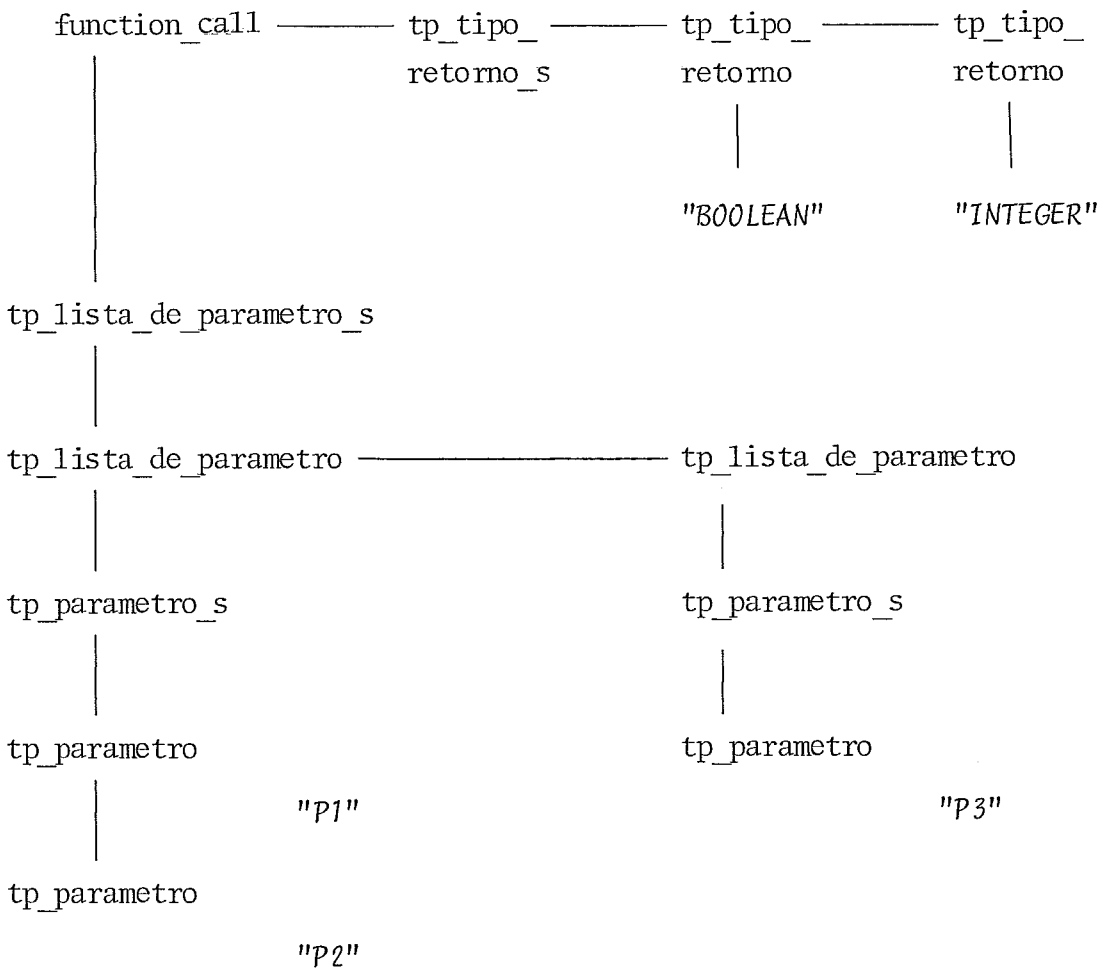


Figura IV.12 - Exemplo do emprego dos novos atributos definidos

Note que, para simplificar a figura, foram listadas somente as novas estruturas incorporadas ao nó "*function_call*".

4.4. ESPECIFICAÇÃO DO ALGORITMO

A rotina "RESOLVE_NOME_E_EXPRESSAO" é chamada sempre que o analisador semântico encontrar uma dessas construções na árvore DIANA sintática. Essa rotina chama 2 rotinas auxiliares: a 1.^a - "DECORA" - faz um percurso descendente na árvore de modo que para cada nó atingido ela monta as estruturas para a construção da lista de tipos do nó apresentadas anteriormente e visita recursivamente seus filhos. O término da visita de um nó filho equivale a um passo da parte ascendente do algoritmo de resolução de tipos apresentado, quando o tipo de um nó pode ser restringido de acordo com os tipos de seus filhos.

Quando termina esse 1.^o percurso na árvore, verifica-se quais informações relativas ao contexto no qual a expressão aparece podem ser usadas para identificação de seu tipo (por exemplo, se a expressão que estiver sendo analisado for a condição de um comando "IF" seu tipo deve ser booleano).

A 2.^a rotina chamada - "DESCIDA_FINAL" - faz outro percurso descendente na árvore, equivalente ao 2.^o passo do algoritmo de resolução de tipo; essa rotina tem 2 parâmetros: o nó a ser visitado e o tipo esperado para esse nó. Note que nesse ponto o contexto já deve ter fixado o tipo que o nó que está sendo visitado deve ter, pois em caso negativo a expressão seria ambígua ou indefinida. Dessa forma, a rotina primeiramente verifica se o nó que está sendo visitado pode ter o tipo que foi passado como parâmetro, ou seja, se esse tipo está contido na lista de tipos do nó em questão; em caso negativo a expressão é inválida. Em caso positivo verifica-se se cada filho do nó tem um tipo definido e, se isso ocorrer, a rotina é recursivamente chamada para tratar esse filho.

A seguir será descrita a rotina que trata o nó DIANA "binary", usado para representar as expressões do tipo "short_circuit" ("and then" e "or else"). A descrição desse nó em IDL é fornecida a seguir:

```

EXP ::= binary;
binary => as_exp1: EXP,
         as_binary_op: BINARY_OP,
         as_exp2: EXP,
         sm_exp_type: TYPE_SPEC;
BINARY_OP ::= SHORT_CIRCUIT_OP;
SHORT_CIRCUIT_OP ::= and_then | or_else;

```

Figura IV.13 - Descrição em IDL do nó "*binary*"

Esses operadores são definidos para 2 operandos do tipo booleano e devolvem um valor que também pertence a esse tipo. As rotinas que tratam o nó "*binary*" são listadas abaixo.

```

PROCEDURE DECORA_BINARY (PNO: PT_TREE) IS           L1
PE1, PE2: PT_TREE;                                L2
BEGIN                                              L3
    PE1 := AS_EXP1 (PNO);                          L4
    PE2 := AS_EXP2 (PNO);                          L5
    CRIA_LISTA_TIPO (PNO);                         L6
    COLOCA_TIPO (PNO, "BOOLEAN");                 L7
    DECORAR (PE1);                                 L8
    DECORAR (PE2);                                 L9
END;                                               L10

```

Figura IV.14 - Rotina que efetua o 1º passo do algoritmo no nó "*binary*".

```

PROCEDURE DESCIDA_FINAL_BINARY (PNO, PL_TIPO: PT_TREE) IS L11
PE1, PE2: PT_TREE; L12
BEGIN L13
  IF NOT CONTIDO (PL_TIPO, PNO) THEN L14
    MSG ("ESPERADO TIPO BOOLEANO", PNO); L15
    SM_EXP_TYPE (PNO) := TIPO_GERAL; L16
  ELSE L17
    SM_EXP_TYPE (PNO) := PL_TIPO; L18
    DESCIDA_FINAL (PE1, PL_TIPO); L19
    DESCIDA_FINAL (PE2, PL_TIPO); L20
  END IF; L21
END; L22

```

Figura IV.15 - Rotina que efetua o 2º passo do algoritmo no nó "*binary*".

O tipo "PT_TREE" (linhas 1, 2, 11 e 12) é o tipo dos nós da árvore DIANA. No início da rotina "DECORA_BINARY" são atribuídos ponteiros às expressões esquerda e direita do nó (L4 e L5). A rotina "CRIA_LISTA_TIPO" é chamada para formar e inicializar a estrutura que conterá a lista de tipos do nó (L6). Como o tipo do valor devolvido pela expressão é booleano, a rotina "COLOCA_TIPO" é chamada para colocar um elemento que represente esse tipo na lista de tipos do nó "*binary*" (L7). Finalmente a rotina é recursivamente chamada para tratar os 2 operandos do nó (L8 e L9).

A rotina "CONTIDO" verifica se o tipo representado pelo 1º parâmetro pertence à lista de tipos do nó apontado por seu 2º parâmetro (L14). No caso, a lista de tipos do nó "*binary*" possui somente um elemento, que representa o tipo booleano. Em caso negativo é emitida uma mensagem de erro (L15) e o atributo "*sm_exp_type*" do nó é preenchido com o valor "tipo-geral", indicador de erro (L16). Na realidade esse valor representa um ponteiro para a descrição de um tipo que só é acessado pelo analisador semântico. Caso a rotina "CONTIDO" devolva o valor verdadeiro, o atributo "*sm_exp_type*" do nó é preenchido com o ponteiro "PL_TIPO" (L18), que necessariamente aponta para a definição do tipo "BOOLEAN". Finalmente, a rotina é recursivamen

te chamada para tratar seus 2 operandos.

Um ponto que merece destaque é o tratamento de situações de erro: sempre que se atinge um nó que configure uma situação semanticamente incorreta seus filhos não são visitados e o atributo "*sm_exp_type*" do nó fica com um valor especial ("tipo-geral") que aponta para a especificação de um tipo especial, que pode ser usado em qualquer circunstância de modo a tentar se eliminar mensagens de erro em cascata, e que é acessível somente pelo analisador semântico.

O apêndice 2 apresenta a listagem das rotinas de resolução de nomes e expressões.

CAPÍTULO V

ANALISADOR SEMÂNTICO ESTÁTICO

5.1. ASPECTOS INICIAIS

A definição da fronteira entre as regras de semântica estática e semântica dinâmica constitui uma decisão delicada na especificação e projeto de uma linguagem de programação e seu respectivo compilador. Basicamente, as regras de semântica estática são as que, não tratando de aspectos léxicos e sintáticos, podem ser testadas em tempo de compilação. Esses testes, por definição, não devem depender de características de execução do programa sendo analisado.

Particularmente, o manual de referência de ADA indica que um compilador para essa linguagem deve ser capaz de, em tempo de compilação, identificar qualquer erro que não seja associado com as exceções predefinidas da linguagem e que não caracterize um programa como errôneo (ARM. Sc 1.6.2). A definição de um conjunto de rotinas que permita a detecção desses erros constitui-se um dos objetivos principais desse trabalho. Por outro lado, o compilador deve gerar código para testar situações de erro de execução e levantar as exceções pré-definidas correspondentes; a definição dos pontos em que esses erros podem ocorrer, a geração de código para os testes necessários bem como sua posterior otimização fogem do escopo desse trabalho.

O tamanho da linguagem e sua complexidade nos levou a estudar a possibilidade de basearmos a definição do analisador semântico em algum trabalho que tivesse formalizado a semântica de ADA. A 1.^a opção escolhida para estudos foi a Definição Formal de ADA publicada em novembro de 1980 [INRIA (14)], que teve como principal objetivo servir como um padrão para a linguagem através da definição de um documento no qual programadores e implementadores pudessem resolver suas dúvidas de maneira não ambígua. A metodologia empregada para a especificação

da Definição Formal é conhecida como semântica denotacional [SCOTT et al. (28)], já foi usada para definição de outras linguagens de programação e permite um tratamento unificado tanto para semântica estática quanto para semântica dinâmica; foi usada uma notação derivada de ADA para sua especificação. Infelizmente, alguns problemas impediram que a Definição Formal fosse usada como base para nosso analisador semântico. Primeiramente, o emprego dos conceitos teóricos envolvidos em semântica denotacional acrescentaria um novo grau de complexidade ao trabalho a ser executado. Por outro lado, o fato da Definição Formal não estar completa e a dúvida sobre a eficiência de um analisador semântico baseado em uma metodologia tão pouco atrativa sob o ponto de vista prático provocaram a decisão de não usarmos a Definição Formal de ADA em nosso projeto.

O 2º trabalho de formalização da semântica estática de ADA estudado foi uma gramática de atributos definida pelo grupo da universidade de Karlsruhe [UHL et al. (29)]. Esse trabalho pareceu mais promissor sob o ponto de vista prático, pois essa equipe estava envolvida na implementação de um front-end para ADA e a gramática de atributos foi desenvolvida com o objetivo de ser utilizada no mesmo. Paralelamente, esse grupo tinha acesso ao sistema GAG descrito por KASTENS et al. (26) que, recebendo uma gramática de atributos como entrada, verifica se ela é ordenada e está bem definida e, em caso afirmativo, gera um programa Pascal equivalente. Estimamos que a impossibilidade de usarmos um sistema do tipo do GAG aumentaria bastante a complexidade do desenvolvimento da especificação do analisador semântico através de uma gramática de atributos.

Devido aos fatos mencionados decidimos iniciar a especificação do analisador semântico a partir da definição da linguagem fornecida em seu manual de referência. Deve-se notar que nesse ponto já possuíamos sua nova especificação, conhecida como ADA-82 [DOD (02)], que foi escrita com um nível muito maior de detalhes e na qual algumas ambiguidades existentes na definição original foram eliminadas. BROSGOL (25) apresenta um resumo das principais modificações efetuadas na nova especificação.

5.2. ESTRUTURA DO ANALISADOR SEMÂNTICO

As principais funções do analisador semântico são listadas abaixo.

- Tratamento de declarações / construção da tabela de símbolos: O corpo de uma unidade de programa ADA normalmente possui 2 partes: a 1.^a é a parte declarada na qual todas as entidades usadas no programa são definidas; a 2.^a é uma sequência de comandos que define o processamento da unidade em questão. Uma parte razoável do trabalho do analisador semântico está relacionada com o tratamento das declarações e da construção da tabela de símbolos que deve guardar as várias características dos objetos declarados. Decidimos pela inclusão de novos atributos na árvore DIANA para o armazenamento dessas características e não pela construção da tabela de símbolos independente como proposto por várias implementações tais como as descritas por UHL et al. (29) e QUINN (16). Note que essa decisão é compatível com a existência de uma tabela "hash" externa usada para acelerar o processo de busca de informações.
- Resolução de nomes e expressões: Outra importante função do analisador semântico é identificar o tipo e o significado dos nomes e expressões que aparecem em um programa ADA. Como apresentado no capítulo correspondente, essa tarefa fica mais complexa devido à possibilidade de 2 subprogramas ou funções possuírem o mesmo designador no mesmo escopo.
- Tratamento de expressões estáticas: O cálculo de expressões estáticas muitas vezes é necessário para realização de alguns testes semânticos, conforme mostrado na próxima figura.

CASE X IS

WHEN 1 + 1 => ...

WHEN 2 => ...

Figura V.1 - Necessidade de avaliação de uma expressão estática em tempo de compilação.

No exemplo dado, é necessária a avaliação das expressões do comando para a identificação de um erro pelas regras de semântica estática de ADA (ARM. SC. 5.4). A verificação se determinada expressão é estática pode ser facilmente obtida pela análise de seus operandos. Porém, o cálculo do valor da mesma pode se tornar uma operação bastante mais complexa, principalmente se levarmos em consideração que a aritmética da máquina hospedeira e da máquina alvo podem não ser compatíveis. Nossa definição do analisador semântico baseia-se na expectativa de uma 1ª implementação na qual a máquina hospedeira seja igual à máquina alvo, o que simplifica bastante o cálculo dessas expressões em tempo de compilação.

- Transformação da árvore DIANA: Como já foi apresentado diversas vezes, o principal objetivo do analisador semântico é a produção da árvore DIANA a partir da árvore DIANA sintática. Basicamente isso é obtido pelo cálculo dos atributos semânticos, apresentados no capítulo III.
- Compilação em separado: A construção do ambiente necessário para análise de módulos compilados em separado bem como do gerenciamento da biblioteca de programas e testes de dependência entre seus diversos módulos não são tarefas do analisador semântico; em nosso projeto, todas essas funções são realizadas pelo tamborete (RANGEL et al. (18) e CHAVES (8)).

O analisador semântico foi obtido através da definição de uma rotina para o tratamento de cada nó de DIANA; com isso foi obtido um sistema bem estruturado e de fácil modifica

ção. Foram também empregadas rotinas que tratam um contexto maior (por exemplo, existe uma rotina que verifica se as declarações incompletas foram complementadas no mesmo bloco) ou de uso mais geral (por exemplo, as rotinas que verificam se um determinado identificador já foi declarado ou as que devolvem um ponteiro para o tipo de uma determinada expressão). A maior parte dessas rotinas não foi especificada, mas existe um comentário sobre as funções das mesmas na 1.^a vez que elas aparecem. Uma boa parte dessas funções efetua simplesmente um percurso pela árvore DIANA realizando uma função bem determinada.

Como foi apresentado no início desta seção, a tabela de símbolos foi incorporada na própria árvore DIANA através da definição de novos atributos; isso simplificou bastante a definição do analisador semântico. Acreditamos que as técnicas usuais de tratamento de tabelas de símbolos podem ser facilmente adaptadas para funcionar corretamente para ADA. Em uma futura implementação essa decisão pode ser modificada principalmente para se aumentar a eficiência ao acesso das informações guardadas. Como o enfoque de nosso trabalho foi mais voltado para a parte lógica do analisador semântico não esperamos grandes dificuldades para que essas modificações e/ou adaptações sejam efetuadas.

O fluxo de processamento do analisador semântico é tal que, partindo da raiz da árvore que está sendo analisada, percorre todos os nós da mesma realizando os testes necessários, normalmente na seguinte ordem:

1. Chame a função apropriada para realizar os testes necessários em seus nós filhos.
2. Teste as regras semânticas associadas ao nó. Esse teste provavelmente usará informações provenientes dos nós já analisados.
3. Preencha os atributos semânticos do nó, realize as normalizações necessárias e outros processamentos eventualmente associados ao nó.
4. Retorne para seu nó pai.

Erros no programa fonte influenciam esse fluxo em 2 sentidos. Em 1º lugar, o analisador semântico espera receber a representação de um programa sintaticamente correto. Isso pode ser obtido se o analisador sintático usar um algoritmo de correção de erros como o proposto por ROHRICH (30). Se isso não acontecer a solução a ser empregada é a de só se efetuar a análise semântica de programas sintaticamente corretos, o que restringirá bastante a flexibilidade do sistema. Porém, estimamos que uma pequena modificação no analisador semântico permitirá seu funcionamento mesmo com programas sintaticamente incorretos se o analisador sintático indicar os trechos inválidos dos mesmos. Como era de se esperar, erros semânticos também influenciam o fluxo de processamento do módulo. Em algumas situações, a partir da detecção de um erro, as sub-árvores do nó que está sendo analisado não serão visitadas. Por exemplo, se o tipo da expressão de um comando "CASE" for indeterminado, não é possível analisar as expressões que aparecem em sua sequência de alternativas. Por outro lado, outros erros não impedem que a análise continue nas sub-árvores do nó em que a situação de erro ocorreu. O exemplo mais típico talvez seja o da declaração inválida de um identificador em uma lista de identificadores. De maneira geral, nossa preocupação principal foi a de se evitar a produção de erros em cascata pelo analisador semântico. Embora esperemos que o esquema de recuperação de erros seja funcional, o teste efetivo do mesmo só poderá ser realizado quando de sua implementação.

O final desse capítulo será dedicado ao estudo de alguns pontos considerados interessantes. Para tanto, as próximas subseções serão relacionadas a certos capítulos do manual de referência da linguagem.

5.3. TRATAMENTO DAS DECLARAÇÕES E TIPOS

Esse capítulo do manual de referência descreve os tipos da linguagem e as regras para declarações de constantes e variáveis. As principais sub-árvores introduzidas pelas construções desse capítulo são as dos nós "var", "constant", "number", "type", "subtype" e "item-s". O tratamento de todas essas sub-árvores não apresenta grande novidade. Em primeiro lugar, uma sequência de declarações é sempre marcada em DIANA por um nó "item-s"; a rotina desse nó basicamente percorre as declarações existentes realizando os testes correspondentes; a marcação de início e final da lista de declarações e verificação se os tipos incompletos e especificações de sub-programas foram complementadas são realizadas por rotinas chamadas durante o processamento do nó "block". Os problemas encontrados nos testes das entidades definidas nesse capítulo são comuns a maior parte das linguagens de programação existentes sendo, portanto, resolvidos com as técnicas usuais.

A derivação de tipos pode provocar a derivação implícita de alguns sub-programas que operem sobre os mesmos, como será apresentado na seção 5.6.

As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 3 do manual de referência da linguagem são as seguintes:

"VERIFICA_CONSTANT", "VERIFICA_VAR", "VERIFICA_NUMBER",
 "VERIFICA_TYPE", "VERIFICA_SUBTYPE", "VERIFICA_CONSTRAINED",
 "VERIFICA_DERIVED", "VERIFICA_RANGE", "VERIFICA_LITERAL_S",
 "VERIFICA_INTEGER", "VERIFICA_FLOAT", "VERIFICA_FIXED",
 "VERIFICA_ARRAY", "VERIFICA_RECORD", "VERIFICA_VARIANT_PART",
 "VERIFICA_VARIANT", "VERIFICA_INNER_RECORD", e
 "VERIFICA_ACCESS".

5.4. NOMES E EXPRESSÕES

Esse capítulo do manual de referência trata das regras aplicadas às diferentes formas de nomes e expressões e das respectivas avaliações. Essas regras apresentam novidades se comparadas com as regras correspondentes de linguagens tradicionais, devido principalmente ao conceito de sobreposição de nomes ("*overloading*"). As técnicas utilizadas para o tratamento dessas regras foram apresentadas em capítulo a parte devido à importância das mesmas. Nesse ponto somente vale apenas destacar a dificuldade encontrada para o tratamento de agregados, pois a flexibilidade oferecida por essa construção teve como contra-partida um grau de complexidade bastante elevado para sua análise. Vejamos o exemplo dado na próxima figura.

```

TYPE ENUM_A IS (A1, A2);
TYPE ENUM_B IS (B1, B2);
TYPE REC (D1: ENUM_A; D2: ENUM_B) IS
RECORD
  CASE D1 IS
    WHEN A1 =>
      C1: INTEGER;
    WHEN A2 =>
      C2: BOOLEAN;
  END CASE;
  CASE D2 IS
    WHEN B1 =>
      C3: INTEGER;
    WHEN B2 =>
      C4: BOOLEAN;
  END CASE;
END RECORD;

```

Figura V.2 - Exemplo de um record com partes variantes.

Pode-se constatar que são necessários 2 passos sobre o agregado mostrado na figura V.3 para a detecção de seu erro.

(C1 => 5, C4 => BOOLEAN, D1 => A1, D2 => B1);

Figura V.3 - Exemplo de um agregado incorreto para o tipo anteriormente definido.

O fato da linguagem não permitir que o tipo dos elementos de 1 agregado sejam usados para determinação do tipo do próprio agregado simplificou bastante a rotina de determinação do tipo de nomes, principalmente no caso de agregados internos a outros agregados. Porém acreditamos que o fato da linguagem permitir associação por nome, por posição e mesmo por uma mistura entre os dois, além de não auxiliar a construção de programas mais claros, aumenta bastante a complexidade do analisador semântico.

Como foi apresentado na seção 3.4.1 desse trabalho, algumas ambiguidades da gramática de ADA provocam a necessidade de modificações na árvore sintática DIANA durante a fase de análise semântica. O exemplo mais típico dessa situação é o tratamento de componentes indexadas, conversões, "slices" e chamadas de subprogramas. Nesses casos o analisador sintático produz uma árvore padrão como mostrado na figura V.4.

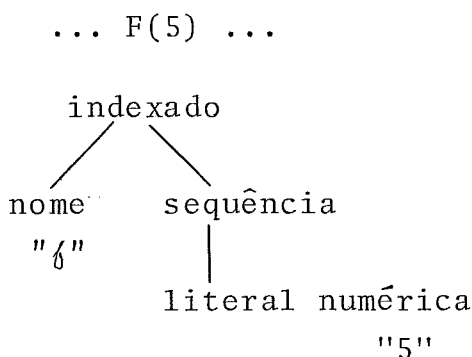
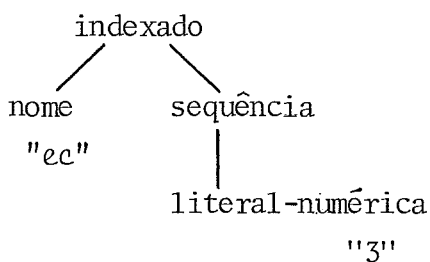


Figura V.4 - Sub-árvore gerada pelo analisador sintático para componente indexado.

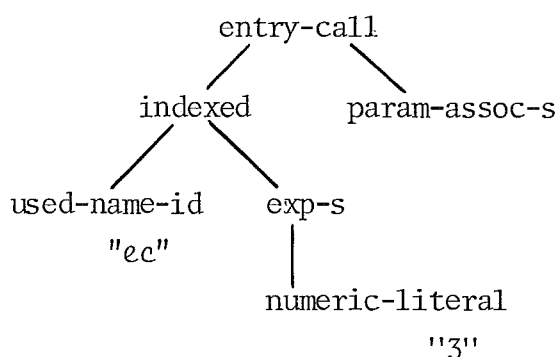
Normalmente a renomeação de alguns nós da árvore é suficiente, não havendo necessidade de modificações na sua estrutura. Um exemplo em que essa 2.^a hipótese ocorre é dado na figu

ra V.5, na qual a sub-árvore modificada é estruturalmente diferente da sub-árvore original.

```
ENTRY EC(1 .. 5);
:
ACCEPT EC(3);
```



(a)



(b)

Figura V.5 - (a) Árvore gerada pelo analisador sintático.

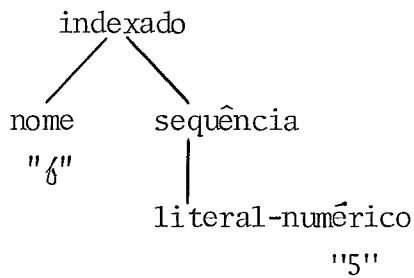
(b) Árvore modificada pelo analisador semântico.

Note que o processo de transformação da árvore apresentada no exemplo não é trivial, pois teve que levar em consideração o fato do componente referenciar um elemento de uma família de entradas ("*entry*") sem parâmetros. Porém, algumas construções permitem várias interpretações para uma única sub-árvore, como apresentado na figura V.6, na qual pode-se observar 2 sub-árvores que representam construções corretas sob o ponto de vista sintático; cabe ao analisador semântico verificar qual das 2 é válida.

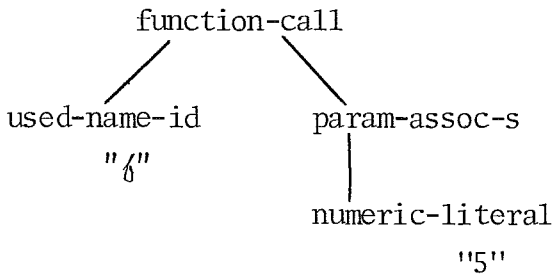
```

TYPE VET IS ARRAY (1 .. 10) OF INTEGER;
FUNCTION F (P1: INTEGER := 10) RETURN VET IS
    TEMP: VET := (1,2,3,4,5,6,7,8,9,10);
BEGIN
    RETURN TEMP;
END;
:
V: INTEGER
V:= F(5);

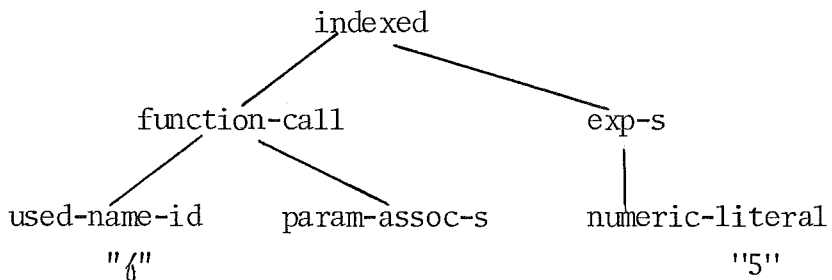
```



(a)



(b)



(c)

Figura V.6 - (a) Árvore gerada pelo analisador sintático.

(b) Interpretação incorreta.

(c) Interpretação correta.

Se no exemplo dado o literal for considerado como parâmetro da função estaria se tentando atribuir um objeto do tipo "VET" a uma variável do tipo inteiro; se por outro lado considerar-se a chamada da função utilizando seu parâmetro default e o literal como um índice do valor devolvido o comando de atribuição estaria válido.

Qualquer função que tenha todos seus parâmetros "default" e que devolva um elemento de um tipo array é digna de ser considerada como uma dor de cabeça potencial para o analisador semântico durante as modificações das sub-árvores de suas chamadas, pois nesses pontos o analisador semântico deve gerar e analisar todas as sub-árvores sintaticamente válidas para a construção em questão. O exemplo apresentado por HERZOG (27) e mostrado na próxima figura demonstra uma construção para qual 2 interpretações são semanticamente válidas, caracterizando uma situação ambígua.

```

PROCEDURE F IS
  TYPE ARR;
  TYPE ACC IS ACCESS ARR;
  TYPE ARR IS ARRAY (1 .. 10) OF ACC;
  X: ACC;
  FUNCTION F(X: INTEGER:= 0) RETURN ACC IS
  BEGIN
    RETURN NEW ARR;
  END;
BEGIN
  X:= F(1);
END

```

Figura V.7 - Exemplo de uma construção ambígua: F(1).

Na definição inicial de ADA chamadas de funções sem parâmetros eram representadas com o símbolo "()", o que eliminava boa parte das dificuldades expostas acima.

A solução desse problema é bastante complexa, já que somente o 2º passo da rotina de resolução de nomes detecta situações de erro. Dessa forma, sempre que uma expressão contiver

uma construção que admita mais que uma transformação válida, a expressão deve ser analisada para cada uma de suas interpretações.

5.5. COMANDOS

Esse capítulo do manual de referência da linguagem discute seus comandos de uso geral, exceto os relacionados a chamadas de sub-programas, processamento concorrente e exceções, que são discutidos à parte em capítulos específicos. Os problemas encontrados para o tratamento dos comandos de ADA também são gerais à grande parte das linguagens de programação, porém alguns pontos merecem destaque.

Alguns comandos só podem aparecer em alguns contextos específicos. Por exemplo, o comando "RETURN" só pode ocorrer dentro do corpo de um sub-programa, de um sub-programa genérico, ou de um comando "ACCEPT" mas não deve aparecer dentro de um comando composto englobado por essas construções (ARM. Sc 5.8.3). Dessa forma, cada vez que o nó representativo de uma dessas construções é encontrado ("block", "loop", ...) é colocado um elemento em uma pilha de contexto, sendo esse retirado no final da análise da construção em questão, assim, sempre se pode verificar em que ponto determinada construção se encontra.

O escopo do parâmetro de um comando "LOOP" vai desde sua especificação até o final do comando, de forma que ele é visível somente na sequência de comandos do loop. Dessa forma, quando um comando "LOOP" é encontrado, além do fato ser marcado na pilha de contexto indicada acima, se o comando tiver um parâmetro esse passa a fazer parte do ambiente da unidade de compilação (tabela de símbolos).

Cada expressão em uma alternativa de um comando "CASE" deve ser estática e ter o mesmo tipo que a expressão do comando, que deve ser discreto. Além disso, cada valor desse tipo deve ser representado uma e só uma vez na sequência de alternativas do comando. Isso exige que a avaliação das expressões das alternativas seja realizada em tempo de compilação, o que causa alguns problemas já apresentados nesse capítulo.

As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 5 do manual de referência da linguagem são as seguintes:

"VERIFICA_LABELED", "VERIFICA_ASSIGN", "VERIFICA_IF",
"VERIFICA_COND_CLAUSE", "VERIFICA_CASE", "VERIFICA_ALTERNATIVE",
"VERIFICA_LOOP", "VERIFICA_FOR", "VERIFICA_REVERSE",
"VERIFICA_BLOCK", "VERIFICA_EXIT", "VERIFICA_RETURN" e
"VERIFICA_GOTO".

5.6. SUBPROGRAMAS

Esse capítulo do manual de referência trata da definição de subprogramas (procedures e funções). Algumas características de ADA nesse ponto a diferenciam de outras linguagens de programação, principalmente a sobreposição de nomes ("overloading"), parâmetros de entrada "default", e possibilidade de associação dos parâmetros por posição ou por nome.

As técnicas utilizadas para identificação de um subprograma quando de sua chamada já foram apresentadas em um capítulo anterior; nessa seção comentaremos brevemente as dificuldades e ambiguidades que podem surgir com o emprego de parâmetros default e as regras usadas para determinação se 2 subprogramas declarados com um mesmo nome em uma mesma parte declarativa são válidos. Consideremos, por exemplo, os subprogramas definidos na figura V.8.

```

PROCEDURE PROC (P1: INTEGER);
PROCEDURE PROC (P2: INTEGER; P3: BOOLEAN: = FALSE);
:
PROC (5, FALSE);
PROC (5);
PROC (P1 => 5);

```

Figura V.8 - Exemplo de especificação de sub-programas sobrepostos e algumas chamadas aos mesmos.

Pela análise dos tipos dos parâmetros usados nas chamadas podemos verificar que a 1.^a chamada é válida e se refere ao 2.^o subprograma. A 2.^a chamada é ambígua, pois pode se referenciar tanto ao 1.^o sub-programa quando ao 2.^o; na 3.^a chamada foi utilizada associação por nome para que a ambiguidade fosse resolvida. Note que se o nome dos 1.^{os} parâmetros dos 2 sub-programas fossem iguais qualquer chamada ao 1.^o seria ambígua. Embora não adicionem nenhuma complexidade adicional ao algoritmo de resolução de nomes, situações desse tipo podem tornar o emprego da linguagem incômodo.

Outro aspecto importante da definição da linguagem é a derivação de sub-programas devido a derivação do tipo de pelo

menos um de seus parâmetros; o manual de referência apresenta as condições nas quais as derivações podem ocorrer (ARM. Sc. 3.4.11). Como DIANA não cria uma ocorrência explícita para o sub-programa derivado, duas ações devem ser tomadas. Em primeiro lugar, sempre que um tipo é derivado verifica-se se algum sub-programa que possua um parâmetro desse tipo é automaticamente derivado; pelas regras da linguagem basta verificar os sub-programas definidos na parte visível do pacote no qual o tipo foi definido. Em 2º lugar, durante a fase de resolução de chamadas de sub-programas deve-se também considerar se o sub-programa analisado pode ter sido derivado.

Dois cuidados adicionais devem ser tomados pelas rotinas que tratam os valores iniciais de parâmetros default. Primeiramente, as regras sintáticas apresentadas no manual de referência não impedem o uso de valores iniciais para parâmetros do tipo "in out" ou "out"; esse teste pode ser facilmente realizado pelo analisador semântico. Por outro lado, o uso de um nome que denote um parâmetro formal não é permitido em uma expressão inicial de uma parte formal se a especificação do parâmetro tiver sido fornecida na mesma parte formal. Para a realização desse teste deve ser criado um novo atributo que permita controlar o acesso a entidades que, embora já definidas, não podem ser acessadas, como no presente caso.

O teste necessário para verificação se o corpo de determinada sub-rotina é equivalente a uma especificação anteriormente fornecida pode ser facilmente implementado por uma rotina que compare as partes formais de ambos. Os problemas resultantes da possibilidade de compilação em separado já foram apresentados no capítulo sobre DIANA.

As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 6 do manual da linguagem, além das apresentadas no apêndice das rotinas de resolução de tipo e expressão, são as seguintes:

"VERIFICA_PROCEDURE", "VERIFICA_FUNCTION", "VERIFICA_IN",
 "VERIFICA_IN_OUT", "VERIFICA_OUT", "VERIFICA_SUBPROGRAM_BODY",
 "VERIFICA_PROCEDURE_CALL", "VERIFICA_FUNCTION_CALL" e
 "VERIFICA_ASSOC".

5.7. PACKAGES

Esse capítulo do manual de referência trata da definição de pacotes ("*packages*"). Embora este seja um conceito novo (e importante) de ADA, a análise das estruturas correspondentes não apresenta muitos problemas sob o ponto de vista da análise semântica. A divisão entre a especificação e o corpo de pacotes e a possibilidade de ocorrerem em unidades de compilação distintas existe também com sub-programas. A existência de tipos privados ou limitados e de constantes postergadas ("*deferred constants*") levanta novamente a necessidade de testes de equivalência para entidades que são declaradas em mais de um ponto do programa, o que não chega a ser problemático.

O fato da declaração de tipo privado e da declaração completa correspondente definirem 2 vistas de um único tipo tem várias implicações para o processo de análise semântica. Fora do pacote que define o tipo privado as características do mesmo são as definidas na parte visível correspondente; nesses pontos as regras da linguagem relativas a tipos não se aplicam ao tipo privado, porém dentro do pacote no qual ele foi definido ele é considerado um tipo comum. A solução empregada para essa situação foi fortemente influenciada pela estrutura de DIANA, na qual um nó do tipo "*package-spec*" possui 2 listas de declarações, a primeira relacionada com as entidades especificadas na parte privada e a 2^a com as entidades da parte visível, conforme apresentado na próxima figura. Como se pode observar nesta, o acesso ao nó "*private-type-id*" permite identificar em 1^o lugar o acesso a um tipo privado (pelo próprio tipo do nó) e em 2^o lugar o acesso à sua estrutura, através do atributo semântico "*sm-type-spec*". Além disso, ao se analisar o corpo de determinado pacote é necessário que os tipos privados declarados na especificação do mesmo sejam marcados, uma vez que somente as estruturas desses se tornam visíveis. Isso pode ser facilmente conseguido com o emprego de um novo atributo de semântica estática usado nas entidades que podem ser declaradas na parte visível de um pacote; a rotina que trata o corpo de um pacote (nó "*package-body*") liga esses atributos nos nós das entidades declaradas na parte visível do pacote quando for cha

mada e os desliga antes de terminar sua execução.

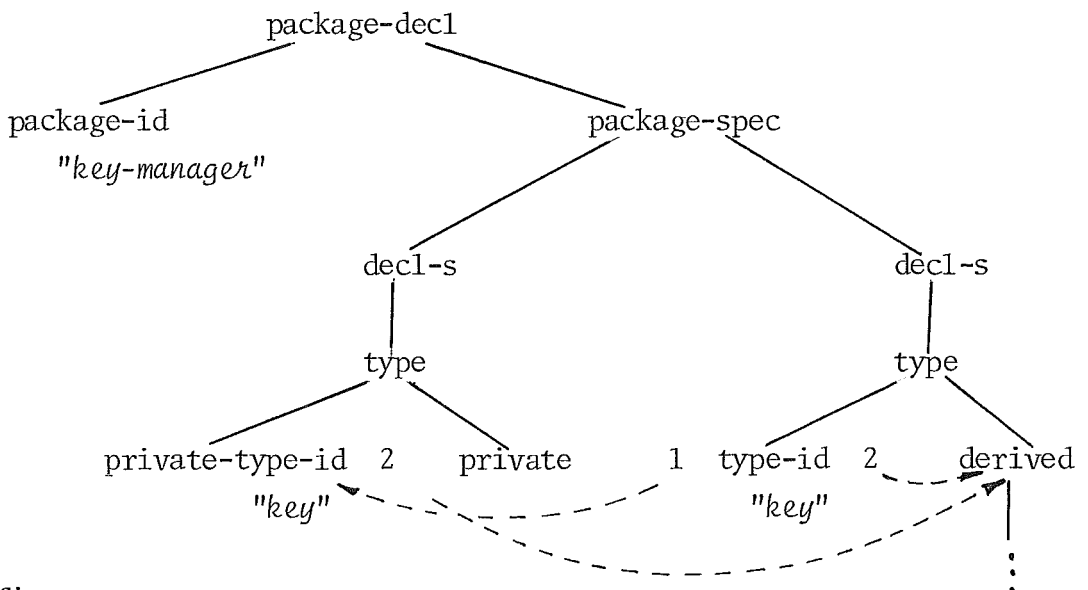
As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 7 do manual de referência da linguagem são as seguintes:

"VERIFICA_PACKAGE_DECL", "VERIFICA_PACKAGE_SPEC",
 "VERIFICA_PACKAGE_BODY", "VERIFICA_PRIVATE",
 "VERIFICA_L_PRIVATE" e "VERIFICA_DEFERRED_CONSTANT".

```

PACKAGE KEY_MANAGER IS
  TYPE KEY IS PRIVATE;
PRIVATE
  TYPE KEY IS NEW INTEGER;
END;

PACKAGE BODY KEY_MANAGER IS
  LAST_KEY := Ø
END;
  
```



1: sm_firt

2: sm_type_spec

Figura V.9 - Exemplo de um "package" com respectiva representação DIANA.

5.8. REGRAS DE VISIBILIDADE

Esse capítulo trata das regras que especificam o escopo de declarações e das que definem os identificadores visíveis em vários pontos de um programa. As principais dificuldades impostas por essas regras são devido a possibilidade de separação entre a especificação e o corpo de pacotes, sobreposição de nomes para designadores de sub-programas e literais de enumeração e existência da cláusula "USE". Devido a essas dificuldades, o manual de referência é bastante prolixo para definição das regras de visibilidade, reservando um capítulo para as mesmas. De forma geral acreditamos que, apesar dos problemas mencionados, as técnicas clássicas de implementação de linguagens de programação para o tratamento de visibilidade podem ser utilizadas com pequenas modificações para ADA.

Um ponto que merece destaque é o tratamento da cláusula "USE", que permite que entidades declaradas nas partes visíveis dos pacotes especificados possam ser diretamente acessadas, sem necessitar o uso de expressões qualificadoras. Se ela aparecer na cláusula de contexto de uma unidade de compilação, ela é válida para toda a unidade; por outro lado, se ela aparecer em uma parte declarativa ("*declaration-part*") ela é válida até o final da mesma. Isso implica na necessidade de um mecanismo eficiente para identificação das entidades atingidas por uma cláusula "USE". Em nosso projeto essa identificação foi obtida através do uso de um novo atributo de semântica estática: sempre que o analisador semântico passa por um nó do tipo "USE" ele percorre as entidades declaradas pelas unidades de compilação especificadas ligando o valor desse atributo; quando o analisador semântico termina a análise da entidade global à parte declarativa (comando bloco, corpo de sub-programa, de pacote ou de task) essa lista novamente é percorrida para se desligar esse atributo.

As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 8 do manual de referência da linguagem são as seguintes:

"VERIFICA_USE" e "VERIFICA_RENAME".

5.9. PROCESSAMENTO PARALELO

Esse capítulo do manual de referência trata da declaração e comandos de controle de "*tasks*".

Os mecanismos de ativação e sincronismo de processos concorrentes possuem uma semântica dinâmica bastante complexa e que vem sendo bastante questionada, principalmente no que diz respeito a seu emprego em arquiteturas de multiprocessadores [SEGRE (31)]. Porém, a semântica estática dos mesmos é bastante simples, não oferecendo problemas muito diferentes dos encontrados para o tratamento de sub-programas - separação entre a especificação de uma task e seu corpo e possibilidade de sobreposição do nome de suas entradas ("*entry*") com outros sub-programas ou literais de enumeração.

As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 9 do manual de referência da linguagem são as seguintes:

"VERIFICA_TASK_DECL", "VERIFICA_TASK_SPEC",
 "VERIFICA_TASK_BODY", "VERIFICA_ENTRY", "VERIFICA_ENTRY_CALL",
 "VERIFICA_ACCEPT", "VERIFICA_DELAY", "VERIFICA_SELECT",
 "VERIFICA_SELECT_CLAUSE", "VERIFICA_COND_ENTRY",
 "VERIFICA_TIMED_ENTRY" e "VERIFICA_ABORT".

5.10. ESTRUTURA DE PROGRAMAS

Esse capítulo do manual de referência trata das regras relacionadas com compilação em separado. Como já foi visto, os principais problemas relacionados com esse assunto, tais como criação de bibliotecas de programas, inclusão e exclusão de unidades de compilação em bibliotecas, ordem de compilação entre unidades de determinado programa, manipulação de versões de determinadas unidades, entre outros, são resolvidos por um módulo independente - TAMBORETE - ou de um sistema mais completo que venha a substituí-lo. Em particular, quando da compilação de determinada unidade, a interface com esse módulo deve criar o ambiente necessário (tabela de símbolos representativa das entidades declaradas nos módulos que são acessados).

5.11. TRATAMENTO DE EXCEÇÕES

Esse capítulo do manual de referência define as facilidades para o tratamento de situações de exceções que ocorrem durante a execução de programas. Tal como ocorre com tasks, os principais problemas relacionados a essas facilidades são relativos à semântica dinâmica das mesmas, não oferecendo dificuldades para sob o ponto de vista de semântica estática.

5.12. UNIDADES GENÉRICAS

Esse capítulo do manual de referência trata de unidades genéricas: sub-programas genéricos ou pacotes genéricos. Embora seja um conceito novo e poderoso, o tratamento de unidades genéricas não aumenta a complexidade do analisador semântico, uma vez que alguns dos problemas inerentes a esse conceito (separação entre a especificação e corpo, teste de perfil de tipos de parâmetros, etc ...) existem para o tratamento de outras construções da linguagem. As regras que controlam a instanciação de unidades genéricas, embora numerosas, são bastante simples. (Existem inúmeras regras que restringem o uso de tipos genéricos formais, tais como a que impede que uma cláusula de representação seja empregada para tipos dessa classe; isso aumenta o tamanho, mas não a complexidade do analisador). Note que, de acordo com a semântica da linguagem, se a especificação e o corpo de uma unidade genérica estiverem corretos assim como a definição de seus parâmetros reais, a instanciação da mesma estará necessariamente correta. Dessa forma ela não precisa ser realizada pelo front-end do compilador, podendo ficar a cargo do back-end do mesmo. Se isso acontecer e se houver mais que uma instanciação de uma única unidade genérica, pode-se tentar otimizar o código gerado pelo compartilhamento de partes do mesmo por várias instâncias. Porém, as características da máquina alvo e principalmente a representação de objetos na mesma devem ser conhecidas nesse ponto.

As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 12 do manual da linguagem são as seguintes:

"VERIFICA_GENERIC", "VERIFICA_GENERIC_ASSOC_S", e
"VERIFICA_INSTANTIATION".

5.1.3 CLÁUSULAS DE REPRESENTAÇÃO E ASPECTOS DEPENDENTES DE MÁQUINA

Esse capítulo do manual de referência trata dos aspectos da linguagem mais relacionados com a representação de objetos nas máquinas alvo; obviamente esse é um assunto bastante delicado, principalmente quando observamos que um dos objetivos do projeto de ADA foi a produção de programas altamente portáteis, com um elevado grau de independência em relação a máquinas específicas. O próprio manual de referência da linguagem indica que uma implementação pode limitar a aceitação de cláusulas de representação aquelas que possam ser tratadas por uma máquina específica (ARM. Sc. 13.1.10). Abaixo listaremos alguns problemas encontrados para o tratamento dessas construções.

Uma cláusula de representação de tipo indica como de terminado tipo deve ser mapeado em determinada máquina; no caso da inexistência dessa cláusula para determinado tipo, um valor inicial é fornecido pela implementação. Porém, alguns usos de um identificador de tipo implicam que a representação do mesmo já deve ter sido determinada; logo, se existir uma cláusula de representação para o tipo ela deve aparecer antes da utilização do mesmo. Em nossa implementação esse controle foi simplificado de forma que a partir da 1^a vez que determinado tipo for usado ficam inválidas cláusulas de representação para o mesmo. Note que os valores padrão podem variar entre diversas máquinas. Sua atribuição bem como a verificação se valores fornecidos por usuários são válidos são tarefas que devem ser efetuadas pelo back-end do compilador.

Para o tratamento das cláusulas que especificam o tamanho (número de bits) a ser alocado a objetos de determinados tipos é necessário que a expressão (estática) fornecida na mesma seja avaliada para se verificar se está sendo reservado espaço suficiente para o armazenamento de todos os valores permitidos para o tipo.

Uma cláusula de representação de records indica o lay-out para o armazenamento de seus objetos. No máximo é permitida uma cláusula por componente, porém não é necessário que

cada componente tenha uma cláusula específica. A sobreposição de componentes de uma mesma parte variante é proibida, sendo permitida para componentes de partes variantes distintas. Novamente, a realização desse teste pode necessitar o conhecimento da máquina alvo para qual o código está sendo gerado.

A inserção de código de máquina é realizada através da chamada a uma rotina que só contenha comandos de código ("*code-statement*"); nesse comando, cada instrução é representada como um agregado de record de um tipo que a represente. Devido ao fato de que uma rotina que contenha um comando de código só deve conter comandos desse tipo e que sua parte declarativa só pode possuir cláusulas "USE" há necessidade de um controle que, embora seja simples, pode-se tornar ineficiente.

As principais rotinas empregadas para a análise das construções introduzidas pelo capítulo 13 do manual de referência da linguagem são as seguintes:

"VERIFICA_SIMPLE_REP", "VERIFICA_RECORD_REP",
"VERIFICA_COMP_REP", "VERIFICA_ADDRESS" e "VERIFICA_CODE".

CAPÍTULO VICONCLUSÕES

Procuramos ao longo desse trabalho descrever uma especificação inicial de um módulo de análise semântica estática para linguagem ADA a ser utilizado no projeto atualmente em desenvolvimento no Programa de Engenharia de Sistemas e Computação da COPPE. Esse objetivo foi atingido primordialmente através da definição das rotinas de resolução de nomes e expressões e das rotinas semânticas definidas para cada nó da árvore DIANA, forma intermediária escolhida em nosso projeto. Com isso esperamos ter obtido um sistema bastante modular.

Como deve ter ficado patente nos capítulos anteriores, faltam alguns pontos concretos para que um programa possa ser obtido a partir da especificação, entre os quais devemos destacar a definição física de DIANA (consideramos que, antes de se obter um sistema de suporte à programação ADA completo, ser esta definição bastante dependente das máquinas nas quais o compilador deve rodar), a definição concreta com o resto do ambiente (o "tamborete" em nosso projeto), a definição de rotinas auxiliares que são usadas por várias rotinas do analisador semântico ou que tenham uma função bastante específica e a definição do módulo responsável pelo cálculo das expressões aritméticas estáticas. De forma geral não esperamos grandes problemas para a conclusão dessas tarefas.

Outro ponto que merece comentários foi a utilização da própria linguagem ADA para a especificação do módulo. Essa decisão, tomada no início de nosso projeto, foi motivada principalmente pelo desconhecimento da máquina que seria utilizada para sua implementação. Basicamente, existem 3 possibilidades para a implementação concreta do módulo.

A 1.^a consiste na implementação de um compilador para um sub-conjunto de ADA que seja suficientemente poderoso para compilar a especificação do módulo. Embora durante o desenvolvimento da especificação não nos tenhamos preocupado com a even

tual adoção de um sub-conjunto de ADA a ser usado na fase de implementação, podemos observar que vários aspectos de difícil implementação da linguagem tais como tratamento de exceções e utilização de unidades genéricas não foram usados.

A 2.^a opção consiste na implementação de um tradutor de um sub-conjunto de ADA para Pascal e depende da existência de um compilador dessa linguagem para implementação do sistema. Esse tradutor, que teria sua principal função no desenvolvimento, poderia ter seus critérios de eficiência e confiabilidade um pouco relaxados para se obter um sistema em um tempo menor, porém necessitaria de um módulo que fizesse o casamento entre a semântica das 2 linguagens. Esperamos que um tradutor desse tipo seja mais rapidamente obtido do que um compilador completo para o mesmo sub-conjunto considerado.

Finalmente a última opção consiste na codificação manual do módulo na linguagem escolhida para sua implementação. Essa opção acarreta o risco de se ter em pouco tempo a especificação do módulo, escrita em ADA, totalmente diferente da versão que estiver sendo realmente implantada.

Obviamente, a eficiência do sistema e seu comportamento quando da ocorrência de erros semânticos só poderão ser analisados após sua implementação. Esperamos alguns problemas para se obter todas facilidades para um sistema ADA se o mesmo tiver que ser implementado em uma máquina do porte de 16 bits, o que nos leva a crer que tal sistema seja indicado para máquinas maiores, possivelmente de 32 bits.

Vale a pena ressaltar que nossa principal preocupação durante o desenvolvimento deste trabalho foi com o aspecto funcional do módulo, ficando alguns detalhes sobre a eficiência do mesmo postergados para a fase de implementação; dessa forma esperamos que a especificação apresentada sirva basicamente como um guia para uma possível implementação do módulo.

Para finalizar devemos acrescentar que alguns detalhes do módulo não foram apresentados não só para não aumentar demasiadamente o tamanho dessa dissertação de mestrado, mas também porque acreditamos que alguns dos detalhes mencionados só poderão ser plenamente identificados na época da implementação do módulo.

REFERÊNCIAS BIBLIOGRÁFICAS

- (01) DOD; "ADA Reference Manual", Proposed Standard Document, United States Department of Defense, julho, 1980.
- (02) DOD, "Military Standard - ADA Programming Language", ANSI/MIL-STD-1815 A, fevereiro, 1983.
- (03) GOOS, G., WULF, Wm.A., (Editors), "DIANA Reference Manual", Department of Computer Science, CMU-CS-81-101, março, 1981.
- (04) EVANS, A., BUTLER, K.J. (Editores), "Revised DIANA Reference Manual", Tartan Laboratories Incorporated, fevereiro, 1983.
- (05) MORRIL, D.E., JAMES, C.L., "The Real ADA, Countess of Lovelace", ACM Sigsoft Engineering Notes, vol.8, nº1, janeiro, 1983.
- (06) WEGNER, P., "Emperors, Generals and Programmers: Reflection on the ADA Controversy", Communications of the ACM, vol.25, nº 1, janeiro, 1982.
- (07) ROMANOWSKY, H., "ADA Publications", ACM ADA Letters, vol.2, nº 6, maio-junho, 1983.
- (08) CHAVES, V.B., tese M.Sc., COPPE/UFRJ, (em preparação).
- (09) MELLO, R.L.F.V., tese M.Sc., COPPE/UFRJ, (em preparação).
- (10) TRINDADE, J.T.P., tese M.Sc., COPPE/UFRJ, (em preparação).
- (11) — , "Matrix of ADA Language Implementation", ACM ADA Letters, vol.3, nº 2, outubro, 1983.

- (12) DAUSMANN, M., DROSSOPOULOU, S., GOOS, G., PERSCH, G., WINTERSTEIN, G., 'AIDA Introduction and User Manual', Technical Report Nr 38/80, Institut fuer Informatik II, Universitaet Karlsruhe, 1980.
- (13) BROSGOL, B.M., NEWCOMER, J.M., LAMB, D.A., LEVINE, D., VAN DEUSEN, M.S., WULF, W.A., "TCOL_{ADA}: Revised Report on An Intermediate Representation for the Preliminary ADA Language", Technical Report CMU-CS-80-105, Carnegie-Mellon University, Computer Science Department, fevereiro, 1980.
- (14) INRIA, "Formal Definition of the ADA Programming Language", INRIA, novembro, 1980.
- (15) PAYTON, T.F., "Summary of the Winter ADATEC Meeting, February, 1983", ACM ADA Letters, vol.2, n° 6, julho, 1983.
- (16) QUINN, M.E., "The ADA Bread Board Compiler: The DIANA Package", Bell Laboratories, 1983.
- (17) NESTOR, J.R., WULF, W.A., LAMB, D.A., "IDL - Interface Description Language: Formal Description", Technical Report CMU-CS-81-139, Carnegie-Mellon University, Computer Science Department, agosto, 1981.
- (18) RANGEL, J.L., ARGOLLO JR. M., CHAVES, V.B., "Aspectos de Compilação em Separado", Anais do III Congresso da Sociedade Brasileira de Computação, Campinas, julho, 1983.
- (19) ICHBIAH, J.D., BARNES, J.G.P., HELIARD, J.C., KRIEG-BRUECKENER, B., ROUBINE, O., WICHMANN, B.A., "Rationale for the Design of the ADA Programming Language", ACM SIGPLAN Notices, vol.14. n° 6. julho, 1979.

- (20) PERSCH, G., WINTERSTEIN, G., DAUSMANN, M., DROSSOPOULOU, S., "Overloading in Preliminary ADA", ACM SIGPLAN, Symposium on the ADA Programming Language, novembro, 1980.
- (21) BAKER, T.P., "A One-Pass Algorithm for Overload Resolution in ADA", ACM TOPLAS, vol.4, n° 4, outubro, 1982.
- (22) GANZINGER, H., HIPKEN, K., "Operator Identification in ADA: Formal Specification, Complexity, and Concrete Implementation" ACM SIGPLAN Notices, Vol.15, n° 2, fevereiro, 1980.
- (23) PENELLO, T., DE REMER, F., MEYERS, R., "A Simplified Operator Identification Scheme for ADA", ACM SIGPLAN Notices, vol.15, n° 7, julho, 1980.
- (24) McKEEMAN, W.M., "Compiler Construction", em "Compiler Construction - An Advanced Course", Bauer, F.L. e Eickel, J. (editores), Springer-Verlag, 1976.
- (25) BROSGOL, M.B., "Summary of ADA Language Changes", ACM ADA Letters, vol.I, n° 3.
- (26) KASTENS, U., HUTT, B., ZIMMERMANN, E., "GAG: A Practical Compiler Generator", Lecture Notes in Computer Science, n° 141, Springer Verlag, 1982.
- (27) HERZOG, W., "More on ADA Language Type Checking Problems", ACM ADA Letters, vol. III, n° 3, dezembro, 1983.
- (28) SCOTT, D.S., STRACHEY, C., "Towards a Mathematical Semantics for Programming Language", Polytechnics Press, Brooklyn, New York, 1971.
- (29) UHL, J., DROSSOPOULOU, S., PERSCH, G., DAUSMANN, M., WINTERSTEIN, G., KIRCHGASSNER, W., "An Attribute Grammar for the Semantic Analysis of ADA", Lecture

Notes in Computer Science, n° 139, Springer-Verlag, 1982.

- (30) ROHRICH, J., "Methods for the Automatic Construction of Errors Correcting Parsers", Acta Informática, vol.13, 1980.
- (31) SEGRE, L., "Alguns Problemas em Relação ao uso e Implementação da Linguagem ADA para Sistemas Distribuídos", Anais do III Simpósio sobre Desenvolvimento de Software Básico para Micros, Rio de Janeiro, dezembro, 1983.

APÊNDICE 1

SUMÁRIO DE DIANA

Esse apêndice foi obtido do manual de referência de DIANA, preparado por Tartan Laboratories Incorporated.

APPENDIX IV
DIANA SUMMARY

This appendix contains a list of all the class and node definitions sorted by the name of the class or node. Class definitions are given first; all class names are upper case. Node definitions follow; node names are lower case. With each definition is listed the section number and page number within Chapter 2 where the corresponding concrete syntax can be found.

ACTUAL ::= EXP;	6.4	61
ALIGNMENT ::= alignment;	13.4.A	74
ALTERNATIVE ::= alternative pragma;	5.4	53
ALTERNATIVE_S ::= alternative_s;	5.4	53
ARGUMENT ::= argument_id;	App. I	76
BINARY_OP ::= SHORT_CIRCUIT_OP;	4.4.A	48
BLOCK_STUB ::= block;	6.3	60
BLOCK_STUB ::= stub;	10.2.B	70
BLOCK_STUB_VOID ::= block stub void;	9.1.A	65
CHOICE ::= EXP DSCRT_RANGE others;	3.7.3.B	43
CHOICE_S ::= choice_s;	3.7.3.A	43
COMP ::= pragma;	3.7.B	41
COMP ::= var variant_part null_comp;	3.7.B	41
COMPILATION ::= compilation;	10.1.A	69
COMP_ASSOC ::= named EXP;	4.3.B	47
COMP_REP ::= comp_rep;	13.4.B	75
COMP_REP ::= pragma;	13.4.B	75
COMP_REP_S ::= comp_rep_s;	13.4.B	75
COMP_REP_VOID ::= COMP_REP void;	3.7.B	41
COMP_UNIT ::= comp_unit;	10.1.B	69
COND_CLAUSE ::= cond_clause;	5.3.A	53
CONSTRAINED ::= constrained;	3.3.2.B	36
CONSTRAINT ::= RANGE float fixed dscrt_range_s dscrt_aggregate;	3.3.2.C	37
CONSTRAINT ::= void;	3.3.2.B	36
CONTEXT ::= context;	10.1.1.A	69
CONTEXT_ELEM ::= pragma;	10.1.B	69
CONTEXT_ELEM ::= use;	10.1.1.A	69
CONTEXT_ELEM ::= with;	10.1.1.B	70
DECL ::= REP use;	3.9.A	44
DECL ::= constant var number type subtype subprogram_decl package_decl task_decl generic exception deferred_constant;	3.1	33
DECL ::= pragma;	3.1	33

DECL_S ::= decl_s;	7.1.B	62
DEF_CHAR ::= def_char;	3.5.1.B	38
DEF_ID ::= attr_id pragma_id ARGUMENT;	App. I	76
DEF_ID ::= comp_id;	3.7.B	41
DEF_ID ::= const_id;	3.2.A	34
DEF_ID ::= dscrm_t_id;	3.7.1	42
DEF_ID ::= entry_id;	9.5.A	66
DEF_ID ::= enum_id;	3.5.1.B	38
DEF_ID ::= exception_id;	11.1	70
DEF_ID ::= function_id;	6.1.A	57
DEF_ID ::= generic_id;	12.1.A	71
DEF_ID ::= in_id;	6.1.C	59
DEF_ID ::= in_out_id out_id;	6.1.C	59
DEF_ID ::= iteration_id;	5.5.B	55
DEF_ID ::= label_id;	5.1.B	51
DEF_ID ::= named_stm_id;	5.5.A	54
DEF_ID ::= number_id;	3.2.B	35
DEF_ID ::= package_id;	7.1.A	62
DEF_ID ::= private_type_id _private_type_id;	7.4.A	63
DEF_ID ::= proc_id;	6.1.A	57
DEF_ID ::= subtype_id;	3.3.2.A	36
DEF_ID ::= task_body_id;	9.1.B	65
DEF_ID ::= type_id;	3.3.1.A	35
DEF_ID ::= var_id;	3.2.A	34
DEF_OCCURRENCE ::= DEF_ID DEF_OP DEF_CHAR;	2.3	32
DEF_OP ::= def_op;	6.1.A	57
DESIGNATOR ::= ID OP;	2.3	32
DESIGNATOR_CHAR ::= DESIGNATOR used_char;	4.1.3	46
DSCRMT_VAR ::= dscrm_t_var;	3.7.1	42
DSCRMT_VAR_S ::= dscrm_t_var_s;	3.7.1	42
DSCRT_RANGE ::= constrained RANGE;	3.6.C	40
DSCRT_RANGE ::= index;	3.6.B	40
DSCRT_RANGE_S ::= dscrt_range_s;	3.6.A	40
DSCRT_RANGE_VOID ::= DSCRT_RANGE void;	9.5.A	66
ENUM_LITERAL ::= enum_id def_char;	3.5.1.B	38
EXCEPTION_DEF ::= rename;	8.5	64
EXCEPTION_DEF ::= void;	11.1	70
EXP ::= NAME numeric_literal null_access aggregate string_literal allocator conversion qualified parenthesized;	4.4.D	49
EXP ::= aggregate;	4.3.A	47
EXP ::= binary;	4.4.A	48
EXP ::= membership;	4.4.B	48
EXP_CONSTRAINED ::= EXP CONSTRAINED;	4.8	50
EXP_S ::= exp_s;	4.1.1	46
EXP_VOID ::= EXP void;	3.2.A	34
FORMAL_SUBPROG_DEF ::= NAME box no_default;	12.1.C	72
FORMAL_TYPE_SPEC ::= formal_dscrt formal_integer formal_fixed formal_float;	12.1.D	72
GENERIC_ASSOC ::= ACTUAL;	12.3.C	73
GENERIC_ASSOC ::= assoc;	12.3.B	73

GENERIC_ASSOC_S ::= generic_assoc_s;	12.3.A	73
GENERIC_HEADER ::= procedure function package_spec;	12.1.A	71
GENERIC_PARAM ::= in in_out type subprogram_decl;	12.1.C	72
GENERIC_PARAM_S ::= generic_param_s;	12.1.B	72
HEADER ::= entry;	9.5.A	66
HEADER ::= function;	6.1.B	58
HEADER ::= procedure;	6.1.B	58
ID ::= DEF_ID USED_ID;	2.3	32
ID_S ::= id_s;	3.2.C	35
INNER_RECORD ::= inner_record;	3.7.3.A	43
ITEM ::= DECL subprogram_body package_body task_body;	3.9.B	44
ITEM_S ::= item_s;	3.9.B	44
ITERATION ::= for reverse;	5.5.B	55
ITERATION ::= void;	5.5.A	54
ITERATION ::= while;	5.5.B	55
LANGUAGE ::= argument_id;	6.1.A	57
LOCATION ::= EXP_VOID pragma_id;	6.1.A	57
LOOP ::= loop;	5.5.A	54
MEMBERSHIP_OP ::= in_op not_in;	4.4.B	48
NAME ::= DESIGNATOR used_char indexed slice selected all attribute attribute_call;	4.1.A	45
NAME ::= function_call;	4.1.B	45
NAME_S ::= name_s;	9.10	68
NAME_VOID ::= NAME void;	5.7	56
OBJECT_DEF ::= EXP_VOID;	3.2.A	34
OBJECT_DEF ::= rename;	8.5	64
OP ::= DEF_OP USED_OP;	2.3	32
PACKAGE_DEF ::= instantiation;	12.3.A	73
PACKAGE_DEF ::= package_spec;	7.1.B	62
PACKAGE_DEF ::= rename;	8.5	64
PACKAGE_SPEC ::= package_spec;	7.1.B	62
PACK_BODY_DESC ::= block stub rename instantiation void;	7.1.A	62
PARAM ::= in;	6.1.C	59
PARAM ::= in_out;	6.1.C	59
PARAM ::= out;	6.1.C	59
PARAM_ASSOC ::= EXP assoc;	6.4	61
PARAM_ASSOC_S ::= param_assoc_s;	2.8.A	33
PARAM_S ::= param_s;	6.1.C	59
PRAGMA ::= pragma;	2.8.A	33
PRAGMA_S ::= pragma_s;	10.1.B	69
RANGE ::= range attribute attribute_call;	3.5	37
RANGE_VOID ::= RANGE void;	3.5.7	39
REP ::= simple_rep address record_rep;	13.1	74
REP_VOID ::= REP	3.7.A	41

void;		
SELECT_CLAUSE ::= pragma;	9.7.1.B	67
SELECT_CLAUSE ::= select_clause;	9.7.1.B	67
SELECT_CLAUSE_S ::= select_clause_s;	9.7.1.A	67
SHORT_CIRCUIT_OP ::= and_then or_else;	4.4.A	48
STM ::= if case named_stm LOOP block accept select cond_entry timed_entry;	5.1.D	52
STM ::= labeled;	5.1.B	51
STM ::= null_stm assign procedure_call exit return goto entry_call delay abort raise code;	5.1.C	51
STM ::= pragma;	9.7.1.B	67
STM ::= terminate;	5.1.A	51
STM_S ::= stm_s;	12.1.C	72
SUBPROGRAM_DEF ::= FORMAL_SUBPROG_DEF;	12.3.A	73
SUBPROGRAM_DEF ::= instantiation;	8.5	64
SUBPROGRAM_DEF ::= rename;	6.1.A	57
SUBPROGRAM_DEF ::= void;	6.1.A	57
SUBP_BODY_DESC ::= block stub instantiation FORMAL_SUBPROG_DEF rename LANGUAGE void;		
SUBUNIT_BODY ::= subprogram_body package_body task_body;	10.2.A	70
TASK_DEF ::= task_spec;	9.1.A	65
TYPE_RANGE ::= RANGE NAME;	4.4.B	48
TYPE_SPEC ::= CONSTRAINED;	3.2.A	34
TYPE_SPEC ::= FORMAL_TYPE_SPEC;	12.1.D	72
TYPE_SPEC ::= enum_literal_s integer fixed float array record access derived;	3.3.1.B	36
TYPE_SPEC ::= l_private;	7.4.A	63
TYPE_SPEC ::= private;	7.4.A	63
TYPE_SPEC ::= task_spec;	9.1.A	65
TYPE_SPEC ::= universal_integer universal_fixed universal_real;	App. I	76
TYPE_SPEC ::= void;	3.8.1	44
UNIT_BODY ::= package_body package_decl subunit generic subprogram_body subprogram_decl void;	10.1.B	69
USED_ID ::= used_object_id used_name_id used_bitn_id;	4.1.A	45

USED_OP ::= used_op used_bitn_op;	4.1.A	45
VARIANT ::= variant;	3.7.3.A	43
VARIANT_S ::= variant_s;	3.7.3.A	43
abort => as_name_s:NAME_S;	9.10	68
abort => lx_srcpos:source_position, lx_comments:comments;	9.10	68
accept => as_name:NAME, as_param_s:PARAM_S, as_stm_s:STM_S;	9.5.C	66
accept => lx_srcpos:source_position, lx_comments:comments;	9.5.C	66
access => as_constrained:CONSTRAINED;	3.8	44
access => lx_srcpos:source_position, lx_comments:comments;	3.8	44
access => sm_size:EXP_VOID, sm_storage_size:EXP_VOID, sm_controlled:Boolean;	3.8	44
address => as_name:NAME, as_exp:EXP;	13.5	75
address => lx_srcpos:source_position, lx_comments:comments;	13.5	75
aggregate => as_list:seq of COMP_ASSOC;	4.3.A	47
aggregate => lx_srcpos:source_position, lx_comments:comments;	4.3.A	47
aggregate => sm_exp_type:TYPE_SPEC, sm_constraint:CONSTRAINT, sm_normalized_comp_s:EXP_S;	4.3.A	47
alignment => as_pragma_s:PRAGMA_S, as_exp_void:EXP_VOID;	13.4.A	74
all => as_name:NAME;	4.1.3	46
all => lx_srcpos:source_position, lx_comments:comments;	4.1.3	46
all => sm_exp_type:TYPE_SPEC;	4.1.3	46
allocator => as_exp_constrained:EXP_CONSTRAINED;	4.8	50
allocator => lx_srcpos:source_position, lx_comments:comments;	4.8	50
allocator => sm_exp_type:TYPE_SPEC, sm_value:value;	4.8	50
alternative => as_choice_s:CHOICE_S, as_stm_s:STM_S;	5.4	53
alternative => lx_srcpos:source_position, lx_comments:comments;	5.4	53
alternative_s => as_list:seq of ALTERNATIVE;	5.4	53
alternative_s => lx_srcpos:source_position, lx_comments:comments;	5.4	53
and_then => lx_srcpos:source_position, lx_comments:comments;	4.4.A	48
argument_id => lx_symrep:symbol_rep;	App. 1	76
array => as_dscrt_range_s:DSCRT_RANGE_S, as_constrained:CONSTRAINED;	3.6.A	40
array => lx_srcpos:source_position, lx_comments:comments;	3.6.A	40
array => sm_size:EXP_VOID, sm_packing:Boolean;	3.6.A	40
assign => as_name:NAME, as_exp:EXP;	5.2	52
assign => lx_srcpos:source_position, lx_comments:comments;	5.2	52
assoc => as_designator:DESIGNATOR, as_actual:ACTUAL;	6.4	61
assoc => lx_srcpos:source_position, lx_comments:comments;	6.4	61
attr_id => lx_symrep:symbol_rep;	App. 1	76
attribute => as_name:NAME, as_id:ID;	4.1.4	47
attribute => lx_srcpos:source_position, lx_comments:comments;	4.1.4	47
attribute => sm_exp_type:TYPE_SPEC, sm_value:value;	4.1.4	47
attribute_call => as_name:NAME, as_exp:EXP;	4.1.4	47
attribute_call => lx_srcpos:source_position, lx_comments:comments;	4.1.4	47

attribute_call => sm_exp_type:TYPE_SPEC, sm_value:value;	4.1.4	47
binary => as_exp1:EXP, as_binary_op: BINARY_OP, as_exp2:EXP;	4.4.A	48
binary => lx_srcpos:source_position, lx_comments:comments;	4.4.A	48
binary => sm_exp_type:TYPE_SPEC, sm_value:value;	4.4.A	48
block => as_item_s:ITEM_S, as_stm_s:STM_S, as_alternative_s:ALTERNATIVE_S;	5.6	55
block => lx_srcpos:source_position, lx_comments:comments;	5.6	55
box => lx_srcpos:source_position, lx_comments:comments;	12.1.C	72
case => as_exp:EXP, as_alternative_s:ALTERNATIVE_S;	5.4	53
case => lx_srcpos:source_position, lx_comments:comments;	5.4	53
choice_s => as_list:seq of CHOICE;	3.7.3.A	43
choice_s => lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
code => as_name:NAME, as_exp:EXP;	13.8	75
code => lx_srcpos:source_position, lx_comments:comments;	13.8	75
comp_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.7.B	41
comp_id => sm_obj_type:TYPE_SPEC, sm_init_exp:EXP_VOID, sm_comp_spec:COMP_REP_VOID;	3.7.B	41
comp_rep => as_name:NAME, as_exp:EXP, as_range:RANGE;	13.4.B	75
comp_rep => lx_srcpos:source_position, lx_comments:comments;	13.4.B	75
comp_rep_s => as_list:seq of COMP_REP;	13.4.B	75
comp_rep_s => lx_srcpos:source_position, lx_comments:comments;	13.4.B	75
comp_unit => as_context:CONTEXT, as_unit_body:UNIT_BODY, as_pragma_s:PRAGMA_S;	10.1.B	69
comp_unit => lx_srcpos:source_position, lx_comments:comments;	10.1.B	69
compilation => as_list:seq of COMP_UNIT;	10.1.A	69
compilation => lx_srcpos:source_position, lx_comments:comments;	10.1.A	69
cond_clause => as_exp_void:EXP_VOID, as_stm_s:STM_S;	5.3.A	53
cond_clause => lx_srcpos:source_position, lx_comments:comments;	5.3.A	53
cond_entry => as_stm_s1:STM_S, as_stm_s2:STM_S;	9.7.2	68
cond_entry => lx_srcpos:source_position, lx_comments:comments;	9.7.2	68
const_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.2.A	34
const_id => sm_address:EXP_VOID, sm_obj_type:TYPE_SPEC, sm_obj_def:OBJECT_DEF, sm_first:DEF_OCCURRENCE;	3.2.A	34
constant => as_id_s:ID_S, as_type_spec:TYPE_SPEC, as_object_def:OBJECT_DEF;	3.2.A	34
constant => lx_srcpos:source_position, lx_comments:comments;	3.2.A	34
constrained => as_name:NAME, as_constraint:CONSTRAINT;	3.3.2.B	36
constrained => cd_impl_size:Integer;	3.3.2.B	36
constrained => lx_srcpos:source_position, lx_comments:comments;	3.3.2.B	36

constrained => sm_type_struct:TYPE_SPEC, sm_base_type:TYPE_SPEC, sm_constraint:CONSTRAINT;	3.3.2.B	36
context => as_list:seq of CONTEXT_ELEM;	10.1.1.A	69
context => lx_srcpos:source_position, lx_comments:comments;	10.1.1.A	69
conversion => as_name:NAME, as_exp:EXP;	4.6	50
conversion => lx_srcpos:source_position, lx_comments:comments;	4.6	50
conversion => sm_exp_type:TYPE_SPEC, sm_value:value;	4.6	50
decl_s => as_list:seq of DECL;	7.1.B	62
decl_s => lx_srcpos:source_position, lx_comments:comments;	7.1.B	62
def_char => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.5.1.B	38
def_char => sm_obj_type:TYPE_SPEC, sm_pos:Integer, sm_rep:Integer;	3.5.1.B	38
def_op => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	6.1.A	57
def_op => sm_spec:HEADER, sm_body:SUBP_BODY_DESC, sm_location:LOCATION, sm_stub:DEF_OCCURRENCE, sm_first:DEF_OCCURRENCE;	6.1.A	57
deferred_constant => as_id_s:ID_S, as_name:NAME;	7.4.B	63
deferred_constant => lx_srcpos:source_position, lx_comments:comments;	7.4.B	63
delay => as_exp:EXP;	9.6	66
delay => lx_srcpos:source_position, lx_comments:comments;	9.6	66
derived => as_constrained:CONSTRAINED;	3.4	37
derived => cd_impl_size:Integer;	3.4	37
derived => lx_srcpos:source_position, lx_comments:comments;	3.4	37
derived => sm_size:EXP_VOID, sm_actual_delta:Rational, sm_packing:Boolean, sm_controlled:Boolean, sm_storage_size:EXP_VOID;	3.4	37
dscrmt_aggregate => as_list:seq of COMP_ASSOC;	3.7.2	42
dscrmt_aggregate => lx_srcpos:source_position, lx_comments:comments;	3.7.2	42
dscrmt_aggregate => sm_normalized_comp_s:EXP_S;	3.7.2	42
dscrmt_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.7.1	42
dscrmt_id => sm_obj_type:TYPE_SPEC, sm_init_exp:EXP_VOID, sm_first:DEF_OCCURRENCE, sm_comp_spec:COMP_REP_VOID;	3.7.1	42
dscrmt_var => as_id_s:ID_S, as_name:NAME, as_object_def:OBJECT_DEF;	3.7.1	42
dscrmt_var => lx_srcpos:source_position, lx_comments:comments;	3.7.1	42
dscrmt_var_s => as_list:seq of DSCRMT_VAR;	3.7.1	42
dscrmt_var_s => lx_srcpos:source_position, lx_comments:comments;	3.7.1	42
dscrt_range_s => as_list:seq of DSCRMT_RANGE;	3.6.A	40
dscrt_range_s => lx_srcpos:source_position, lx_comments:comments;	3.6.A	40
entry => as_dscrt_range_void:DSCRMT_RANGE_VOID, as_param_s:PARAM_S;	9.5.A	66
entry => lx_srcpos:source_position, lx_comments:comments;	9.5.A	66
entry_call => as_name:NAME, as_param_assoc_s:PARAM_ASSOC_S;	9.5.B	66
entry_call => lx_srcpos:source_position,	9.5.B	66

lx_comments:comments;		
entry_call => sm_normalized_param_s:EXP_S;	9.5.B	66
entry_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	9.5.A	66
entry_id => sm_spec:HEADER, sm_address:EXP_VOID;	9.5.A	66
enum_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.5.1.B	38
enum_id => sm_obj_type:TYPE_SPEC, sm_pos:Integer, sm_rep:Integer;	3.5.1.B	38
enum_literal_s => as_list:seq of ENUM_LITERAL;	3.5.1.A	37
enum_literal_s => cd_impl_size:Integer;	3.5.1.A	37
enum_literal_s => lx_srcpos:source_position, lx_comments:comments;	3.5.1.A	37
enum_literal_s => sm_size:EXP_VOID;	3.5.1.A	37
exception => as_id_s:ID_S, as_exception_def:EXCEPTION_DEF;	11.1	70
exception => lx_srcpos:source_position, lx_comments:comments;	11.1	70
exception_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	11.1	70
exception_id => sm_exception_def:EXCEPTION_DEF;	11.1	70
exit => as_name_void:NAME_VOID, as_exp_void:EXP_VOID;	5.7	56
exit => lx_srcpos:source_position, lx_comments:comments;	5.7	56
exit => sm_stm:LOOP;	5.7	56
exp_s => as_list:seq of EXP;	4.1.1	46
exp_s => lx_srcpos:source_position, lx_comments:comments;	4.1.1	46
fixed => as_exp:EXP;	3.5.9	39
fixed => as_range_void:RANGE_VOID;		
fixed => cd_impl_size:Integer;	3.5.9	39
fixed => lx_srcpos:source_position, lx_comments:comments;	3.5.9	39
fixed => sm_size:EXP_VOID, sm_actual_delta:Rational, sm_bits:Integer, sm_base_type:TYPE_SPEC;	3.5.9	39
float => as_exp:EXP, as_range_void:RANGE_VOID;	3.5.7	39
float => cd_impl_size:Integer;	3.5.7	39
float => lx_srcpos:source_position, lx_comments:comments;	3.5.7	39
float => sm_size:EXP_VOID, sm_type_struct:TYPE_SPEC, sm_base_type:TYPE_SPEC;	3.5.7	39
for => as_id:ID, as_dscrt_range:DSCRT_RANGE;	5.5.B	55
for => lx_srcpos:source_position, lx_comments:comments;	5.5.B	55
formal_dscrt => lx_srcpos:source_position, lx_comments:comments;	12.1.D	72
formal_fixed => lx_srcpos:source_position, lx_comments:comments;	12.1.D	72
formal_float => lx_srcpos:source_position, lx_comments:comments;	12.1.D	72
formal_integer => lx_srcpos:source_position, lx_comments:comments;	12.1.D	72
function => as_param_s:PARAM_S, as_name_void:NAME_VOID;	6.1.B	58
function => lx_srcpos:source_position, lx_comments:comments;	6.1.B	58
function_call => as_name:NAME, as_param_assoc_s:PARAM_ASSOC_S;	6.4	61
function_call => lx_srcpos:source_position, lx_comments:comments;	6.4	61
function_call => sm_exp_type:TYPE_SPEC, sm_value:value, sm_normalized_param_s:EXP_S,	6.4	61

function_id =>	lx_prefix:Boolean; lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	6.1.A	57
function_id =>	sm_spec:HEADER, sm_body:SUBP_BODY_DESC, sm_location:LOCATION, sm_stub:DEF_OCCURRENCE, sm_first:DEF_OCCURRENCE;	6.1.A	57
generic =>	as_id:ID, as_generic_param_s:GENERIC_PARAM_S, as_generic_header:GENERIC_HEADER;	12.1.A	71
generic =>	lx_srcpos:source_position, lx_comments:comments;	12.1.A	71
generic_assoc_s =>	as_list:seq of GENERIC_ASSOC;	12.3.A	73
generic_assoc_s =>	lx_srcpos:source_position, lx_comments:comments;	12.3.A	73
generic_id =>	lx_symrep:symbol_rep, lx_srcpos:source_position, lx_comments:comments;	12.1.A	71
generic_id =>	sm_generic_param_s:GENERIC_PARAM_S, sm_spec:GENERIC_HEADER, sm_body:BLOCK_STUB_VOID, sm_first:DEF_OCCURRENCE, sm_stub:DEF_OCCURRENCE;	12.1.A	71
generic_param_s =>	as_list:seq of GENERIC_PARAM;	12.1.B	72
generic_param_s =>	lx_srcpos:source_position, lx_comments:comments;	12.1.B	72
goto =>	as_name:NAME;	5.9	56
goto =>	lx_srcpos:source_position, lx_comments:comments;	5.9	56
id_s =>	as_list:seq of ID;	3.2.C	35
id_s =>	lx_srcpos:source_position, lx_comments:comments;	3.2.C	35
if =>	as_list:seq of COND_CLAUSE;	5.3.A	53
if =>	lx_srcpos:source_position, lx_comments:comments;	5.3.A	53
in =>	as_id_s:ID_S, as_name:NAME, as_exp_void:EXP_VOID;	6.1.C	59
in =>	lx_srcpos:source_position, lx_comments:comments, lx_default:Boolean;	6.1.C	59
in_id =>	lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	6.1.C	59
in_id =>	sm_obj_type:TYPE_SPEC, sm_init_exp:EXP_VOID, sm_first:DEF_OCCURRENCE;	6.1.C	59
in_op =>	lx_srcpos:source_position, lx_comments:comments;	4.4.B	48
in_out =>	as_id_s:ID_S, as_name:NAME, as_exp_void:EXP_VOID;	6.1.C	59
in_out =>	lx_srcpos:source_position, lx_comments:comments;	6.1.C	59
in_out_id =>	lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	6.1.C	59
in_out_id =>	sm_obj_type:TYPE_SPEC, sm_first:DEF_OCCURRENCE;	6.1.C	59
index =>	as_name:NAME;	3.6.B	40
index =>	lx_srcpos:source_position, lx_comments:comments;	3.6.B	40
indexed =>	as_name:NAME, as_exp_s:EXP_S;	4.1.1	46
indexed =>	lx_srcpos:source_position, lx_comments:comments;	4.1.1	46
indexed =>	sm_exp_type:TYPE_SPEC;	4.1.1	46
inner_record =>	as_list:seq of COMP;	3.7.3.A	43
inner_record =>	lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
instantiation =>	as_name:NAME, as_generic_assoc_s:GENERIC_ASSOC_S;	12.3.A	73

instantiation => lx_srcpos:source_position, lx_comments:comments;	12.3.A	73
instantiation => sm_decl_s:DECL_S;	12.3.A	73
integer => as_range:RANGE;	3.5.4	38
integer => cd_impl_size:Integer;	3.5.4	38
integer => lx_srcpos:source_position, lx_comments:comments;	3.5.4	38
integer => sm_size:EXP_VOID, sm_type_struct:TYPE_SPEC, sm_base_type:TYPE_SPEC;	3.5.4	38
item_s => as_list:seq of ITEM;	3.9.B	44
item_s => lx_srcpos:source_position, lx_comments:comments;	3.9.B	44
iteration_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	5.5.B	55
iteration_id => sm_obj_type:TYPE_SPEC;	5.5.B	55
l_private => lx_srcpos:source_position, lx_comments:comments;	7.4.A	63
l_private => sm_discriminants:DSCRMT_VAR_S;	7.4.A	63
l_private_type_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	7.4.A	63
l_private_type_id => sm_type_spec:TYPE_SPEC;	7.4.A	63
label_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	5.1.B	51
label_id => sm_stm:STM;	5.1.B	51
labeled => as_id_s:ID_S, as_stm:STM;	5.1.B	51
labeled => lx_srcpos:source_position, lx_comments:comments;	5.1.B	51
loop => as_iteration:ITERATION, as_stm_s:STM_S;	5.5.A	54
loop => lx_srcpos:source_position, lx_comments:comments;	5.5.A	54
membership => as_exp:EXP, as_membership_op:MEMBERSHIP_OP, as_type_range:TYPE_RANGE;	4.4.B	48
membership => lx_srcpos:source_position, lx_comments:comments;	4.4.B	48
membership => sm_exp_type:TYPE_SPEC, sm_value:value;	4.4.B	48
name_s => as_list:seq of NAME;	9.10	68
name_s => lx_srcpos:source_position, lx_comments:comments;	9.10	68
named => as_choice_s:CHOICE_S, as_exp:EXP;	4.3.B	47
named => lx_srcpos:source_position, lx_comments:comments;	4.3.B	47
named_stm => as_id:ID, as_stm:STM;	5.5.A	54
named_stm => lx_srcpos:source_position, lx_comments:comments;	5.5.A	54
named_stm_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	5.5.A	54
named_stm_id => sm_stm:STM;	5.5.A	54
no_default => lx_srcpos:source_position, lx_comments:comments;	12.1.C	72
not_in => lx_srcpos:source_position, lx_comments:comments;	4.4.B	48
null_access => lx_srcpos:source_position, lx_comments:comments;	4.4.D	49
null_access => sm_exp_type:TYPE_SPEC, sm_value:value;	4.4.D	49
null_comp => lx_srcpos:source_position, lx_comments:comments;	3.7.B	41
null_stm => lx_srcpos:source_position, lx_comments:comments;	5.1.F	52
number => as_id_s:ID_S, as_exp:EXP;	3.2.B	35
number => lx_srcpos:source_position, lx_comments:comments;	3.2.B	35

number_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.2.B	35
number_id => sm_obj_type:TYPE_SPEC, sm_init_exp:EXP;	3.2.B	35
numeric_literal => lx_srcpos:source_position, lx_comments:comments, lx_numrep:number_rep;	4.4.D	49
numeric_literal => sm_exp_type:TYPE_SPEC, sm_value:value;	4.4.D	49
or_else => lx_srcpos:source_position, lx_comments:comments;	4.4.A	48
others => lx_srcpos:source_position, lx_comments:comments;	3.7.3.B	43
out => as_id_s:ID_S, as_name:NAME, as_exp_void:EXP_VOID;	6.1.C	59
out => lx_srcpos:source_position, lx_comments:comments;	6.1.C	59
out_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	6.1.C	59
out_id => sm_obj_type:TYPE_SPEC, sm_first:DEF_OCCURRENCE;	6.1.C	59
package_body => as_id:ID, as_block_stub:BLOCK_STUB;	7.1.C	63
package_body => lx_srcpos:source_position, lx_comments:comments;	7.1.C	63
package_decl => as_id:ID, as_package_def:PACKAGE_DEF;	7.1.A	62
package_decl => lx_srcpos:source_position, lx_comments:comments;	7.1.A	62
package_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	7.1.A	62
package_id => sm_spec:PACKAGE_SPEC, sm_body:PACK_BODY_DESC, sm_address:EXP_VOID, sm_stub:DEF_OCCURRENCE, sm_first:DEF_OCCURRENCE;	7.1.A	62
package_spec => as_decl_s1:DECL_S, as_decl_s2:DECL_S;	7.1.B	62
package_spec => lx_srcpos:source_position, lx_comments:comments;	7.1.B	62
param_assoc_s => as_list:seq of PARAM_ASSOC;	2.8.A	33
param_assoc_s => lx_srcpos:source_position, lx_comments:comments;	2.8.A	33
param_s => as_list:seq of PARAM;	6.1.C	59
param_s => lx_srcpos:source_position, lx_comments:comments;	6.1.C	59
parenthesized => as_exp:EXP;	4.4.D	49
parenthesized => lx_srcpos:source_position, lx_comments:comments;	4.4.D	49
parenthesized => sm_exp_type:TYPE_SPEC, sm_value:value;	4.4.D	49
pragma => as_id:ID, as_param_assoc_s:PARAM_ASSOC_S;	2.8.A	33
pragma => lx_srcpos:source_position, lx_comments:comments;	2.8.A	33
pragma_id => as_list:seq of ARGUMENT;	App. I	76
pragma_id => lx_symrep:symbol_rep;	App. I	76
pragma_s => as_list:seq of PRAGMA;	10.1.B	69
pragma_s => lx_srcpos:source_position, lx_comments:comments;	10.1.B	69
private => lx_srcpos:source_position, lx_comments:comments;	7.4.A	63
private => sm_discriminants:DSCRMT_VAR_S;	7.4.A	63
private_type_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	7.4.A	63
private_type_id => sm_type_spec:TYPE_SPEC;	7.4.A	63
proc_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	6.1.A	57

proc_id => sm_spec:HEADER, sm_body:SUBP_BODY_DESC, sm_location:LOCATION, sm_stub:DEF_OCCURRENCE, sm_first:DEF_OCCURRENCE;	6.1.A	57
procedure => as_param_s:PARAM_S;	6.1.B	58
procedure => lx_srcpos:source_position, lx_comments:comments;	6.1.B	58
procedure_call => as_name:NAME, as_param_assoc_s:PARAM_ASSOC_S;	6.4	61
procedure_call => lx_srcpos:source_position, lx_comments:comments;	6.4	61
procedure_call => sm_normalized_param_s:EXP_S;	6.4	61
qualified => as_name:NAME, as_exp:EXP;	4.7	50
qualified => lx_srcpos:source_position, lx_comments:comments;	4.7	50
qualified => sm_exp_type:TYPE_SPEC, sm_value:value;	4.7	50
raise => as_name_void:NAME_VOID;	11.3	71
raise => lx_srcpos:source_position, lx_comments:comments;	11.3	71
range => as_exp1:EXP, as_exp2:EXP;	3.5	37
range => lx_srcpos:source_position, lx_comments:comments;	3.5	37
range => sm_base_type:TYPE_SPEC;	3.5	37
record => as_list:seq of COMP;	3.7.A	41
record => lx_srcpos:source_position, lx_comments:comments;	3.7.A	41
record => sm_size:EXP_VOID, sm_discriminants:DSCRMT_VAR_S, sm_packing:Boolean, sm_record_spec:REP_VOID;	3.7.A	41
record_rep => as_name:NAME, as_alignment:ALIGNMENT, as_comp_rep_s:COMP_REP_S;	13.4.A	74
record_rep => lx_srcpos:source_position, lx_comments:comments;	13.4.A	74
rename => as_name:NAME;	8.5	64
rename => lx_srcpos:source_position, lx_comments:comments;	8.5	64
return => as_exp_void:EXP_VOID;	5.8	56
return => lx_srcpos:source_position, lx_comments:comments;	5.8	56
reverse => as_id:ID, as_dscrt_range:DSCRRT_RANGE;	5.5.B	55
reverse => lx_srcpos:source_position, lx_comments:comments;	5.5.B	55
select => as_select_clause_s:SELECT_CLAUSE_S, as_stm_s:STM_S;	9.7.1.A	67
select => lx_srcpos:source_position, lx_comments:comments;	9.7.1.A	67
select_clause => as_exp_void:EXP_VOID, as_stm_s:STM_S;	9.7.1.B	67
select_clause => lx_srcpos:source_position, lx_comments:comments;	9.7.1.B	67
select_clause_s => as_list:seq of SELECT_CLAUSE;	9.7.1.A	67
select_clause_s => lx_srcpos:source_position, lx_comments:comments;	9.7.1.A	67
selected => as_name:NAME, as_designator_char:DESIGNATOR_CHAR;	4.1.3	46
selected => lx_srcpos:source_position, lx_comments:comments;	4.1.3	46
selected => sm_exp_type:TYPE_SPEC;	4.1.3	46
simple_rep => as_name:NAME, as_exp:EXP;	13.3	74
simple_rep => lx_srcpos:source_position, lx_comments:comments;	13.3	74
slice => as_name:NAME, as_dscrt_range:DSCRRT_RANGE;	4.1.2	46
slice => lx_srcpos:source_position, lx_comments:comments;	4.1.2	46
slice => sm_exp_type:TYPE_SPEC,	4.1.2	46

sm_constraint:CONSTRAINT;		
stm_s => as_list:seq of STM;	5.1.A	51
stm_s => lx_srcpos:source_position, lx_comments:comments;	5.1.A	51
string_literal => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.4.D	49
string_literal => sm_exp_type:TYPE_SPEC, sm_constraint:CONSTRAINT, sm_value:value;	4.4.D	49
stub => lx_srcpos:source_position, lx_comments:comments;	10.2.B	70
subprogram_body => as_designator:DESIGNATOR, as_header:HEADER, as_block_stub:BLOCK_STUB;	6.3	60
subprogram_body => lx_srcpos:source_position, lx_comments:comments;	6.3	60
subprogram_decl => as_designator:DESIGNATOR, as_header:HEADER, as_subprogram_def:SUBPROGRAM_DEF;	6.1.A	57
subprogram_decl => lx_srcpos:source_position, lx_comments:comments;	6.1.A	57
subtype => as_id:ID, as_constrained:CONSTRAINED;	3.3.2.A	36
subtype => lx_srcpos:source_position, lx_comments:comments;	3.3.2.A	36
subtype_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.3.2.A	36
subtype_id => sm_type_spec:CONSTRAINED;	3.3.2.A	36
subunit => as_name:NAME, as_subunit_body:SUBUNIT_BODY;	10.2.A	70
subunit => lx_srcpos:source_position, lx_comments:comments;	10.2.A	70
task_body => as_id:ID, as_block_stub:BLOCK_STUB;	9.1.B	65
task_body => lx_srcpos:source_position, lx_comments:comments;	9.1.B	65
task_body_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	9.1.B	65
task_body_id => sm_type_spec:TYPE_SPEC, sm_body:BLOCK_STUB_VOID, sm_first:DEF_OCCURRENCE, sm_stub:DEF_OCCURRENCE;	9.1.B	65
task_decl => as_id:ID, as_task_def:TASK_DEF;	9.1.A	65
task_decl => lx_srcpos:source_position, lx_comments:comments;	9.1.A	65
task_spec => as_decl_s:DECL_S;	9.1.A	65
task_spec => lx_srcpos:source_position, lx_comments:comments;	9.1.A	65
task_spec => sm_body:BLOCK_STUB_VOID, sm_address:EXP_VOID, sm_storage_size:EXP_VOID;	9.1.A	65
terminate => lx_srcpos:source_position, lx_comments:comments;	9.7.1.B	67
timed_entry => as_stm_s1:STM_S, as_stm_s2:STM_S;	9.7.3	68
timed_entry => lx_srcpos:source_position, lx_comments:comments;	9.7.3	68
type => as_id:ID, as_dscrmt_var_s:DSCRMT_VAR_S, as_type_spec:TYPE_SPEC;	3.3.1.A	35
type => lx_srcpos:source_position, lx_comments:comments;	3.3.1.A	35
type_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.3.1.A	35
type_id => sm_type_spec:TYPE_SPEC, sm_first:DEF_OCCURRENCE;	3.3.1.A	35
universal_fixed => ;	App. I	76
universal_integer => ;	App. I	76
universal_real => ;	App. I	76

use => as_list:seq of NAME;	8.4	64
use => lx_srcpos:source_position, lx_comments:comments;	8.4	64
used_btn_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_btn_id => sm_operator:operator;	4.1.A	45
used_btn_op => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_btn_op => sm_operator:operator;	4.1.A	45
used_char => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_char => sm_defn:DEF_OCCURRENCE, sm_exp_type:TYPE_SPEC, sm_value:value;	4.1.A	45
used_name_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_name_id => sm_defn:DEF_OCCURRENCE;	4.1.A	45
used_object_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_object_id => sm_exp_type:TYPE_SPEC, sm_defn:DEF_OCCURRENCE, sm_value:value;	4.1.A	45
used_op => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_op => sm_defn:DEF_OCCURRENCE;	4.1.A	45
var => as_id_s:ID_S, as_type_spec:TYPE_SPEC, as_object_def:OBJECT_DEF;	3.2.A	34
var => lx_srcpos:source_position, lx_comments:comments;	3.2.A	34
var_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.2.A	34
var_id => sm_obj_type:TYPE_SPEC, sm_address:EXP_VOID, sm_obj_def:OBJECT_DEF;	3.2.A	34
variant => as_choice_s:CHOICE_S, as_record:INNER_RECORD;	3.7.3.A	43
variant => lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
variant_part => as_name:NAME, as_variant_s:VARIANT_S;	3.7.3.A	43
variant_part => lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
variant_s => as_list:seq of VARIANT;	3.7.3.A	43
variant_s => lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
void => ;	2	32
while => as_exp:EXP;	5.5.B	55
while => lx_srcpos:source_position, lx_comments:comments;	5.5.B	55
with => as_list:seq of NAME;	10.1.1.B	70
with => lx_srcpos:source_position, lx_comments:comments;	10.1.1.B	70

APÊNDICE 2

ESPECIFICAÇÃO DAS ROTINAS DE
RESOLUÇÃO DE NOMES E EXPRESSÕES

```

0001 PROCEDURE DECORA_BINARY( PNO : PT_TREE ) IS
0002 -----
0003 PE1 , PE2 : PT_TREE ;
0004 BEGIN
0005     PE1 := AS_EXP1( PNO ) ;
0006     PE2 := AS_EXP2( PNO ) ;
0007     CRIA_LISTA_TIPO( PNO ) ;
0008     COLOCA_TIPO( PNO , "BOOLEAN" ) ;
0009     DECORAR( PE1 ) ;
0010     DECORAR( PE2 ) ;
0011 END DECORA_BINARY ;
0012 -----
0013 ---
0014 -----
0015 ---
0016 PROCEDURE DESCIDA_FINAL_BINARY( PNO , PL_TIPO : PT_TREE ) IS
0017 -----
0018 PE1 , PE2 : PT_TREE ;
0019 BEGIN
0020     IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0021         MSG( "ESPERADO TIPO BOOLEANO" , PNO ) ;
0022         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0023     ELSE
0024         DESCIDA_FINAL( PE1 , PL_TIPO ) ;
0025         DESCIDA_FINAL( PE2 , PL_TIPO ) ;
0026     END IF ;
0027 END DESCIDA_FINAL_BINARY ;
0028 ---
0029 -----
0030 ---
0031 --- DBS : A 1. FASE DA ROTINA ( DECORA_BINARY ) SIMPLEMENTE PASSA O
0032 --- PROCESSAMENTO PARA AS 2 SUB-EXPRESSOES . NOTE QUE COMO O
0033 --- TIPO DA EXPRESSAO GLOBAL E' CONHECIDO ( BOOLEAN ) , NAO E'
0034 --- NECESSARIO LIMITAR-SE O TIPO DA EXPRESSAO PELO TIPO DAS
0035 --- SUB-EXPRESSOES.
0036 ---
0037 -----
0038 PROCEDURE DECORA_MEMBERSHIP( PNO : PT_TREE ) IS
0039 -----
0040 PEXP , PTR , PTB , PDEF : PT_TREE ;
0041 BEGIN
0042     PEXP := AS_EXP( PNO ) ;
0043     PTR  := AS_TYPE_RANGE( PNO ) ;
0044     CRIA_LISTA_TIPO( PNO ) ;
0045     COLOCA_TIPO( PNO , "BOOLEAN" ) ;
0046     DECORA( PEXP ) ;
0047     IF APANHA_NOME_NO( PTR ) = DN_RANGE THEN
0048         VERIFICA_RANGE( PTR ) ;
0049         CRIA_LISTA_TIPO( PTR ) ;
0050         COLOCA_NA_LISTA( PTR , SM_EXP_TYPE( PTR ) ) ;
0051         FILTRA_LISTA( PEXP , PTR ) ;
0052     ELSE
0053         IF NOT DECLARADO ( PTR ) THEN
0054             MSG( "IDENTIFICADOR NAO DECLARADO", PTR ) ;
0055         ELSE
0056             PDEF := APONTA_DEFINICAO( PNO ) ;
0057             IF APANHA_NOME_NO( PDEF ) /= DN_TYPE_ID OR
0058                APANHA_NOME_NO( PDEF ) /= DN_SUBTYPE_ID THEN
0059                 MSG( "ESPERADOR IDENTIFICADOR DE TIPO" , PTR ) ;
0060             ELSE

```



```

0061         PTR := APANHA_TIPO_BASE( PDEF ) ;
0062         COLOCA_NA_LISTA( PTR , PTR ) ;
0063         FILTRA_LISTA( PEXF , PTR ) ;
0064     END IF ;
0065 END IF ;
0066 END IF ;
0067 END DECORA_MEMBERSHIP ;
0068 ----
0069 ----
0070 ----
0071 PROCEDURE DESCIDA_FINAL_MEMBERSHIP( PNO , PL_TIPO : PT_TREE ) IS
0072     -----
0073     PEXF , PTR , PL_TIPO_EXP : PT_TREE ;
0074     BEGIN
0075         PEXF := AS_EXP( PNO ) ;
0076         PTR := AS_TYPE_RANGE( PNO ) ;
0077         IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0078             MSG( "ESPERADO TIPO BOOLEANO" , PNO ) ;
0079             SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0080         ELIF TIPO_DEFINIDO( PEXF ) /= 1 THEN
0081             MSG( "EXPRESSAO INVALIDA" , PEXF ) ;
0082             SM_EXP_TYPE( PEXF ) := SUPER_TIPO ;
0083         ELSE
0084             SM_EXP_TYPE( PNO ) := "BOOLEAN:*****" ;
0085             PL_TIPO_EXP := APANHA_TIPO( PEXF ) ;
0086             DESCIDA_FINAL( PEXF , PL_TIPO_EXP ) ;
0087         END IF ;
0088     END DESCIDA_FINAL_MEMBERSHIP ;
0089     ----
0090     ----
0091     ----
0092     --- OBS : - A ROTINA "FILTRA_LISTA" PERCORRE AS LISTAS DE TIPOS DOS
0093     --- NO'S AFONTOADOS POR SEUS PARAMETROS ELIMINANDO OS ELEMENTOS
0094     --- QUE NAO APARECAM EM ANHAS AS LISTAS E COMO SEMPRE E'
0095     --- POSSIVEL DESCOBRIR O TIPO DA SUB-ARVORE "RANGE" OU DA
0096     --- SUB-ARVORE "TYPE_MARK" , ESSA ROTINA DEIXA NO MAXIMO
0097     --- UM TIPO NAS 2 LISTAS .
0098     ----
0099     --- - A FUNCAO "TIPO_DEFINIDO" CALCULA QUANTOS ELEMENTOS
0100     --- EXISTEM NA LISTA DE TIPOS DO NO' AFONTOADO POR SEU
0101     --- PARAMETRO E NOTE QUE NA SITUACAO OU O TIPO E' CONHE_
0102     --- CIDO ( E EXISTE UM UNICO ELEMENTO NA LISTA ) OU O TIPO E'
0103     --- INDETERMINADO ( E NAO EXISTE NENHUM ELEMENTO NA LISTA ) .
0104     ----
0105     --- - A FUNCAO "APANHA_TIPO" DEVOLVE UM PONTEIRO PARA O TIPO
0106     --- ( UNICO !!! ) DO NO' AFONTOADO POR SEU PARAMETRO .
0107     ----
0108     ----
0109     ----
0110     ----
0111 PROCEDURE DECORA_NUMERIC_LITERAL( PNO : PT_TREE ) IS
0112     -----
0113     BEGIN
0114         CRIA_LISTA_TIPO( PNO ) ;
0115         IF E_INTEIRO( PNO ) THEN
0116             COLOCA_TIPO( PNO , "UNIVERSAL_INTEGER" ) ;
0117         ELSE
0118             COLOCA_TIPO( PNO , "UNIVERSAL_REAL" ) ;
0119         END IF ;
0120     END DECORA_NUMERIC_LITERAL ;

```

```

0121 ----
0122 -----
0123 ----
0124 PROCEDURE DESCIDA_FINAL_NUMERIC_LITERAL( PNO , PL_TIPO : PL_TREE ) IS
0125 -----
0126 BEGIN
0127     IF E_INTEIRO( PNO ) THEN
0128         IF NOT TIPO_INTEIRO( PL_TIPO ) THEN
0129             MSG("EXPRESSAO INVALIDA" , PNO ) ;
0130             SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0131         ELSE
0132             SM_EXP_TYPE( PNO ) := PL_TIPO ;
0133         END IF ;
0134     ELSE
0135         IF NOT TIPO_REAL( PL_TIPO ) THEN
0136             MSG("EXPRESSAO INVALIDA" , PNO ) ;
0137             SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0138         ELSE
0139             SM_EXP_TYPE( PNO ) := PL_TIPO ;
0140         END IF ;
0141     END IF ;
0142 END DESCIDA_FINAL_NUMERIC_LITERAL ;
0143 ----
0144 -----
0145 ----
0146 --- OBS : - A FUNCAO "E_INTEIRO" VERIFICA SE O TIPO DA LITERAL NUME_
0147 ---         RICA E' INTEIRO OU NAO
0148 ----
0149 --- - AS FUNCOES "TIPO_INTEIRO" E "TIPO_REAL" VERIFICAM SE
0150 ---   O TIPO REPRESENTADO PELOS RESPECTIVOS PARAMETROS TEM
0151 ---   COMO TIPO BASE UM TIPO INTEIRO OU REAL , RESPECTIVAMENTE .
0152 ----
0153 -----
0154 ----
0155 PROCEDURE DECORA_NULL_ACCESS( PNO : PL_TREE ) IS
0156 -----
0157 BEGIN
0158     CRIA_LISTA_TIPO( PNO ) ;
0159     COLOCA_CLASSE( PNO , "ACESSO_UNIVERSAL" ) ;
0160 END DECORA_NULL_ACCESS ;
0161 ----
0162 -----
0163 ----
0164 PROCEDURE DESCIDA_FINAL_NULL_ACCESS( PNO , PL_TIPO ) IS
0165 -----
0166 BEGIN
0167     IF NOT TIPO_ACESSO( PL_TIPO ) THEN
0168         MSG("EXPRESSAO INVALIDA" , PNO ) ;
0169         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0170     ELSE
0171         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0172     END IF ;
0173 END DESCIDA_FINAL_NULL_ACCESS ;
0174 ----
0175 -----
0176 ----
0177 --- OBS : - ESSA ROTINA E' PRATICAMENTE IGUAL AS ROTINAS DO NO'
0178 ---         "NUMERIC_LITERAL" ; O TIPO "ACESSO_UNIVERSAL" NAO E' PRE_
0179 ---         DEFINIDO NA LINGUAGEM , SENDO USADO SOMENTE PELO ANALISA_
0180 ---         DOR SEMANTICO .

```

```

0181 ----
0182 -----
0183 ----
0184 PROCEDURE DECORA_AGGREGATE ( PNO : PT_TREE ) IS
0185 -----
0186 BEGIN
0187     CRIA_LISTA_TIPO( PNO ) ;
0188     COLOCA_CLASSE( PNO , ARRAY_RECORD ) ;
0189 END DECORA_AGGREGATE ;
0190 ----
0191 -----
0192 ----
0193 PROCEDURE DESCIDA_FINAL_AGGREGATE ( PNO , PL_TIPO : PT_TREE ) IS
0194 -----
0195 BEGIN
0196     IF NOT TIPO_ARRAY( PL_TIPO ) THEN
0197         MSG("EXPRESSAO INVALIDA" , PNO ) ;
0198         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0199     ELSE
0200         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0201     END IF ;
0202 END DESCIDA_FINAL_AGGREGATE ;
0203 ----
0204 -----
0205 ----
0206 --- OBS : - A 1. FASE DA ROTINA SIMPLEMENTE INDICA QUE A CLASSE DO
0207 --- TIPO CORRESPONDENTE AO NO' DEVE SER ARRAY OU RECORD .
0208 ---
0209 --- - COMO O TIPO DE UM AGREGADO DEVE SER DETERMINADO SOMENTE
0210 --- PELO CONTEXTO NO QUAL ELE APARECE ( EXCLUINDO O PROPRIO
0211 --- AGREGADO ) , O UNICO PROCESSAMENTO NECESSARIO E' VERIFICAR
0212 --- SE O TIPO RESULTANTE E' DA CLASSE ARRAY OU RECORD E , EM
0213 --- CASO AFIRMATIVO , PREENCHER O CAMPO SM_EXP_TYPE DO NO'
0214 --- NA DESCIDA_FINAL . NOTE QUE SE O TIPO DO AGREGADO NAO FOR
0215 --- CONHECIDO OCORRERA' UM ERRO ANTES DO NO' SER Atingido .
0216 ----
0217 -----
0218 ----
0219 PROCEDURE DECORA_STRING_LITERAL( PNO : PT_TREE ) IS
0220 -----
0221 BEGIN
0222     MONTA_LISTA_TIPO( PNO ) ;
0223     COLOCA_CLASSE( PNO , ARRAY_UNIDIMENSIONAL_CHARACTER ) ;
0224 END DECORA_STRING_LITERAL ;
0225 ----
0226 -----
0227 ----
0228 PROCEDURE DESCIDA_FINAL_STRING_LITERAL ( PNO , PL_TIPO : PT_TREE ) IS
0229 -----
0230 BEGIN
0231     IF NOT TIPO_ARRAY_UNIDIMENSIONAL_CHARACTER( PL_TIPO ) THEN
0232         MSG("EXPRESSAO INVALIDA" , PNO) ;
0233         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0234     ELSE
0235         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0236     END IF ;
0237 END DESCIDA_FINAL_STRING_LITERAL ;
0238 ----
0239 -----
0240 ----

```

```

0241 --- OBS : - O TIPO DE UMA STRING_LITERAL DEVE SER DETERMINADO SOMENTE
0242 --- PELO CONTEXTO NO QUAL A LITERAL APARECE , EXCLUINDO A PRO_
0243 --- PRIA LITERAL MAS USANDO-SE O FATO DA LITERAL REPRESENTAR
0244 --- UM VALOR DE UM ARRAY UNIDIMENSIONAL COM COMPONENTES DO
0245 --- TIPO PREDEFINIDO "CHARACTER" .
0246 ---
0247 --- - AS OBSERVAÇÕES FEITAS PARA O PROCESSAMENTO DE UM NO'
0248 --- "AGGREGATE" SÃO VÁLIDAS PARA O NO' "STRING_LITERAL" .
0249 ---
0250 -----
0251 ---
0252 PROCEDURE DECORA_ALLOCATOR( PNO ; FT_TREE ) IS
0253 -----
0254 P1 , P2 : FT_TREE ;
0255 BEGIN
0256     P1 := AS_EXP_CONSTRAINED( PNO ) ; -- CONSTRAINED / QUALIFIED
0257     P2 := AS_NAME( P1 ) ;
0258     CONTROLISTA_TIPO_ACESSO( PNO , P2 ) ;
0259     IF APANHA_NOME_NO( P1 ) /= DN_QUALIFIED THEN
0260         DECORA( P1 ) ;
0261     END IF ;
0262 END DECORA_ALLOCATOR ;
0263 ---
0264 -----
0265 ---
0266 PROCEDURE DESCIDA_FINAL_ALLOCATOR( PNO , PL_TIPO : FT_TREE ) IS
0267 -----
0268 P1 : FT_TREE ;
0269 BEGIN
0270     IF CONTIDO( PL_TIPO , PNO ) THEN
0271         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0272         P1 := AS_EXP_CONSTRAINED( PNO ) ;
0273         IF APANHA_NOME_NO( P1 ) = DN_QUALIFIED THEN
0274             PL_TIPO_ACC := APANHA_TIPO_NO_ACESSADO( PNO ) ;
0275             DESCIDA_FINAL( P1 , PL_TIPO ) ;
0276         END IF ;
0277     ELSE
0278         MSG("EXPRESSÃO INVÁLIDA", PNO ) ;
0279         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0280     END IF ;
0281 END DESCIDA_FINAL_ALLOCATOR ;
0282 ---
0283 -----
0284 ---
0285 ---
0286 --- OBS : - A ROTINA "CONTROLISTA_TIPO_ACESSO" CONSTRÓI A LISTA DE
0287 --- TIPOS POSSÍVEIS DE "PNO" A PARTIR DOS TIPOS DE ACESSO
0288 --- DE "USED_NAME_ID" .
0289 ---
0290 --- - A ROTINA "APANHA_TIPO_ACESSADO" DEVOLVE UM PONTEIRO
0291 --- PARA O TIPO APONTADO PELO TIPO DE ACESSO .
0292 ---
0293 -----
0294 ---
0295 PROCEDURE DECORA_CONVERSION ( PNO : FT_TREE ) IS
0296 -----
0297 PN , PEXP , PDEF , PTB : FT_TREE ;
0298 BEGIN
0299     PN := AS_NAME( PNO ) ;
0300     PEXP := AS_EXP( PNO ) ;

```

```

0301 IF NOT DECLARADO( PNO ) THEN
0302 MSG("IDENTIFICADOR NAO DECLARADO" , PN ) ¶
0303 ELSE
0304 PDEF := APONTA_DEFINICAO( PN ) ¶
0305 IF APANHA_NOME_NO( PDEF ) /= DN_TYPE THEN
0306 MSG("ESPERADOR IDENTIFICADOR DE TIPO" , PN ) ¶
0307 ELSE
0308 PTB := APANHA_TIPO_BASE( PDEF ) ¶
0309 SM_EXP_TYPE( PNO ) := PTB ¶
0310 CRIA_LISTA_TIPO( PNO ) ¶
0311 COLOCA_TIPO( PNO , PTB ) ¶
0312 DECORA( PEXP ) ¶
0313 END IF ¶
0314 END IF ¶
0315 END DECORA_CONVERSION ¶
0316 ---
0317 ---
0318 ---
0319 PROCEDURE DESCIDA_FINAL_CONVERSION( PNO , PL_TIPO : PT_TREE ) IS
0320 -----
0321 PL_TIPO_EXP : PT_TREE ¶
0322 BEGIN
0323 IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0324 MSG("CONVERSAO DE TIPO INVALIDA" , PNO) ¶
0325 SM_EXP_TYPE( PNO ) := SUPER_TIPO ¶
0326 ELSIF NOT TIPO_DEFINIDO(PEXP ) THEN
0327 MSG("EXPRESSAO INVALIDA" , PEXP ) ¶
0328 SM_EXP_TYPE( PNO ) := SUPER_TIPO ¶
0329 ELSE
0330 PL_TIPO_EXP := APANHA_TIPO_EXP( PEXP ) ¶
0331 DESCIDA_FINAL( PEXP , PL_TIPO_EXP ) ¶
0332 END IF ¶
0333 END DESCIDA_FINAL_CONVERSION ¶
0334 ---
0335 -----
0336 ---
0337 --- OBS : - O TIPO DO NO "CONVERSION" E DADO PELO "TYPE_MARK"
0338 --- CORRESPONDENTE , AO PASSO QUE O TIPO DA EXPRESSAO DEVE
0339 --- SER CALCULADA INDEPENDENTEMENTE DO CONTEXTO .
0340 ---
0341 --- - A ROTINA "TIPO_DEFINIDO" VERIFICA SE A SUB-ARVORE APONTADA
0342 --- PELO 1. PARAMETRO JA TEM 1 TIPO UNICO DEFINIDO .
0343 ---
0344 -----
0345 ---
0346 PROCEDURE DECORA_QUALIFIED( PNO : PT_TREE ) IS
0347 -----
0348 PN , PE , PDEF , PTB : PT_TREE ¶
0349 BEGIN
0350 PN := AS_NAME( PNO ) ¶
0351 PE := AS_EXP( PNO ) ¶
0352 IF NOT DECLARADO( PN ) THEN
0353 MSG("IDENTIFICADOR NAO DECLARADO" , PN) ¶
0354 ELSE
0355 PDEF := APONTA_DEFINICAO( PNO ) ¶
0356 IF APANHA_NOME_NO( PDEF ) /= DN_TYPE_ID THEN
0357 MSG("EXPRESSAO INVALIDA" , PN) ¶
0358 ELSE
0359 CRIA_LISTA_TIPO( PNO ) ¶
0360 PTB := APANHA_TIPO_BASE( PDEF ) ¶

```

```

0361         COLOCAL_TIPO( PNO , PTB ) ;
0362         DECORA( PE ) ;
0363     END IF ;
0364 END IF ;
0365 END DECORA_QUALIFIED ;
0366 ---
0367 -----
0368 ---
0369 PROCEDURE DESCIDA_FINAL_QUALIFIED( PNO , PL_TIPO : PT_TREE ) IS
0370 -----
0371 PE : PT_TREE ;
0372 BEGIN
0373     PE := AS_EXP( PNO ) ;
0374     IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0375         MSG("EXPRESSAO INVALIDA" , PNO) ;
0376         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0377     ELSE
0378         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0379         DESCIDA_FINAL( PE , PL_TIPO ) ;
0380     END IF ;
0381 END DESCIDA_FINAL_QUALIFIED ;
0382 ---
0383 -----
0384 ---
0385 --- OBS : - VER AS OBSERVAÇÕES DA ROTINA DO NO' "CONVERSION"
0386 ---
0387 -----
0388 ---
0389 PROCEDURE DECORA_PARENTHESESIZED( PNO : PT_TREE ) IS
0390 -----
0391 PE : PT_TREE ;
0392 BEGIN
0393     PE := AS_EXP( PNO ) ;
0394     CRIA_LISTA_TIPO( PNO ) ;
0395     DECORA( PE ) ;
0396     COPIA_LISTA_DE_TIPO( PE , PNO ) ;
0397 END DECORA_PARENTHESESIZED ;
0398 ---
0399 -----
0400 ---
0401 PROCEDURE DESCIDA_FINAL_PARENTHESESIZED( PNO , PL_TIPO : PT_TREE ) IS
0402 -----
0403 PE : PT_TREE ;
0404 BEGIN
0405     PE := AS_EXP( PNO ) ;
0406     IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0407         MSG("EXPRESSAO INVALIDA" , PNO) ;
0408         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0409     ELSE
0410         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0411         DESCIDA_FINAL( PE , PL_TIPO ) ;
0412     END IF ;
0413 END DESCIDA_FINAL_PARENTHESESIZED ;
0414 ---
0415 -----
0416 ---
0417 --- OBS : - A ROTINA "COPIA_LISTA_DE_TIPO" COPIA SIMPLEMENTE A
0418 --- LISTA DE TIPOS DO NO' APONTADO PELO 1. PARAMETRO PARA
0419 --- O NO' APONTADO PELO 2. PARAMETRO.
0420 ---

```

```

0421 -----
0422 ----
0423 PROCEDURE DECORA_USED_CHAR( PNO : PT_TREE ) IS
0424 -----
0425 BEGIN
0426     CRIA_LISTA_TIPO( PNO ) ;
0427     COLOCA_CLASSE( PNO , TIPO_CARACTER ) ;
0428 END DECORA_USED_CHAR ;
0429 ----
0430 -----
0431 ----
0432 PROCEDURE DESCIDA_FINAL_USED_CHAR( PNO , PL_TIPO : PT_TREE ) IS
0433 -----
0434 BEGIN
0435     IF NOT E_TIPO_DE_ENUMERACAO( PL_TIPO ) THEN
0436         MSG("EXPRESSAO INVALIDA", PNO) ;
0437         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0438     ELSIF NOT E_LITERAL_DE_ENUMERACAO( PNO , PL_TIPO ) THEN
0439         MSG("EXPRESSAO INVALIDA", PNO) ;
0440         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0441     ELSE
0442         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0443     END IF ;
0444 END DESCIDA_FINAL_USED_CHAR ;
0445 ----
0446 -----
0447 ----
0448 ---- OBS : - COMO UM NO' "USED_CHAR" E' 1 FOLHA DA EXPRESSAO , A UNICA
0449 ----          ACAO NECESSARIA NA 1. FASE DA ROTINA E' MONTAR A LISTA
0450 ----          DE TIPOS DO NO' E COLOCAR COMO UNICO ELEMENTO DESSA LISTA
0451 ----          UM NO' QUE REPRESENTA A CLASSE "TIPO_CARACTER" .
0452 ----
0453 ----          A 2. FASE ( DESCIDA_FINAL ) VERIFICA SE O TIPO RECEBIDO
0454 ----          E' DE ENUMERACAO E , EM CASO POSITIVO , SE O CARACTER E'
0455 ----          UM LITERAL DE ENUMERACAO DESSE TIPO.
0456 ----
0457 -----
0458 ----
0459 PROCEDURE DECORA_INDEXED( PNO : PT_TREE ) IS
0460 -----
0461 PN , PES , PE : PT_TREE ;
0462 BEGIN
0463     PN := AS_NAME( PNO ) ;
0464     PES := AS_EXP_S( PNO ) ;
0465     CRIA_LISTA_TIPO( PNO ) ;
0466     DECORA( PN ) ;
0467     MONTA_TIPO_COMPONENTES( PNO , PN ) ;
0468     PE := AS_LIST( PES ) ;
0469     TRATA_LISTA_DE_INDICES( PNO , PE ) ;
0470 END DECORA_INDEXED ;
0471 ----
0472 -----
0473 ----
0474 PROCEDURE DESCIDA_FINAL_INDEXED( PNO , PL_TIPO ) IS
0475 -----
0476 PN , PES , PTA : PT_TREE ;
0477 BEGIN
0478     PN := AS_NAME( PNO ) ;
0479     PES := AS_EXP_S( PNO ) ;
0480     IF NOT CONTIDO( PL_TIPO , PNO ) THEN

```

```

0481     MSG("EXPRESSAO INVALIDA" , PNO) ?
0482     SM_EXP_TIPO( PNO ) := SUPER_TIPO ?
0483 ELSE
0484     SM_EXP_TYPE( PNO ) := PL_TIPO ?
0485     IF NOT TIPO_UNICO_ARRAY( PN , PL_TIPO ) THEN
0486         MSG("EXPRESSAO INVALIDA" , PN ) ?
0487         SM_EXP_TYPE( PN ) := SUPER_TIPO ?
0488     ELSE
0489         PTA := APANHA_TIPO_ARRAY( PN , PL_TIPO ) ?
0490         DESCIDA_FINAL( PN , PTA ) ?
0491         VERIFICA_LISTA_INDICES( PES , PTA ) ?
0492     END IF ?
0493 END IF ?
0494 END DESCIDA_FINAL_INDEXED ?
0495 ---
0496 ---
0497 ---
0498 --- OBS : - A ROTINA "MONTA_TIPO_COMPONENTES" PERCORRE A LISTA DE TIPOS
0499 --- DA SUBARVORE APONTADA POR "PN" PROCURANDO TIPOS DE ARRAYS?
0500 --- PARA CADA TIPO ENCONTRADO COLOCA O TIPO DA COMPONENTE
0501 --- CORRESPONDENTE NA LISTA DE TIPOS DO NO "INDEXED".
0502 ---
0503 ---
0504 --- - A ROTINA "TRATA_LISTA_DE_INDICES" PERCORRE A LISTA DE
0505 --- INDICES CRIANDO UMA LISTA DE TIPOS PARA CADA INDICE
0506 --- A PARTIR DA LISTA DE TIPO DO NO "INDEXED". A ROTINA
0507 --- "DECORA" E' CHAMADA PARA CADA INDICE PERCORRIDO , E , DE
0508 --- ACORDO COM SEU RESULTADO , A LISTA DE TIPOS DO NO
0509 --- "INDEXED" E' RESTRINGIDA .
0510 ---
0511 ---
0512 PROCEDURE DECORA_SLICE( PNO : PT_TREE ) IS
0513 ---
0514 PN , PDR , PTI : PT_TREE ?
0515 BEGIN
0516     PN := AS_NAME( PNO ) ?
0517     PDR := AS_DESCR_RANGE( PNO ) ?
0518     CRIA_LISTA_TIPO( PNO ) ?
0519     DECORA( PN ) ?
0520     MONTA_TIPO_COMPONENTES( PNO , PN ) ?
0521     IF APANHA_NOME_NO( PDR ) = ON_CONSTRAINED THEN
0522         PTI := CALCULA_TIPO_CONSTRAINED( PDR ) ?
0523     ELSE
0524         PTI := CALCULA_TIPO_RANGE( PDR ) ?
0525     END IF ?
0526     TRATA_INTERVALO( PNO , PTI ) ?
0527 END DECORA_SLICE ?
0528 ---
0529 ---
0530 ---
0531 PROCEDURE DESCIDA_FINAL_SLICE( PNO , PL_TIPO : PT_TREE ) IS
0532 ---
0533 PN , PDR , PTA : PT_TREE ?
0534 BEGIN
0535     PN := AS_NAME( PNO ) ?
0536     PDR := AS_DESCR_RANGE( PNO ) ?
0537     IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0538         MSG(" EXPRESSAO INVALIDA" , PNO) ?
0539         SM_EXP_TYPE( PNO ) := SUPER_TIPO ?
0540     ELSE

```



```

0541     SM_EXP_TYPE( PNO ) := PL_TIPO ;
0542     IF NOT TIPO_UNICO_ARRAY( FN , PL_TIPO ) THEN
0543         MSG("EXPRESSAO INVALIDA" , FN) ;
0544         SM_EXP_TYPE( FN ) := SUPER_TIPO ;
0545     ELSE
0546         PTA := APANHA_TIPO_ARRAY( FN , PL_TIPO ) ;
0547         DESCIDA_FINAL( FN , PTA ) ;
0548     END IF ;
0549 END IF ;
0550 END DESCIDA_FINAL_SLICE ;
0551 ---
0552 ---
0553 ---
0554 ---     OBS: - A ROTINA "TRATA_INTERVALO" E' EQUIVALENTE 'A ROTINA
0555 ---     "TRATA_LISTA_DE_INDICES" JA APRESENTADA .
0556 ---
0557 ---
0558 ---
0559 PROCEDURE DECORA_SELECTED( PNO ; PT_TREE ) IS
0560     -----
0561     FN , PDC ; PT_TREE ;
0562 BEGIN
0563     FN := AS_NAME( PNO ) ;
0564     PDC := AS_DESIGNATOR_CHAR( PNO ) ;
0565     CRIA_LISTA_TIPO( PNO ) ;
0566     DECORA( FN ) ;
0567     MONTA_TIPO_SELECTED( PNO , FN ) ;
0568     DECORA( PDC ) ;
0569     COMPARA_LISTA( PNO , PDC ) ;
0570 END DECORA_SELECTED ;
0571 ---
0572 ---
0573 ---
0574 PROCEDURE DESCIDA_FINAL_SELECTED( PNO , PL_TIPO ; PT_TREE ) IS
0575     -----
0576     FN , PDC , PTR ; PT_TREE ;
0577 BEGIN
0578     FN := AS_NAME( PNO ) ;
0579     PDC := AS_DESIGNATOR_CHAR( PNO ) ;
0580     IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0581         MSG("COMPONENTE SELECIONADO INVALIDO" , PNO) ;
0582         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0583     ELSE
0584         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0585         PTR := APANHA_TIPO_NOME( FN , PNO ) ;
0586         DESCIDA_FINAL( FN , PTR ) ;
0587     END IF ;
0588 END DESCIDA_FINAL_SELECTED ;
0589 ---
0590 ---
0591 ---
0592 ---     OBS: - A ROTINA "MONTA_TIPO_SELECTED" PERCORRE A LISTA DE TIPOS
0593 ---     DO NO' APONTADO PELO 2. PARAMETRO E , PARA CADA NO' EN_
0594 ---     CONTRADO , COLOCA NA LISTA DE TIPOS DO NO' APONTADO PELO
0595 ---     1. PARAMETRO OS TIPOS CABIVEIS (POR EX. SE UM TIPO DE
0596 ---     RECORO FOR ESPECIFICADO NA LISTA DO 1. PARAMETRO , OS
0597 ---     TIPOS DE TODAS SUAS COMPONENTES SERIAM COLOCADOS NA LISTA
0598 ---     DO 2. PARAMETRO .
0599 ---
0600 ---     - A FUNCAO "APANHA_TIPO_NOME" DEVOOLVE O TIPO DA SUBARVORE

```

```

0601 ---- AFONTADA POR "PN" A PARTIR DO TIPO DA SUBARVORE AFONTADA
0602 ---- POR "PNO".
0603 ----
0604 -----
0605 ----
0606 PROCEDURE DECORA_ATTRIBUTE( PNO : PT_TREE ) IS
0607 -----
0608 PN , PID , PTA : PT_TREE ;
0609 BEGIN
0610     PN := AS_NAME( PNO ) ;
0611     PID := AS_ID( PNO ) ;
0612     IF E_ATRIBUTO( PID ) THEN
0613         PTA := APANHA_TIPO_ATRIBUTO( PID ) ;
0614         CRIA_LISTA_TIPO( PNO ) ;
0615         COLOCA_NA_LISTA( PNO , PTA ) ;
0616         DECORA( PN ) ;
0617     ELSE
0618         MSG("ATRIBUTO INVALIDO", PID) ;
0619         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0620     END IF ;
0621 END DECORA_ATTRIBUTE ;
0622 ----
0623 -----
0624 ----
0625 PROCEDURE DESCIDA_FINAL_ATTRIBUTE( PNO , PL_TIPO : PT_TREE ) IS
0626 -----
0627 PN , PLT : PT_TREE ;
0628 BEGIN
0629     PN := AS_NAME( PNO ) ;
0630     IF NOT CONTIDO( PNO , PL_TIPO ) THEN
0631         MSG("EXPRESSAO INVALIDA", PNO) ;
0632         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0633     ELSIF NOT PREFIXO_OK_PARA_ATRIBUTO( PN , PID ) THEN
0634         MSG("TIPO DO PREFIXO INVALIDO PARA ATRIBUTO" , PN ) ;
0635         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0636     ELSE
0637         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0638         PLT := APANHA_TIPO( PN ) ;
0639         DESCIDA_FINAL( PN , PLT ) ;
0640     END IF ;
0641 END DESCIDA_FINAL_ATTRIBUTE ;
0642 ----
0643 -----
0644 ----
0645 --- OBS: - A FUNCAO "E_ATRIBUTO" VERIFICA SE O NO' AFONTADO POR SEU
0646 --- PARAMETRO REPRESENTA 1 ATRIBUTO (ADDRESS , AFT , BASE ... )
0647 ---
0648 --- - A FUNCAO "APANHA_TIPO_ATRIBUTO" DEVOLVE O TIPO DO ATRIBU
0649 --- TO REPRESENTADO PELO NO' AFONTADO POR SEU PARAMETRO .
0650 --- ( POR EX. , O ATRIBUTO "ADDRESS" TEM O TIPO ADDRESS DEFI
0651 --- NIDO NO PACOTE "SYSTEM" )
0652 ---
0653 --- - A FUNCAO "PREFIXO_OK_PARA_ATRIBUTO" VERIFICA SE O TIPO DA
0654 --- SUB-ARVORE AFONTADA PELO 1. PARAMETRO E' VALIDO PARA O
0655 --- ATRIBUTO AFONTADO PELO 2. PARAMETRO (POR EX. O ATRIBUTO
0656 --- "FIRST" SO' E' VALIDO COM 1 PREFIXO DO TIPO ESCALAR )
0657 ---
0658 --- - A FUNCAO "APANHA_TIPO" DEVOLVE O TIPO DA SUBARVORE AFON
0659 --- TADA POR SEU PARAMATRO (NOTE QUE ESTA SUBARVORE REPRESENTA
0660 --- 1 PREFIXO VALIDO PARA O ATRIBUTO EM QUESTAO ) .

```

```

0661 ----
0662 -----
0663 ----
0664 PROCEDURE DECORA_ATTRIBUTE_CALL( PNO : PT_TREE ) IS
0665 -----
0666 PN , PE : PT_TREE ;
0667 BEGIN
0668     FN := AS_NAME( PNO ) ;
0669     PE := AS_EXP( PNO ) ;
0670     DECORA( PN ) ;
0671     CRIA_LISTA_TIPO( PNO ) ;
0672     COPIA_LISTA( PNO , PN ) ;
0673     DECORA( PE ) ;
0674 END DECORA_ATTRIBUTE_CALL ;
0675 ----
0676 -----
0677 ----
0678 PROCEDURE DESCIDA_FINAL_ATTRIBUTE_CALL( PNO : PT_TREE ) IS
0679 -----
0680 PN , PE : PT_TREE ;
0681 BEGIN
0682     FN := AS_NAME( PNO ) ;
0683     PE := AS_EXP( PNO ) ;
0684     IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0685         MSG("ATTRIBUTE_CALL INVALIDO" , PNO ) ;
0686         SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0687     ELSE
0688         SM_EXP_TYPE( PNO ) := PL_TIPO ;
0689         DESCIDA_FINAL( FN , PL_TIPO ) ;
0690         DESCIDA_FINAL( PE , "INTEGER" ) ;
0691     END IF ;
0692 END DESCIDA_FINAL_ATTRIBUTE_CALL ;
0693 ----
0694 -----
0695 ----
0696 PROCEDURE DECORA_NOME( PNO : PT_TREE ) IS
0697 -----
0698 PNO , PDEF , PTO : PT_TREE ;
0699 BEGIN
0700     IF NOT DECLARADO( PNO ) THEN
0701         MSG("IDENTIFICADOR NAO DECLARADO" , PNO) ;
0702     ELSE
0703         PDEF := APONTA_DEFINICAO( PNO ) ;
0704         IF E_OBJETO( PDEF ) THEN
0705             TRANSFORMA_USED_OBJECT_ID( PNO ) ;
0706             PTO := APANHA_TIPO_OBJETO( PDEF ) ;
0707             MONTA_LISTA_TIPO( PNO ) ;
0708             COLOCA_TIPOA( PNO , PTO ) ;
0709         ELSIF E_NAME( PDEF ) THEN
0710             TRANSFORMA_USED_NAME_ID( PNO ) ;
0711             PTN := APANHA_TIPO_NONE( PDEF ) ;
0712             MONTA_LISTA_TIPO( PNO ) ;
0713             COLOCA_TIPO( PNO , PTO ) ;
0714         ELSE
0715             -- FUNCTION_CALL SEM PARAMETROS
0716             TRANSFORMA_FUNCTION_CALL( PNO ) ;
0717             DECORA( PNO ) ;
0718         END IF ;
0719     END IF ;
0720 END DECORA_NOME ;
0721 ----

```

```

0721 -----
0722 --
0723 PROCEDURE DESCIDA_FINAL_NOME ( PNO , PL_TIPO ) IS
0724 -----
0725 BEGIN
0726     IF NOT CONTIDO( PL_TIPO , PNO ) THEN
0727         MSG("TIPO DE NOME OU IDENTIFICADOR INVALIDO", PNO ) ;
0728     END IF ;
0729 END DESCIDA_FINAL_NOME ;
0730 -----
0731 -----
0732 --
0733 --- OBS: - A FUNCAO "E_OBJETO" VERIFICA SE O IDENTIFICADOR DESIGNADO
0734 --- POR SEU PARAMETRO E' UM OBJETO SEM CASO AFIRMATIVO , A
0735 --- ROTINA "TRANSFORMA_USED_OBJECT_ID" TRANSFORMA O NO' EM
0736 --- UM NO' DO TIPO "USED_OBJECT_ID" .
0737 ---
0738 --- - AS MESMAS OBSERVAÇÕES SÃO VALIDAS PARA AS ROTINAS
0739 --- "E_NAME" E "TRANSFORMA_USED_NAME_ID" .
0740 ---
0741 -----
0742 --
0743 PROCEDURE DECORA_FUNCTION_CALL( PNO ; PT_TREE ) IS
0744 -----
0745 PPAS , PN , P1 , P2 , PPAR : PT_TREE ;
0746 POR_NOME : BOOLEAN := FALSE ;
0747 BEGIN
0748     PPAS := AS_PARAM_ASSOC_S( PNO ) ;
0749     PN := AS_NAME( PNO ) ;
0750     CRIA_LISTA_TIPO( PNO ) ;
0751     CRIA_LISTA_PARAMATRO( PNO ) ;
0752     PPAR := AS_LIST( PPAS ) ;
0753 ---
0754 --- PERCORRE PARAMETROS
0755 ---
0756 WHILE PPAR /= NULL
0757 LOOP
0758     IF APANHA_NOME_NO( PPAR ) = DN_ASSOC THEN
0759         POR_NOME := TRUE ;
0760         P1 := AS_DESIGNATOR( PPAR ) ;
0761         P2 := AS_ACTUAL( PPAR ) ;
0762         IF NOT E_PARAMETRO( PNO , P1 ) THEN
0763             MSG("ASSOCIACAO POR NOME INVALIDA" , P1 ) ;
0764             SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0765             EXIT ;
0766         ELIF JA_MARCADO( PNO , P1 ) THEN
0767             MSG("PARAMATRO JA FOI USADO" , P1 ) ;
0768             SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0769             EXIT ;
0770         ELSE
0771             MARCA( PNO , P1 ) ;
0772             DECORA( P2 ) ;
0773             COMPARA_DFCOES_FUNCDOES( PNO , P1 ) ;
0774         END IF ;
0775     ELSE -- ASSOCIACAO POR POSICAO
0776         IF POR_NOME THEN
0777             MSG("NAO ERA ESPERADA ASSOCIACAO POR POSICAO" , PPAR ) ;
0778             SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0779             EXIT ;
0780         ELSE

```

```

0781 MARCA( PNO , PPAR ) ;
0782 DECORA( PPAR ) ;
0783 COMPARA OPCOES_FUNCDES( PNO , PPAR ) ;
0784 END IF ;
0785 END IF ;
0786 END LOOP ;
0787 CORTA_LISTA( PNO ) ;
0788 END DECORA_FUNCTION_CALL ;
0789 ---
0790 -----
0791 ---
0792 PROCEDURE DESCIDA_FINAL_FUNCTION_CALL( PNO , PL_TIPO : PL_TREE ) IS
0793 -----
0794 PPAR , PN , P1 , P2 : PL_TREE ;
0795 BEGIN
0796 IF NOT SOBROU_FUNCDAO( PNO ) THEN
0797 MSG("CHAMADA DE FUNCAO INDEFINIDA" , PNO ) ;
0798 SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0799 ELSIF NOT CONTIDO( PL_TIPO , PNO ) THEN
0800 MSG("CHAMADA DE FUNCAO INVALIDA" , PNO ) ;
0801 SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0802 ELSIF NOT FUNCAO_UNICA( PNO ) THEN
0803 MSG("CHAMADA DE FUNCAO AMBIGUA" , PNO ) ;
0804 SM_EXP_TYPE( PNO ) := SUPER_TIPO ;
0805 ELSE
0806 SM_EXP_TYPE( PNO ) := PL_TIPO ;
0807 PPAR := AS_PARAM_ASSOC_S( PNO ) ;
0808 PPAR := AS_LIST( PPAR ) ;
0809 WHILE PPAR /= NULL
0810 LOOP
0811 IF APANHA_NOME_NO( PPAR ) = DN_ASSOC THEN
0812 P1 := AS_DESIGNATOR( PPAR ) ;
0813 P2 := AS_ACTUAL( PPAR ) ;
0814 PTIPO := APANHA_TIPO_PARAMETRO_REAL( P1 ) ;
0815 DESCIDA_FINAL( P2 , PTIPO ) ;
0816 ELSE
0817 PTIPO := APANHA_TIPO_PROXIMO_PARAMETRO( PPAR ) ;
0818 DESCIDA_FINAL( PPAR , PTIPO ) ;
0819 END IF ;
0820 END LOOP ;
0821 END IF ;
0822 END DESCIDA_FINAL_FUNCTION_CALL ;
0823 ---
0824 -----
0825 ---
0826 --- OBS : - A ROTINA "CRIA_LISTA_PARAMETRO" CONSTROI UMA LISTA
0827 --- COM INFORMACOES SOBRE OS PARAMETROS DE CADA UMA DAS
0828 --- FUNCDES QUE TENHAM O MESMO DESIGNADOR USADO NA CHAMADA .
0829 ---
0830 --- - DURANTE O 1. PERCURSO NA ARVORE E' VERIFICADA SE A
0831 --- ASSOCIACAO DE PARAMETROS NOMINAL / POSICIONAL ESTA
0832 --- CORRETA . DESSA FORMA , O 2. PASSO NA ARVORE ( ROTINA
0833 --- "DESCIDA_FINAL_FUNCTION_CALL" ) NAO PRECISA FAZER ESSES
0834 --- TESTES .
0835 ---
0836 --- - A ROTINA "MARCA" MARCA OS ELEMENTOS DAS LISTAS CONSTRUI
0837 --- DAS QUE JA FORAM ESPECIFICADAS .
0838 ---
0839 --- - A ROTINA "COMPARA OPCOES_FUNCDES" USA AS INFORMACOES
0840 --- CONTIDAS NAS LISTAS DE PARAMETROS E O RESULTADO DA

```

0841 --- ROTINA "DECORA" PARA ELIMINAR ALGUMAS FUNCOES .
0842 ---
0843 --- - A ROTINA "TESTA_RETORNO" VERIFICA SE EXISTE ALGUMA
0844 --- FUNCAO CUJO TIPO SEJA IGUAL AO TIPO PASSADO PELO PAI
0845 --- DA SUB_ARVORE DA CHAMADA DA FUNCAO , ELIMINANDO AS
0846 --- FUNCOES QUE NAO APRESENTEM ESSA CARACTERISTICA . ESSA
0847 --- MESMA FUNCAO EMITE MENSAGEM DE ERRO NO CASO DE NAO
0848 --- SOBRAR NENHUMA FUNCAO NA LISTA , OU SE SOBRAR MAIS QUE
0849 --- UMA FUNCAO , DEVOLVENDO O VALOR "FALSE" NESSE CASO .
0850 ---
0851 --- - A ROTINA "DESCIDA_FINAL_FUNCTION_CALL" VERIFICA SE
0852 --- SOBROU ALGUMA FUNCAO APÓS O PROCESSAMENTO DOS PARAMETROS
0853 --- ("SOBROU_FUNCAO") . EM CASO AFIRMATIVO VERIFICA SE OS
0854 --- TIPOS DAS FUNCOES SÃO COMPATIVÉIS COM O CONTEXTO
0855 --- ("CONTIDO") E FINALMENTE SE EXISTE UMA ÚNICA FUNCAO
0856 --- COM O TIPO ESPECIFICADO ("FUNCAO_UNICA") .
0857 ---
0858 ---
0859 ---
***** FIM DO ARQUIVO.

APÊNDICE 3

ESPECIFICAÇÃO DAS ROTINAS
DE ANÁLISE SEMÂNTICA

```

0001 ---
0002 --- BINARY => AS_EXP1 : EXP ;
0003 --- AS_BINARY_OF : BINARY_OF ;
0004 --- AS_EXP2 : EXP ;
0005 --- SM_EXP_TYPE : TYPE_SPEC ;
0006 --- SM_VALUE : VALUE ;
0007 ---
0008 ---
0009 --- TODOS OS TESTES SEMANTICOS DESSE NO JA FORAM REALIZADOS PELA
0010 --- ROTINA DE RESOLUCAO DE EXPRESSOES.
0011 ---
0012 --- BLOCK => AS_ITEM_S : ITEM_S ;
0013 --- AS_STM_S : STM_S ;
0014 --- AS_ALTERNATIVE_S : ALTERNATIVE_S ;
0015 ---
0016 PROCEDURE VERIFICA_BLOCK(PNO : PT_TREE) IS
0017 PITEM , PSTM , FALT : PT_TREE ;
0018 BEGIN
0019 INICIO_DE_BLOCK(PNO) ;
0020 PITEM := AS_ITEM_S(PNO) ;
0021 PSTM := AS_STM_S(PNO) ;
0022 FALT := AS_ALTERNATIVE_S(PNO) ;
0023 VERIFICA_ITEM_S(PITEM) ;
0024 TIPOS_COMPLETOS(PITEM) ;
0025 SUBPROGRAMAS_COMPLETOS(PITEM) ;
0026 VERIFICA_STM_S(PSTM) ;
0027 VERIFICA_ALTERNATIVE_S_BLOCK(FALT) ;
0028 FINAL_DE_BLOCK(PNO) ;
0029 END VERIFICA_BLOCK ;
0030 -----
0031 ---
0032 --- SE UMA DECLARACAO DE TIPO INCOMPLETO OCORRER EM UMA PARTE
0033 --- DECLARATIVA, ENTAO A DECLARACAO DE TIPO COMPLETO DEVE OCORRER APOS
0034 --- E NA MESMA PARTE DECLARATIVA, A ROTINA "TIPOS_COMPLETOS" EFETUA
0035 --- ESSES TESTES (ARM-SC.3.8.1-3).
0036 --- AS REGRAS APRESENTADAS NAS SECDOES 3.9 E 7.1 EXIGEM QUE A
0037 --- DECLARACAO DE UM SUBPROGRAMA E O SEU CORPO CORRESPONDENTE OCORRAM
0038 --- NA MESMA PARTE DECLARATIVA (ARM-SC.6.3-7); A ROTINA
0039 --- "SUBPROGRAMAS_COMPLETOS" EFETUA ESSES TESTES.
0040 --- A ROTINA "VERIFICA_ALTERNATIVE_S_BLOCK" E RESPONSAVEL PELOS TESTES
0041 --- DA CONSTRUCAO "CHOICE" (SO NOME DE IDENTIFICADORES DE EXCECOES E
0042 --- OPCAO "OTHERS" COMO A ULTIMA - ARM-SC.11.2-5).
0043 --- NOTE QUE O NO "ALTERNATIVE_S" PODE OCORRER EM 2 CONTEXTOS COM
0044 --- SEMANTICAS DIFERENTES (BLOCK E COMANDO CASE); POR ISSO A
0045 --- NECESSIDADE DE 2 ROTINAS DIFERENTES.
0046 ---
0047 --- BOX => LX_SRCPOS : SOURCE_POSITION ,
0048 --- LX_COMMENTS : COMMENTS ;
0049 ---
0050 --- OBS : NAO HA NECESSIDADE DE TESTES SEMANTICOS PARA ESSE NO.
0051 ---
0052 --- CASE => AS_EXP : EXP ;
0053 --- AS_ALTERNATIVE_S : ALTERNATIVE_S ;
0054 ---
0055 PROCEDURE VERIFICA_CASE(PNO : PT_TREE) IS
0056 PEXP , FALT : PT_TREE ;
0057 BEGIN
0058 PEXP := AS_EXP(PNO) ;
0059 FALT := AS_ALTERNATIVE_S(PNO) ;
0060 RESOLVE_NOME_E_EXPRESSAO(PEXP) ;

```



```

0061 VERIFICA_EXP(PEXP) ;
0062 IF APANHA_TIPO_EXP(PEXP) /- TIPO_GERAL THEN
0063     IF E_TIPO_GENERICO(PEXP) THEN
0064         MSG("TIPO DA EXPRESSAO NAO DEVE SER GENERICO",PEXP) ;
0065     ELSE
0066         VERIFICA_ALTERNATIVE_ELSE_CASE(PALT,PN0) ;
0067     END IF ;
0068 END IF ;
0069 END VERIFICA_CASE ;
0070 -----
0071 --- O TIPO DA EXPRESSAO DO COMANDO "CASE" NAO DEVE SER GENERICO
0072 --- (ARM-SC.5.4-3) ; A ROTINA "E_TIPO_GENERICO" VERIFICA ESSE FATO.
0073 --- A ROTINA "VERIFICA_ALTERNATIVE_ELSE_CASE" VERIFICA SE O TIPO DAS
0074 --- EXPRESSOES DAS DIVERSAS OPCOES IGUAL AO TIPO DA EXPRESSAO DO
0075 --- COMANDO, SE SAO ESTATICAS E SE TODOS OS VALORES DO TIPO BASE DA
0076 --- EXPRESSAO ESTAO REPRESENTADOS UMA E SO UMA VEZ NO CONJUNTO DE
0077 --- ALTERNATIVAS; ESSA ROTINA TAMBEM VERIFICA O POSICIONAMENTO CORRETO
0078 --- DA OPCAO "OTHERS", SE ELA OCORRER.
0079 --- NOTE QUE SE HOUVER QUALQUER ERRO NA EXPRESSAO DO COMANDO, SUA
0080 --- SEQUENCIA DE ALTERNATIVAS NAO E VERIFICADA.
0081 --- O FATO DA EXPRESSAO DO COMANDO PRECISAR SER DO TIPO DISCRETO E
0082 --- USADO NA ROTINA DE RESOLUCAO DE NOME E EXPRESSAO, O QUE ELIMINA
0083 --- A NECESSIDADE DESSE TESTE NESSE PONTO.
0084 ---
0085 --- CHOICE_S => AS_LIST ; SEQ OF CHOICE ;
0086 ---
0087 --- ESSE NO PODE APARECER EM 2 CONTEXTOS BASTANTES DIFERENTES : COMO
0088 --- FILHO DE UM NO "VARIANT", NA PARTE VARIANTE DE UM RECORD, OU COMO
0089 --- FILHO DE UM NO "NAMED", EM UM AGREGADO. DEVIDO A COMPLEXIDADE DOS
0090 --- TESTES SEMANTICOS NECESSARIOS E A DIFERENCA DA SEMANTICA Nesses 2
0091 --- CONTEXTOS, A SUB-ARVORE DE "CHOICE_S" SERA VERIFICADA DENTRO DAS
0092 --- ROTINAS "VERIFICA_VARIANT_PART" E "VERIFICA_AGGREGATE".
0093 ---
0094 ---
0095 --- CODE => AS_NAME ; NAME ;
0096 ---     AS_EXP ; EXP ;
0097 ---
0098 PROCEDURE VERIFICA_CODE(PNO ; PT_TREE) IS
0099 PN , PEXP ; PT_TREE ;
0100 BEGIN
0101     PN := AS_NAME(PNO) ;
0102     PEXP := AS_EXP(PNO) ; --- AGGREGATE
0103     IF NOT CONTEXTO_SUBPROGRAMA_BODY(PNO) THEN
0104         MSG("COMANDO CODE SO E PERMITIDO EM CORPO DE SUBPROGRAMA") ;
0105     ELSE
0106         RESOLVE_NOME_E_EXPRESSAO(PN) ;
0107         VERIFICA_NAME(PN) ;
0108         IF NOT (APANHA_NOME_NO(PN) = DN_USED_NAME_ID AND THEN
0109             E_RECORD(PN)) THEN
0110             MSG("ESPERADO DESIGNADOR DE RECORD",PN) ;
0111         ELSE
0112             SM_EXP_TYPE(PN) := APANHA_TIPO(PN) ;
0113             VERIFICA_AGGREGATE(PEXP) ;
0114         END IF ;
0115     END IF ;
0116 END VERIFICA_CODE ;
0117 -----
0118 ---
0119 --- UM COMANDO "CODE" SO E PERMITIDO NA SEQUENCIA DE COMANDOS DE UM
0120 --- SUBPROGRAMA (ARM-SC.13.8-3) ; A ROTINA "CONTEXTO_SUBPROGRAMA_BODY"

```

```

0121 --- FAZ ESSE TESTE.
0122 --- CADA INSTRUCAO DE MAQUINA APARECE COMO UM AGREGADO DE RECORDI QUE
0123 --- DEFINE A INSTRUCAO CORRESPONDENTE (ARM-SC.13.8-4) E ESSAS CONDICAOES
0124 --- SAO TESTADAS NO 2. COMANDO "IF".
0125 --- A VERIFICACAO DAS OUTRAS CONDICAOES QUE O SUBPROGRAMA DEVE SATISFAZER
0126 --- (SO CONTER COMANDOS DE INSERCAO DE CODIGO, NAO TER "EXCEPTION
0127 --- HANDLER, ETC) E FEITA NA ROTINA "VERIFICA_SUBPROGRAM_BODY".
0128 ---
0129 --- COMP_ID => SM_OBJ_TYPE : TYPE_SPEC ;
0130 ---             SM_INIT_EXP : EXP_VOID ;
0131 ---             SM_COMP_SPEC : COMP_REP_VOID ;
0132 ---
0133 PROCEDURE VERIFICA_COMP_ID(PNO,PTIPO,PEXP,PPREC : PT_TREE) IS
0134 BEGIN
0135     IF NOT NOME_DISTINTO(PREC , PNO) THEN
0136         MSG("IDENTIFICADOR JA FOI USADO NESTE RECORD",PNO) ;
0137     ELSE
0138         SM_OBJ_TYPE(PNO) := PTIPO ;
0139         SM_INIT_EXP(PNO) := PEXP ;
0140     END IF ;
0141 END VERIFICA_COMP_ID ;
0142 -----
0143 ---
0144 --- OS IDENTIFICADORES DE TODOS COMPONENTES DE UM RECORDI DEVEM SER
0145 --- DISTINTOS (ARM-SC.3.7-3) E ISSO E TESTADO PELA ROTINA "NOME_DISTINTO",
0146 --- QUE TEM COMO PARAMETRO O PONTEIRO DO NO "RECORDI" DO TIPO
0147 --- CORRESPONDENTE.
0148 ---
0149 --- COMP_REF => AS_NAME : NAME ;
0150 ---             AS_EXP : EXP ;
0151 ---             AS_RANGE : RANGE ;
0152 ---
0153 --- COMP_REF_S => AS_LIST : SEQ OF COMP_REF ;
0154 ---
0155 PROCEDURE VERIFICA_COMP_REF(PNO,PNREC : PT_TREE) IS
0156 PN,PEXP,PRG : PT_TREE ;
0157 BEGIN
0158     PN := AS_NAME(PNO) ;
0159     PEXP := AS_EXP(PNO) ;
0160     PRG := AS_RANGE(PNO) ;
0161     RESOLVE_NOME_E_EXPRESSAO(PN) ;
0162     VERIFICA_NAME(PN) ;
0163     RESOLVE_NOME_E_EXPRESSAO(PEXP) ;
0164     VERIFICA_EXP(PEXP) ;
0165     VERIFICA_RANGE(PRG) ;
0166     IF NOT (EXPRESSAO_ESTATICA(PEXP) AND THEN
0167             TIPO_INTEIRO(PEXP)) THEN
0168         MSG("EXPRESSAO DEVE SER ESTATICA E TER TIPO INTEIRO",PEXP) ;
0169     ELSIF NOT (RANGE_ESTATICO(PRG) AND THEN
0170             TIPO_RANGE_INTEIRO(PRG)) THEN
0171         MSG("RANGE DEVE SER ESTATICO E TER TIPO INTEIRO",PRG) ;
0172     ELSIF NOT COMPONENTE_VALIDA(PN,PNREC) THEN
0173         MSG("COMPONENTE INVALIDA PARA RECORD",PN) ;
0174     ELSIF ESPECIFICADA(PN,PNREC) THEN
0175         MSG("COMPONENTE JA FOI ESPECIFICADA NA CLAUSULA",PN) ;
0176     ELSIF NOT CONSTRAINT_ESTATICA(PN,PNREC) THEN
0177         MSG("CONSTRAINT NAO ESTATICA APLICADA A COMPONENTE",PN) ;
0178     ELSIF NOT ESPACO_SUFICIENTE(PN,PNREC) THEN
0179         MSG("ESPACO INSUFICIENTE PARA VALORES DA COMPONENTE",PN) ;
0180     ELSE

```

```

0181     PC := APANHA_DEFINICAO_COMPONENTE(FN,PNREC) ;
0182     SM_COMP_SPEC(PC) := PNO ;
0183     END IF ;
0184 END VERIFICA_COMP_REF ;
0185 -----
0186 ---
0187 --- A EXPRESSAO DA CLAUSULA DE REPRESENTACAO DEVE SER ESTATICA E SER
0188 --- DE UM TIPO INTEIRO(ARM-SC.15.4-3) ; ESSAS CONDICOES SAO VERIFICADAS
0189 --- PELAS ROTINAS "EXPRESSAO_ESTATICA" E "TIPO_INTEIRO".
0190 --- AS MESMAS OBSERVACOES SAO VALIDAS PARA O INTERVALO ("RANGE") DA
0191 --- CLAUSULA ; AS ROTINAS "RANGE_ESTATICO" E "TIPO_RANGE_INTEIRO" SAO
0192 --- USADAS PARA OS TESTES CORRESPONDENTES.
0193 --- A COMPONENTE ESPECIFICADA DEVE SER UM CAMPO DO RECORD PARA O QUAL
0194 --- FOI APLICADA A CLAUSULA ; ISSO E TESTADO PELA FUNCAO
0195 --- "COMPONENTE_VALIDA".
0196 --- NO MAXIMO UMA CLAUSULA E PERMITIDA PARA CADA COMPONENTE DO RECORD
0197 --- EM QUESTAO(ARM-SC.15.4-6) ; A FUNCAO "ESPECIFICADA" VERIFICA SE JA
0198 --- FOI APLICADA UMA CLAUSULA NA COMPONENTE QUE ESTA SENDO ANALISADA.
0199 --- ESSA CLAUSULA SO PODE SER APLICADA A UMA COMPONENTE SE ALGUMA
0200 --- RESTRICAO DA COMPONENTE ("CONSTRAINT") OU DE UMA DE SUAS
0201 --- SUBCOMPONENTES FOREM ESTATICAS(ARM-SC.15.4-7) ; ESSE FATO E VERIFICADO
0202 --- PELA FUNCAO "CONSTRAINT_ESTATICA".
0203 --- ESSA CLAUSULA DEVE ESPECIFICAR ESPACO SUFICIENTE PARA ACOMODAR
0204 --- QUALQUER VALOR VALIDO DA COMPONENTE(ARM-SC.13.4-7) ; A FUNCAO
0205 --- "ESPACO_SUFICIENTE" REALIZA O TESTE NECESSARIO, QUE E DEPENDENTE
0206 --- DE MAQUINA.
0207 ---
0208 --- COMP_REP_S => AS_LIST : SEQ OF COMP_REF ;
0209 ---
0210 PROCEDURE VERIFICA_COMP_REP_S(PNO,PNREC : PT_TREE) IS
0211 PCR : PT_TREE ;
0212 BEGIN
0213     PCR := AS_LIST(PNO) ;
0214     WHILE PCR /= NIL
0215     LOOP
0216         VERIFICA_COMP_REF(PCR,PNREC) ;
0217         PCR := NEXT(PCR) ;
0218     END LOOP ;
0219     OVERLAPING(PNREC,PNO) ;
0220 END VERIFICA_COMP_REP_S ;
0221 -----
0222 ---
0223 --- A ALOCACAO DE ESPACO PARA AS DIVERSAS COMPONENTES DE UMA MESMA
0224 --- PARTE VARIANTE DE UM RECORD NAO PODE SER SOBREPUSTA, MAS A
0225 --- SOBREPOSICAO E PERMITIDA PARA PARTES VARIANTES(ARM-SC.13.4-7) ; A
0226 --- ROTINA "OVERLAPING" VERIFICA SE ESSE FATO OCORRE, USANDO A SUB-
0227 --- ARVORE DA DECLARACAO DO RECORD.
0228 --- NOTE QUE NAO E NECESSARIO QUE TODAS COMPONENTES DO RECORD EM
0229 --- QUESTAO TENHAM UMA CLAUSULA DE REPRESENTACAO PROPRIA(ARM-SC.13.4-6)
0230 --- O 2. PARAMETRO DESSA ROTINA APONTA PARA A SUB-ARVORE EM QUE O
0231 --- RECORD QUE ESTA SENDO ESPECIFICADO FOI DECLARADO ; NOTE QUE ESSA
0232 --- SUB-ARVORE E MODIFICADA PELA ROTINA "VERIFICA_COM_rep".
0233 ---
0234 --- COMPILATION => AS_LIST : SEQ OF COMP_UNIT ;
0235 ---
0236 --- COMP_UNIT => AS_CONTEXT : CONTEXT ;
0237 ---                 AS_UNIT_BODY : UNIT_BODY ;
0238 ---                 AS_PRAGMA_S : PRAGMA_S ;
0239 ---
0240 PROCEDURE VERIFICA_COMPILATION(PNO : PT_TREE) IS

```

```

0241 PCV := PT_TREE ;
0242 BEGIN
0243     PCV := AS_LIST(PNO) ;
0244     WHILE PCV /= NIL
0245     LOOP
0246         VERIFICA_COMP_UNIT(PCV) ;
0247         PCV := NEXT(PCV) ;
0248     END LOOP ;
0249 END VERIFICA_COMPILATION ;
0250 -----
0251 PROCEDURE VERIFICA_COMP_UNIT(PNO : P_TREE) IS
0252 PCTX,PBD,PPB : PT_TREE ;
0253 BEGIN
0254     PCTX := AS_CONTEXT(PNO) ;
0255     PBD := AS_UNIT_BODY(PNO) ;
0256     PPB := AS_PRAGMA_S(PNO) ;
0257     VERIFICA_CONTEXTO(PCTX) ;
0258     VERIFICA_UNIT_BODY(PBD) ;
0259     VERIFICA_PRAGMA_S(PPB) ;
0260     IF NOT ERRO THEN
0261         TAMBORETE ;
0262     END IF ;
0263 END VERIFICA_COMP_UNIT ;
0264 -----
0265 ---
0266 --- SE UMA UNIDADE DE COMPILACAO FOR ANALISADA COM SUCESSO, O TAMBORETE
0267 --- E CHAMADO PARA IMPRIMIR AS INFORMACOES RELEVANTES DA UNIDADE NA
0268 --- BIBLIOTECAQUE ESTIVER SENDO USADA.
0269 --- O AMBIENTE NECESSARIO PARA COMPILACAO DA UNIDADE E MONTADO PELO
0270 --- TAMBORETE ATRAVES DE CHAMADAS FEITAS PELA ROTINA
0271 --- "VERIFICA_CONTEXT".
0272 ---
0273 --- COND_CLAUSE => AS_EXP_VOID : EXP_VOID ;
0274 ---                 AS_STM_S : STM_S ;
0275 ---
0276 PROCEDURE VERIFICA_COND_CLAUSE(PNO : PT_TREE) IS
0277 PEXP , PSTM : PT_TREE ;
0278 BEGIN
0279     PEXP := AS_EXP_VOID(PNO) ;
0280     PSTM := AS_STM_S(PNO) ;
0281     IF APANHA_NOME_NO(PEXP) /= DN_VOID THEN
0282         RESOLVE_NOME_E_EXPRESSAO(PEXP) ;
0283         VERIFICA_EXP(PEXP) ;
0284     END IF ;
0285     VERIFICA_STM_S(PSTM) ;
0286 END VERIFICA_COND_CLAUSE ;
0287 -----
0288 ---
0289 --- A EXPRESSAO DE UM COMANDO "IF" DEVE SER BOOLEANA? ISSO E TESTADO
0290 --- DENTRO DA ROTINA "RESOLVE_NOME_E_EXPRESSAO", QUE USA ESSE FATO
0291 --- PARA DEFINIR O TIPO DA EXPRESSAO.
0292 ---
0293 --- COND_ENTRY => AS_STM_S1 : STM_S ;
0294 ---                AS_STM_S2 : STM_S ;
0295 ---
0296 PROCEDURE VERIFICA_COND_ENTRY(PNO : PT_TREE) IS
0297 PS1 , PS2 : PT_TREE ;
0298 BEGIN
0299     PS1 := AS_STM_S1(PNO) ;
0300     PS2 := AS_STM_S2(PNO) ;

```

```

0301 IF NOT DENTRO_DE_TASK(PNO) THEN
0302     MSG("CONDICIONAL_ENTRY_CALL SO DEVE OCORRER EM UMA TASK",PNO) ;
0303 ELSE
0304     VERIFICA_STM_S(P51) ;
0305     VERIFICA_STM_S(P52) ;
0306 END IF ;
0307 END VERIFICA_COND_ENTRY ;
0308 -----
0309 ---
0310 --- A PROPRIA FUNCAO DESSE COMANDO EXIGE QUE ELE OCORRA NO CORPO DE
0311 --- UMA TASK? ISSO E TESTADO PELA FUNCAO "DENTRO_DE_TASK".
0312 --- A SINTAXE DA LINGUAGEM OBRIGA QUE O 1. COMANDO DA 1. SEQUENCIA
0313 --- DE COMANDOS SEJA UMA "ENTRY_CALL"? DESSA FORMA, NAO E NECESSARIO
0314 --- TESTAR ESSE FATO NESTA ROTINA.
0315 ---
0316 --- CONST_ID => SM_ADDRESS : EXP_VOID ,
0317 ---                SM_OBJ_TYPE : TYPE_SPEC ,
0318 ---                SM_OBJ_DEF  : OBJECT_DEF ,
0319 ---                SM_FIRST   : DEF_OCCURRENCE ;
0320 ---
0321 --- CONST_ID => SM_ADDRESS : EXP_VOID ,
0322 ---                SM_OBJ_TYPE : TYPE_SPEC ,
0323 ---                SM_OBJ_DEF  : OBJECT_DEF ,
0324 ---                SM_FIRST   : DEF_OCCURRENCE ;
0325 ---
0326 PROCEDURE VERIFICA_CONST_ID(PNO ,PPAI ; FT_TREE) IS
0327 PP : FT_TREE ;
0328 BEGIN
0329     IF DECLARADO(PNO) THEN
0330         IF DECLARACAO_COMPLETA(PNO) THEN
0331             PP := PRIMEIRA_DECLARACAO(PNO) ;
0332             IF NOT PARTE_PRIVADA(PNO) THEN
0333                 MSG("DECLARACAO DEVEIA OCORRER EM PARTE PRIVADA",PNO) ;
0334             ELSIF NOT PRIMEIRA_COMPLEMENTACAO(PP) THEN
0335                 MSG("CONSTANTE JA FOI DEFINIDA",PNO) ;
0336             ELSE
0337                 SM_OBJ_TYPE(PP) := AS_TYPE_SPEC(PPAI) ;
0338                 SM_OBJ_TYPE(PNO) := AS_TYPE_SPEC(PPAI) ;
0339                 SM_OBJ_DEF(PP) := AS_OBJECT_DEF(PPAI) ;
0340                 SM_OBJ_DEF(PNO) := AS_OBJECT_DEF(PPAI) ;
0341                 SM_FIRST(PNO) := PP ;
0342             END IF ;
0343         ELSE
0344             MSG("IDENTIFICADOR JA USADO",PNO) ;
0345         END IF ;
0346     ELSE -- 1. DECLARACAO
0347         IF APANHA_NOME_NO(PPAI) = DN_DEFERRED THEN
0348             SM_FIRST(PNO) := PNO ;
0349         ELSE
0350             SM_OBJ_TYPE(PNO) := AS_TYPE_SPEC(PPAI) ;
0351             SM_OBJ_DEF(PNO) := AS_OBJECT_DEF(PPAI) ;
0352             SM_FIRST(PNO) := PNO ;
0353         END IF ;
0354     END IF ;
0355 END VERIFICA_CONST_ID ;
0356 -----
0357 ---
0358 --- O NO "CONST_ID" PODE APARECER EM 2 CONTEXTOS : COMO FILHO DE UM NO
0359 --- "CONSTANT" OU DE UM NO "DEFERRED_CONSTANT"(NO CASO DE SER A
0360 --- DECLARACAO COMPLETA DE UMA CONSTANTE POSTERGADA)? A FUNCAO DECLARACAO

```

```

0361 -- COMPLETA VERIFICA SE ESSE 2. CASO OCORRE.
0362 -- A FUNCAO "PARTE_PRIVADA" VERIFICA SE O NO APARECE NA SUB-ARVORE DA
0363 -- PARTE PRIVADA DE UM PACOTE.
0364 -- A FUNCAO "PRIMEIRA_COMPLEMENTACAO" VERIFICA SE A DECLARACAO ANALISADA
0365 -- E A 1. DECLARACAO COMPLETA DE UMA CONSTANTE POSTERIOR.
0366 -- OS ATRIBUTOS SEMANTICOS SAO PREENCHIDOS MESMO QUE A ESPECIFICACAO
0367 -- DE TIPO OU EXPRESSAO INICIAL DA CONSTANTE ESTEJAM INVALIDOS.
0368 --
0369 -- CONSTANT => AS_IDLS : IDLS ,
0370 --             AS_TYPE_SPEC : TYPE_SPEC ,
0371 --             AS_OBJECT_DEF : OBJECT_DEF ;
0372 --
0373 PROCEDURE VERIFICA_CONSTANT(PNO : PT_TREE) IS
0374 PIDS,PID,PTP,POBJ : PT_TREE ;
0375 BEGIN
0376     PIDS := AS_IDLS(PNO) ;
0377     PTP := AS_TYPE_SPEC(PNO) ;
0378     POBJ := AS_OBJECT_DEF(PNO) ;
0379     VERIFICA_TYPE_SPEC(PTP) ;
0380     RESOLVE_NOME_E_EXPRESSAO(POBJ) ;
0381     VERIFICA_OBJECT_DEF(POBJ) ;
0382     PID := AS_LIST(PIDS) ;
0383     WHILE PID /= NIL
0384     LOOP
0385         VERIFICA_CONST_ID(PID , PNO) ;
0386         PID := NEXT(PID) ;
0387     END LOOP ;
0388     TORNA_IDENTIFICADOR_VISIVEL(PIDS) ;
0389 END VERIFICA_CONSTANT ;
0390 -----
0391 --
0392 -- COMO AS INFORMACOES DA ESPECIFICACAO DO TIPO DA CONSTANTE SAO USADAS
0393 -- PARA RESOLUCAO DE SUA EXPRESSAO INICIAL, NAO E NECESSARIO VERIFICAR
0394 -- A COMPATIBILIDADE DE TIPOS.
0395 --
0396 -- CONSTRAINED => AS_NAME : NAME ,
0397 --                AS_CONSTRAINT : CONSTRAINT ,
0398 --                SM_TYPE_STRUCT : TYPE_SPEC ,
0399 --                SM_BASE_TYPE : TYPE_SPEC ,
0400 --                SM_CONSTRAINT : CONSTRAINT ;
0401 --
0402 PROCEDURE VERIFICA_CONSTRAINT(PNO : PT_TREE) IS
0403 PN,PR,PX,POBJ : PT_TREE ;
0404 BEGIN
0405     PN := AS_NAME(PNO) ;
0406     PR := AS_CONSTRAINT(PNO) ;
0407     RESOLVE_NOME_E_EXPRESSAO(PN) ;
0408     VERIFICA_NAME(PN) ;
0409     POBJ := APANHA_OBJETO(PN) ;
0410     IF (APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID) OR ELSE
0411         (APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_TYPE_ID AND
0412         APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_SUBTYPE_ID) THEN
0413         MSG("ESPERADO IDENTIFICADOR DE TIPO",PN) ;
0414     ELSE
0415         VERIFICA_CONSTRAINT(PR) ;
0416         IF NOT TIPOS_COMPATIVEIS(PN,PR) THEN
0417             MSG("RESTRICAO NAO E COMPATIVEL COM TIPO",PR) ;
0418         ELSE
0419             PX := SM_DEFN(POBJ) ;
0420             IF APANHA_NOME_NO(PX) = DN_TYPE_ID THEN

```

```

0421     IF APANHA_NOME_NO(SM_TYPE_SPEC(PX)) /= DN_DERIVED THEN
0422         SM_BASE_TYPE(PNO) := SM_TYPE_SPEC(PX) ;
0423         SM_TYPE_STRUCT(PNO) := SM_TYPE_SPEC(PX) ;
0424         SM_CONSTRAINT(PNO) := PR ;
0425     ELSE
0426         SM_BASE_TYPE(PNO) := TIPO_BASE(SM_TYPE_SPEC(PX)) ;
0427         SM_TYPE_STRUCT(PNO) := ESTRUTURA(SM_TYPE_SPEC(PX)) ;
0428         IF APANHA_NOME_NO(PR) /= DN_VOID THEN
0429             SM_CONSTRAINT(PNO) := PR ;
0430         ELSE
0431             SM_CONSTRAINT(PNO) := ULTIMA_RESTRICAO(SM_TYPE_SPEC(PX)) ;
0432         END IF ;
0433     END IF ;
0434 ELSE -- DN_SUBTYPE_ID
0435     SM_BASE_TYPE(PNO) := SM_BASE_TYPE(SM_TYPE_SPEC(PX)) ;
0436     SM_TYPE_STRUCT(PNO) := SM_TYPE_STRUCT(SM_TYPE_SPEC(PX)) ;
0437     IF APANHA_NOME_NO(PR) /= DN_VOID THEN
0438         SM_CONSTRAINT(PNO) := PR ;
0439     ELSE
0440         SM_CONSTRAINT(PNO) := SM_CONSTRAINT(SM_TYPE_SPEC(PX)) ;
0441     END IF ;
0442 END IF ;
0443 END IF ;
0444 END IF ;
0445 END VERIFICA_CONSTRAINT ;
0446 -----
0447 --
0448 -- CONTEXT => AS-LIST : SEQ OF CONTEXT-ELEMENT ;
0449 --
0450 PROCEDURE VERIFICA_CONTEXT(PNO : PT-TREE) IS
0451 PELEM : PT-TREE ;
0452 BEGIN
0453     PELEM := AS-LIST(PNO) ;
0454     WHILE PELEM /= NIL
0455     LOOP
0456         CASE APANHA_NOME_NO(PELEM) IS
0457             WHEN DN_USE =>
0458                 VERIFICA_USE(PELEM) ;
0459             WHEN DN_WITH =>
0460                 VERIFICA_WITH(PELEM) ;
0461             WHEN DN_PRAGMA =>
0462                 VERIFICA_PRAGMA(PELEM) ;
0463         END CASE ;
0464         PELEM := NEXT(PELEM) ;
0465     END LOOP ;
0466 END VERIFICA_CONTEXT ;
0467 -----
0468 --
0469 -- DEF-CHAR => SM-OBJ-TYPE : TYPE-SPEC ,
0470 --             SM-POS : INTEGER ,
0471 --             SM-REP : INTEGER ;
0472 --
0473
0474 PROCEDURE VERIFICA_DEF_CHAR(PNO, PPAI:PT-TREE ; NUMR:INTEGER) IS
0475 PDEF : PT-TREE ;
0476 BEGIN
0477     IF JA_USADO(PNO) THEN
0478         PDEF := APANHA_DECLARACAO(PNO) ; -- DEF-ID
0479         IF SM-OBJ-TYPE(PDEF) = PPAI THEN
0480             MSG("CARACTER JA FOI USADO NESSE TIPO DE ENUMERACAO",PNO)

```

```

0481     ELSE
0482         SM-OBJ-TYPE(PNO) := PPAI ?
0483         SM-POS(PNO) := NUMB ?
0484         SM-REP(PNO) := NUMB ?
0485     END IF ?
0486     ELSE
0487         SM-OBJ-TYPE(PNO) := PPAI ?
0488         SM-POS(PNO) := NUMB ?
0489         SM-REP(PNO) := 0 ?
0490     END IF ?
0491 END VERIFICA-DEF-CHAR ?
0492 -----
0493 ---
0494     O PONTEIRO "PPAI" APONTA PARA O NO "ENUM-LITERAL-S" DO TIPO DE
0495     ENUMERACAO EM QUESTAO.
0496     UM CARACTER DE ENUMERACAO PODE SER DEFINIDO EM MAIS QUE UM
0497     TIPO DE ENUMERACAO (ARM 9C3.5.1-4) ? O UNICO TESTE NECESSARIO
0498     E VERIFICAR SE UM CARACTER QUE JA TENHA SIDO DECLARADO
0499     (COMO CARACTER DE ENUMERACAO) NAO O FOI NA MESMA DECLARACAO .
0500 ---
0501 --- DECL-S => AS-LIST : SEQ OF DECL ?
0502 ---
0503 PROCEDURE VERIFICA-DECL-S(PNO : PT-TREE) IS
0504 PDCL : PT-TREE?
0505 BEGIN
0506     PDCL := AS-LIST(PNO) ?
0507     WHILE PDCL /= NIL
0508     LOOP
0509         VERIFICA-DECL(PDCL) ?
0510         PDCL := NEXT(PDCL) ?
0511     END LOOP ?
0512 END VERIFICA-DECL-S ?
0513 -----
0514 ---
0515 --- DEF-OP : SM-SPEC : HEADER ,
0516 ---         SM-BODY : SUBP-BODY-DESC ,
0517 ---         SM-LOCATION : LOCATION ,
0518 ---         SM-STUB : DEF-OCCURRENCE ,
0519 ---         SM-FIRST : DEF-OCCURRENCE ?
0520 ---
0521 PROCEDURE VERIFICA-DEF-OP(PNO , PPAI : PT-TREE) IS
0522 PSP1 , PSP2 , PSTB : PT-TREE ?
0523 BEGIN
0524     SM-LOCATION(PNO) := APONTA-VOID ?
0525     IF APANHA-NOME-NO(PPAI) = DN-SUBPROGRAM-DECL THEN
0526         IF DECLARACAO-EQUIVALENTE(PNO) THEN
0527             MSG("ESPECIFICACAO DUPLICADA",PNO) ?
0528         ELSE
0529             SM-SPEC(PNO) := AS-HEADER(PPAI) ?
0530             SM-FIRST(PNO) := PNO ?
0531         END IF ?
0532     ELSE --- DN-SUBPROGRAM-BODY
0533         IF CORPO-EQUIVALENTE(PNO) THEN
0534             MSG("CORPO DUPLICADO",PNO) ?
0535         ELSE
0536             SM-SPEC(PNO) := AS-HEADER(PPAI) ?
0537             SM-BODY(PNO) := AS-BLOCK-STUB(PPAI) ?
0538             IF EXISTE-ESPECIFICACAO(PNO) THEN
0539                 PESF := APANHA-ESPECIFICACAO(PNO) ?
0540                 SM-FIRST(PNO) := PESF ?

```



```

0541         SM-BODY(PESP) := PNO ;
0542     ELSE
0543         SM-FIRST(PNO) := PNO ;
0544     END IF ;
0545     IF EXISTE-STUB(PNO) THEN
0546         SM-STUB(PNO) := APANHA-STUB(PNO) ;
0547     END IF ;
0548     END IF ;
0549 END VERIFICA-DEF-OP ;
0550 -----
0551 ---
0552 ---
0553 AS FUNCOES "DECLARACAO-EQUIVALENTE" E "CORPO-EQUIVALENTE"
0554 VERIFICAM SE EXISTE UMA DECLARACAO OU UM CORPO DE FUNCAO
0555 EQUIVALENTE (DESIGNADOR E PARAMETROS) NA MESMA PARTE
0556 DECLARATIVA .
0557 O ATRIBUTO "SM-LOCATION" E PREENCHIDO SE APARECER O PRAGMA
0558 "IN-LINE" OU UMA ESPECIFICACAO DE ENDEREÇO APLICADA A
0559 FUNCAO .
0560 ---
0561 --- DEFERRED-CONSTANT => AS-ID-S : ID-S ,
0562 --- AS-NAME : NAME ;
0563 ---
0564 PROCEDURE VERIFICA-DEFERRED-CONSTANT(PNO : PT-TREE) IS
0565 PIDS , PID , PTP : PT-TREE ;
0566 BEGIN
0567     PIDS := AS-ID-S(PNO) ;
0568     PTP := AS-TYPE-SPEC(PNO) ;
0569     IF NOT CONTEXTO-DEFERRED-CONSTANT(PNO) THEN
0570         MSG("CONTEXTO INVALIDO PARA DECLARACAO DE CONSTANTE",PNO) ;
0571     ELSE
0572         VERIFICA-TYPE-SPEC(PTP) ;
0573         PID := AS-LIST(PIDS) ;
0574         WHILE PID /= NIL
0575             LOOP
0576                 VERIFICA-CONST-ID(PID,PNO) ;
0577                 PID := NEXT(PID) ;
0578             END LOOP ;
0579             TORNA-IDENTIFICADOR-VISIVEL(PIDS) ;
0580     END IF ;
0581 END VERIFICA-DEFERRED-CONSTANT ;
0582 -----
0583 ---
0584 UMA DECLARACAO DE CONSTANTE POSTERGADA SO E PERMITIDA NA
0585 PARTE VISIVEL DE UM PACOTE (ARM SC. 7.4-3) ; ISSO E TESTADO
0586 PELA FUNCAO "CONTEXTO-DEFERRED-CONSTANT" .
0587 ---
0588 --- DELAY => AS-EXP : EXP ;
0589 ---
0590 PROCEDURE VERIFICA-DELAY(PNO : PT-TREE ) IS
0591 PEXP : PT-TREE ;
0592 BEGIN
0593     PEXP := AS-EXP(PNO) ;
0594     IF NOT CONTEXTO-TASK-BODY(PNO) THEN
0595         MSG("COMANDO DELAY DEVE APARECER NO CORPO DE UMA TASK",PNO)
0596     ELSE
0597         RESOLVE-NOME-E-EXPRESSAO(PEXP) ;
0598         VERIFICA-EXP(PEXP) ;
0599         IF NOT TIPO-DURATION(PEXP) THEN
0600             MSG("ESPERADA FUNCAO DO TIPO 'DURATION' ",PEXP) ;

```

```

0601     END IF ;
0602     END IF ;
0603     END VERIFICA-DELAY ;
0604     -----
0605     ---
0606     UM COMANDO DELAY SO PODE OCORRER DENTRO DO CORPO DE UMA TASK ;
0607     A FUNCAO "CONTEXTO-TASK-BODY" REALIZA ESSE TESTE ,
0608     A EXPRESSAO DO COMANDO DEVE TER O TIPO PREDEFINIDO
0609     "DURATION" (ARM SC. 9.6-3) ; A FUNCAO "TIPO-DURATION"
0610     REALIZA ESSE TESTE .
0611     ---
0612     --- DERIVED => AS-CONSTRAINED : CONSTRAINED ;
0613     --- SM-SIZE : EXP-VOID ;
0614     --- SM-ACTUAL-DELTA : RATIONAL ;
0615     --- SM-PACKING : BOOLEAN ;
0616     --- SM-CONTROLLED : BOOLEAN ;
0617     ---
0618     PROCEDURE VERIFICA-DERIVED(PNO : PT-TREE) IS
0619     PCTN : PT-TREE ;
0620     BEGIN
0621     PCTN := AS-CONSTRAINED(PNO) ;
0622     VERIFICA-CONSTRAINED(PCTN) ;
0623     END VERIFICA-DERIVED ;
0624     -----
0625     ---
0626     --- DISCRIM-AGGREGATE => AS-LIST : SEQ OF COMP-ASSOC ;
0627     --- SM-NORMALIZED-COMP-S : EXP-S ;
0628     ---
0629     FUNCTION ESTRUTURA-DISCRIM-AGGREGATE(PNO : PT-TREE)
0630     RETURN BOOLEAN IS
0631     PD ; PCH ; PC ; PCS : PT-TREE ;
0632     CHAVE-OTHERS : BOOLEAN ;
0633     BEGIN
0634     PD := AS-LIST(PNO) ;
0635     WHILE (APANHA-NOME-NO(PD) /= DN-NAMED) AND
0636     (PD /= NIL)
0637     LOOP -- PULA ASSOCIACAO POR POSICAO
0638     PD := NEXT(PD) ;
0639     END LOOP ;
0640     WHILE APANHA-NOME-NO(PD) = DN-NAMED
0641     LOOP
0642     PCS := AS-CHOICE-S(PD) ;
0643     PC := AS-CHOICE(PC) ;
0644     CHAVE-OTHERS := TRUE ;
0645     WHILE PC /= NIL
0646     LOOP
0647     IF APANHA-NOME-NO(PC) = DN-OTHERS THEN
0648     IF NOT CHAVE-OTHERS THEN
0649     MSG("OPCAO 'OTHERS' DEVE SER A PRIMEIRA",PC) ;
0650     RETURN FALSE ;
0651     ELSIF NEXT(PC) /= NIL THEN
0652     MSG("OPCAO 'OTHERS' DEVE APARECER SOZINHA",PC) ;
0653     RETURN FALSE ;
0654     ELSIF NEXT(PD) /= NIL THEN
0655     MSG("OPCAO 'OTHERS' DEVE SER A ULTIMA",PC) ;
0656     RETURN FALSE ;
0657     END IF ;
0658     END IF ;
0659     PC := NEXT(PC) ;
0660     CHAVE-OTHERS := FALSE ;

```



```

0721             END IF ;
0722             END IF ;
0723             END IF ;
0724             END IF ;
0725             FLO := NEXT(PLO) ;
0726             END LOOP ;      -- FLO /= NIL
0727             DISCRIMINANTE-COMPLETO(PDEF) ;
0728             NORMALIZA(PNO) ;
0729             END IF ;
0730             END IF ;
0731             END VERIFICA-DSCRNT-AGGREGATE ;
0732             -----
0733             ---
0734             A ESTRUTURA DO AGREGADO (ASSOCIACAO POR NOME X POR POSICAO , OPCAO
0735             'OTHERS' , ...) E TESTADA PELA FUNCAO "ESTRUTURA-DSCRNT-AGGREGATE".
0736             UMA RESTRICAO DE DISCRIMINANTE SO E PERMITIDA EM UMA INDICACAO
0737             DE SUBTIPO APOS UMA 'TYPE-MARK'. ESSA DEVE DENOTAR UM TIPO COM
0738             DISCRIMINANTES OU UM TIPO DE ACESSO QUE DESIGNE UM TIPO COM
0739             DISCRIMINANTES(ARM SC. 3.7.2-1). ESSAS CONDICAOES SAO TESTADAS NO
0740             1. 'IF' E PELA FUNCAO 'TIPO-COM-DISCRIMINANTES'.
0741             PARA CADA ASSOCIACAO , A EXPRESSAO E O DISCRIMINANTE DEVEM TER O
0742             MESMO TIPO(ARM SC. 3.7.2-4). AS FUNCOES 'APANHA-TIPO-EXP' E
0743             'APANHA-TIPO-DISC' REALIZAM ESSE TESTE.
0744             PARA UMA ASSOCIACAO NOMINAL , O NOME DO DISCRIMINANTE DEVE SER
0745             VALIDO PARA O TIPO EM QUESTAO(ARM SC. 3.7.2-4). ISSO E TESTADO
0746             PELA FUNCAO 'DISCRIMINANTE-VALIDO'.
0747             A FUNCAO 'DISCRIMINANTE-LIVRE' , USADA NO PROCESSAMENTO DA OPCAO
0748             'OTHERS' , DEVOLVE UM PONTEIRO PARA UM DISCRIMINANTE AINDA NAO
0749             ESPECIFICADO NO AGREGADO.
0750             UMA RESTRICAO DE DISCRIMINANTES DEVE PROVER EXATAMENTE UM VALOR
0751             PARA CADA DISCRIMINANTE DO TIPO(ARM SC. 3.7.2-4). A ROTINA
0752             'MARCA-DISCRIMINANTE' E A FUNCAO 'DISCRIMINANTE-COMPLETO'
0753             REALIZAM ESSE TESTE .
0754             A ROTINA 'NORMALIZA' CONSTRUI A LISTA NORMALIZADA DO NO
0755             (SM-NORMALIZED-COMP-S).
0756             A ROTINA 'DESMARCA-DISCRIMINANTES' PERCORRE A LISTA DE
0757             DISCRIMINANTES DO TIPO DESLIGANDO O ATRIBUTO
0758             'SM-DISCRIMINANTE-ESPECIFICADO'.
0759             ---
0760             ---
0761             --- DSCRNT-ID => SM-OBJ-TYPE : TYPE-SPEC ,
0762             ---             SM-INIT-EXP : EXP-VOID ,
0763             ---             SM-FIRST : DEF-OCURRENCE ,
0764             ---             SM-COMP-SPEC : COMP-REP-VOID ,
0765             ---             SE-DISCRIMINANTE-ESPECIFICADO : BOOLEAN ;
0766             ---
0767             --- DSCRNT-VAR => AS-ID-S : ID-S ,
0768             ---             AS-NAME : NAME ,
0769             ---             AS-OBJECT-DEF : OBJECT-DEF ;
0770             ---
0771             --- DSCRNT-VAR-S => AS-LIST : SEQ OF DSCRNT-VAR ;
0772             ---
0773             PROCEDURE VERIFICA-DSCRNT-VAR-S(PNO : PT-TREE) IS
0774             PDV : PT-TREE ; EXP : BOOLEAN ;
0775             BEGIN
0776             IF NOT CONTEXTO-DISCRIMINANTE(PNO) THEN
0777             MSG("DISCRIMINANTE INVALIDO PARA TIPO",PNO) ;
0778             ELSE
0779             PDV := AS-LIST(PNO) ;
0780             IF APANHA-NOME-NO(AS-OBJECT-DEF(PDV)) = DN-VOID THEN

```

```

0781     EXP := FALSE ;    ---   LISTA SEM VALORES INICIAIS
0782     ELSE
0783     EXP := TRUE ;     ---   LISTA COM VALORES INICIAIS
0784     END IF ;
0785     WHILE PDV /= NIL
0786     LOOP
0787     VERIFICA-DSCRMT-VAR(PDV,EXP,NO) ;
0788     PDV := NEXT(PDV) ;
0789     END LOOP ;
0790     END IF ;
0791     END VERIFICA-DSCRMT-VAR-S ;
0792     -----
0793     PROCEDURE VERIFICA-DSCRMT-VAR(PNO , PDVS : PT-TREE ;
0794     EXP : BOOLEAN ) IS
0795     PIDS , FN , PEXP , PID , PF : PT-TREE ;
0796     BEGIN
0797     PIDS := AS-ID-S(PNO) ;
0798     FN := AS-NAME(PNO) ;
0799     PEXP := AS-OBJECT-DEF(PNO) ;
0800     PID := AS-LIST(PIDS) ;
0801     RESOLVE-NOME-E-EXPRESSAO(PN) ;
0802     VERIFICA-NAME(PN) ;
0803     POBJ := APANHA-OBJETO(PN) ;
0804     IF NOT DECLARADO(POBJ) THEN    ---   USED-NAME-ID
0805     MSG("IDENTIFICADOR NAO DECLARADO",FN) ;
0806     ELSE
0807     PDEF := APANHA-DECLARACAO(POBJ) ;
0808     IF APANHA-NOME-NO(PDEF) /= DN-TYPE-ID THEN
0809     MSG("ESPERADO IDENTIFICADOR DE TIPO",FN) ;
0810     ELSIF NOT TIPO-DISCRETO(PDEF) THEN
0811     MSG("TIPO DE VIA SER DISCRETO",FN) ;
0812     END IF ;
0813     IF EXP THEN
0814     IF APANHA-NOME-NO(PEXP) = DN-VOID THEN
0815     MSG("ESPERADA INICIALIZACAO",FN) ;
0816     ELSE
0817     RESOLVE-NOME-E-EXPRESSAO(PEXP) ;
0818     VERIFICA-EXP(PEXP) ;
0819     IF NOT TIPOS-COMPATIVELIS(PEXP,PN) THEN
0820     MSG("TIPO DA EXPRESSAO INCOMPATIVEL",PEXP) ;
0821     END IF ;
0822     END IF ;
0823     ELSIF APANHA-NOME-NO(PEXP) /= DN-VOID THEN
0824     MSG("NAO ERA ESPERADA INICIALIZACAO",PEXP) ;
0825     END IF ;
0826     WHILE PID /= NIL
0827     LOOP
0828     IF NOVO-DISCRIMINANTE(PID , PDVS) THEN
0829     MSG("IDENTIFICADOR JA USADO COMO DISCRIMINANTE",PID) ;
0830     ELSE
0831     SM-OBJ-TYPE(PID) := FN ;
0832     SM-INIT-VALUE(PID) := PEXP ;
0833     SM-COMP-SPEC(PID) := APONTA-VOID ;
0834     IF NOT PRIMEIRA-ESPECIFICACAO(PID) THEN
0835     SM-FIRST(PID) := APANHA-FIRST(PID) ;
0836     ELSE
0837     SM-FIRST(PID) := PID ;
0838     END IF ;
0839     END IF ;
0840     END LOOP ;

```

```

0841     END IF ;
0842 END VERIFICA-DSCRNT-VAR ;
0843 -----
0844 ---
0845     UMA PARTE DISCRIMINANTE SO E VALIDA NA DECLARACAO DE UM RECORD ,
0846     DE UM TIPO PRIVADO OU INCOMPLETO OU NA DECLARACAO DE UM
0847     PARAMETRO GENERICO EM UMA DECLARACAO DE TIPO FORMAL (ARM SC. 3.7.1-1)
0848     ; ESSA CONDICAO E TESTADA PELA FUNCAO 'CONTEXTO-DISCRIMINANTE'.
0849     EXPRESSOES INICIAIS DEVEM EXISTIR OU PARA TODOS OS DISCRIMINANTES
0850     OU PARA NENHUM DISCRIMINANTE ; O 2. PARAMETRO DA ROTINA
0851     'VERIFICA-DSCRNT-VAR' GUARDA ESSA INFORMACAO.
0852     O TIPO DE UM DISCRIMINANTE DEVE SER DISCRETO (ARM SC. 3.7.1-1) ; ESSE
0853     FATO E TESTADO PELA ROTINA 'TIPO-DISCRETO'.
0854     O TIPO DA EXPRESSAO DE INICIALIZACAO DEVE SER IGUAL AO TIPO DO
0855     DISCRIMINANTE (ARM SC. 3.7.1-4) ; ISSO E TESTADO PELA FUNCAO
0856     'TIPOS-INCOMPATIVEIS'.
0857     O 3. PARAMETRO APONTA PARA O NO 'DSCRNT-VAR-S' , RAIZ DA
0858     SUB-ARVORE QUE ESTA SENDO ANALISADA.
0859 ---
0860 --- ENTRY => AS-DSCRNT-RANGE-VOID ; DSCRNT-RANGE-VOID ,
0861 ---     AS-PARAM-S ; PARAM-S ;
0862 ---
0863 --- ENTRY-ID => SM-SPEC ; HEADER ,
0864 ---     SM-ADDRESS ; PARAM-S ;
0865 ---
0866 PROCEDURE VERIFICA-ENTRY (PNO ; PT-TREE) IS
0867 PDR , PPS ; PT-TREE ;
0868 BEGIN
0869     PDR := AS-DSCRNT-RANGE-VOID (PNO) ;
0870     PPS := AS-PARAM-S (PNO) ;
0871     IF NOT CONTEXTO-TASK-SPECIFICATION (PNO) THEN
0872         MSG (" 'ENTRY' SO E PERMITIDA EM ESPECIFICACAO DE TASK" , PNO) ;
0873     ELSE
0874         VERIFICA-DSCRNT-RANGE-VOID (PDR) ;
0875         VERIFICA-PARAM-S (PPS) ;
0876     END IF ;
0877 END VERIFICA-ENTRY ;
0878 -----
0879 PROCEDURE VERIFICA-ENTRY-ID (PNO , PPAI ; PT-TREE) IS
0880 BEGIN
0881     IF DECLARADO (PNO) THEN
0882         IF DECLARACAO-EQUIVALENTE (PNO , PPAI) THEN
0883             MSG (" 'OVERLOADING' INVALIDO" , PNO) ;
0884         ELSE
0885             SM-SPEC (PNO) := AS-HEADER (PPAI) ;
0886             SM-ADDRESS (PNO) := APONTA-VOID ;
0887         END IF ;
0888     ELSE
0889         SM-SPEC (PNO) := AS-HEADER (PPAI) ;
0890         SM-ADDRESS (PNO) := APONTA-VOID ;
0891     END IF ;
0892 END VERIFICA-ENTRY-ID ;
0893 -----
0894 ---
0895     UMA DECLARACAO DE 'ENTRY' SO E PERMITIDA EM UMA ESPECIFICACAO
0896     DE TASK (ARM SC. 9.5-1) ; ISSO E TESTADO PELA FUNCAO
0897     'CONTEXTO-TASK-SPECIFICATION'.
0898     DUAS DECLARACOES QUE OCORRAM NA MESMA PARTE DECLARATIVA NAO
0899     DEVEM SER HOMOGRAFAS , A MENOS DE ALGUMAS REGRAS DEFINIDAS NA
0900     SECAO 9.3-17. A FUNCAO 'DECLARACOES-EQUIVALENTE' VERIFICA SE

```

```

0901     EXISTE OUTRA DECLARACAO NA MESMA PARTE DECLARATIVA COM ESSA
0902     CARACTERISTICA.
0903     O ATRIBUTO 'SM-ADDRESS' SO SERA PREENCHIDO SE HOUVER UM FRAGMENTO
0904     'ADDRESS' PARA A 'ENTRY' EM QUESTAO.
0905
0906 ---
0907 --- ENTRY_CALL => AS_NAME : NAME ;
0908 ---                   AS_PARAM_ASSOC_S : PARAM_ASSOC_S ;
0909 ---                   SM_NORMALIZED_PARAM_S : EXPL_S ;
0910 ---
0911 ---
0912 PROCEDURE VERIFICA_ENTRY_CALL(PNO : PT_TREE) IS
0913 FN , PPAS , POBJ : PT_TREE ;
0914 BEGIN
0915     FN := AS_NAME(PNO) ;
0916     PPAS := AS_PARAM_ASSOC_S(PNO) ;
0917     IF NOT CONTEXTO_TASK_BODY(PNO) THEN
0918         MSG("'ENTRY_CALL' SO EM 'TASK_BODY'",PNO) ;
0919     ELSE
0920         RESOLVE_NOME_E_EXPRESSAO(FN) ;
0921         VERIFICA_NAME(FN) ;
0922         POBJ := APANHA_OBJETO(FN) ;
0923         IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
0924             APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_ENTRY_ID THEN
0925             MSG("ESPERADO IDENTIFICADOR DE 'ENTRY'",POBJ) ;
0926         ELSIF VERIFICA_PARAM_ASSOC_S(PNO) THEN
0927             NORMALIZA(PNO) ;
0928         END IF ;
0929         VERIFICA_INDICES(PNO , POBJ) ;
0930     END IF ;
0931 END VERIFICA_ENTRY_CALL ;
0932
0933 ---
0934 --- SE A 'ENTRY' CORRESPONDENTE NAO REPRESENTAR UMA FAMILIA , O
0935 --- 'NAME' NAO DEVE REPRESENTAR UM COMPONENTE INDEXADO ; CASO
0936 --- CONTRARIO, O COMPONENTE INDEXADO DEVE POSSUIR UM UNICO INDICE,
0937 --- CUJA TIPO ESTEJA DE ACORDO COM A DEFINICAO DA DECLARACAO DA
0938 --- 'ENTRY'; ESSE FATO E TESTADO PELA ROTINA 'VERIFICA_INDICES'.
0939 ---
0940 --- ENUM_ID => SM_OBJ_TYPE : TYPE_SPEC ;
0941 ---                   SM_POS : INTEGER ;
0942 ---                   SM_REP : INTEGER ;
0943 ---
0944 --- ENUM_LITERAL_S => AS_LIST : SEQ OF ENUM_LITERAL ;
0945 ---                   SM_SIZE : EXP_VOID ;
0946 ---
0947 PROCEDURE VERIFICA_ENUM_LITERAL_S(PNO : PT_TREE) IS
0948 PID : PT_TREE ;
0949 NUMB : INTEGER := 0 ;
0950 BEGIN
0951     PID := AS_LIST(PNO) ;
0952     WHILE PID /= NIL
0953     LOOP
0954         VERIFICA_ENUM_LITERAL(PID , PNO , NUMB) ;
0955         NUMB := NUMB + 1 ;
0956         PID := NEXT(PID) ;
0957     END LOOP ;
0958 END VERIFICA_ENUM_LITERAL_S ;
0959
0960 PROCEDURE VERIFICA_ENUM_ID(PNO,PPAI : PT_TREE ; NUMB : INTEGER) IS

```

```

0961 FLISTA : PT_TREE ?
0962 BEGIN
0963   IF DECLARADO_NO_CONTEXTO(PNO) THEN
0964     FLISTA := APANHA_LISTA_DEFINICAO(PNO) ?
0965     IF NOT SOBRE_POSICAO_POSSIVEL(PNO,PPAI,FLISTA) THEN
0966       MSG("SOBREPOSICAO INVALIDA",PNO)?
0967     ELSE
0968       SM_OBJ_TYPE(PNO) := PPAI ?
0969       SM_POS(PNO) := NUMB ?
0970       SM_REF(PNO) := 0 ?
0971     END IF ?
0972   ELSE
0973     SM_OBJ_TYPE(PNO) := PPAI ?
0974     SM_POS(PNO) := NUMB ?
0975     SM_REF(PNO) := 0 ?
0976   END IF ?
0977 END VERIFICA_ENUM_ID ?
0978 -----
0979 ---
0980 --- O PONTEIRO 'PPAI' APONTA PARA O NO 'ENUM_LITERAL_S' DO TIPO
0981 --- DE ENUMERACAO EM QUESTAO.
0982 --- OS IDENTIFICADORES E CARACTERES EMPREGADOS EM UM TIPO DE
0983 --- ENUMERACAO DEVEM SER DISTINTOS(ARM-SC.3.5.1-3) DESSA FORMA,
0984 --- NO CASO DO IDENTIFICADOR JA TER SIDO DECLARADO, E TESTADO SE
0985 --- SE O TIPO DO MESMO E IGUAL AO TIPO DE ENUMERACAO.
0986 --- A FUNCAO 'SOBREPOSICAO_POSSIVEL' VERIFICA SE, NO CASO DO MESMO
0987 --- IDENTIFICADOR JA TER SIDO DECLARADO, A NOVA DECLARACAO E
0988 --- COMPATIVEL COM AS REGRAS DE VISIBILIDADE E SOBREPOSICAO.
0989 ---
0990 --- EXCEPTION => AS_IDS : IDS ,
0991 ---           AS_EXCEPTION_DEF : EXCEPTION_DEF ?
0992 ---
0993 --- EXCEPTION_ID => SM_EXCEPTION_DEF : EXCEPTION_DEF ?
0994 ---
0995 PROCEDURE VERIFICA_EXCEPTION(PNO : PT_TREE) IS
0996 PIDS , PID , PED : PT_TREE ?
0997 BEGIN
0998   PIDS := AS_IDS(PNO) ?
0999   PED := AS_EXCEPTION_DEF(PNO) ?
1000   VERIFICA_EXCEPTION_DEF(PDEF) ?
1001   PID := AS_LIST(PIDS) ?
1002   WHILE PID /= NIL
1003   LOOP
1004     VERIFICA_EXCEPTION_ID(PID,PNO) ?
1005     PID := NEXT(PID) ?
1006   END LOOP ?
1007 END VERIFICA_EXCEPTION ?
1008 -----
1009 PROCEDURE VERIFICA_EXCEPTION_ID(PNO , PPAI :PT_TREE) IS
1010 BEGIN
1011   IF DECLARADO_NO_CONTEXTO(PNO) THEN
1012     MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PID)?
1013   ELSE
1014     SM_EXCEPTION_DEF(PNO) := AS_EXCEPTION_DEF(PPAI) ?
1015   END IF ?
1016 END VERIFICA_EXCEPTION_ID ?
1017 ---
1018 --- EXIT => AS_NAME_VOID : NAME_VOID ,
1019 ---           AS_EXP_VOID : EXP_VOID ,
1020 ---           SM_STM : LOOP ?

```



```

1021  ---
1022  PROCEDURE VERIFICA_EXIT(PNO : PT_TREE) IS
1023  FN , PEXP, POBJ , PLF : PT_TREE ?
1024  BEGIN
1025      PN := AS_NAME_VOID(PNO) ?
1026      PEXP := AS_EXP_VOID(PNO) ?
1027      IF NOT CONTEXTO_LOOP(PNO) THEN
1028          MSG("CONTEXTO INVALIDO PARA COMANDO 'EXIT'",PNO) ?
1029      ELSE
1030          IF APANHA_NOME_NO(PN) /= DN_VOID THEN
1031              RESOLVE_NOME_E_EXPRESSAO(PN) ?
1032              VERIFICA_NAME(PN) ?
1033              POBJ := APANHA_OBJETO(PN) ?
1034              IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
1035                  APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_LOOP_ID THEN
1036                  MSG("ESPERADO IDENTIFICADOR DE 'LOOP'",POBJ) ?
1037              ELSIF NOT LOOP_ABERTO(POBJ) THEN
1038                  MSG("LOOP JA FECHADO",PN) ?
1039              END IF ?
1040          END IF ?
1041          IF APANHA_NOME_NO(PEXP) /= DN_VOID THEN
1042              RESOLVE_NOME_E_EXPRESSAO(PEXP) ?
1043              VERIFICA_EXP(PEXP) ?
1044              IF NOT TIPO_BOOLEANO(PEXP) THEN
1045                  MSG("ESPERADA EXPRESSAO DO TIPO BOOLEANO"<PEXP) ?
1046              END IF ?
1047          END IF ?
1048      END IF ?
1049      PLF := APONTA_LOOP(PNO) ?
1050      SM_STM(PNO) := PLF ?
1051  END VERIFICA_EXIT ?
1052  -----
1053  ---
1054  --- UM COMANDO 'EXIT' DEVE OCORRER DENTRO DE UM COMANDO 'LOOP' ?
1055  --- ALEM DISSO, ELE NAO DEVE APARECER EM UM CORPO DE SUBPROGRAMA
1056  --- PACOTE, TASK, GENERIC OU COMANDO 'ACCEPT' (ARM-SC.5.7-3) NESSE
1057  --- E TESTADO PELA FUNCAO 'CONTEXTO_LOOP'.
1058  --- UM COMANDO 'EXIT' QUE CONTENHA UM NOME DEVE OCORRER EM UM LOOP
1059  --- COM NOME E SE REFERE A ESSE NOME (ARM-SC.5.7-3) ? DESSA FORMA, SE
1060  --- O ATRIBUTO 'AS_NAME_VOID' NAO APONTAR PARA UM NO DO TIPO 'VOID'
1061  --- , VERIFICA-SE SE O NOME EM QUESTAO REPRESENTA O NOME DE UM
1062  --- LOOP NAO FECHADO.
1063  --- A FUNCAO 'APONTA_LOOP' FAZ USO DAS INFORMACOES COLETADAS
1064  --- DURANTE O PROCESSAMENTO DO NO 'LOOP'.
1065  ---
1066  --- EXP_S => AS_LIST : SEQ OF EXP ?
1067  ---
1068  PROCEDURE VERIFICA_EXP_S(PNO : PT_TREE) IS
1069  PE : PT_TREE ?
1070  BEGIN
1071      PE := AS_LIST(PNO) ?
1072      WHILE PE /= NIL
1073      LOOP
1074          VERIFICA_EXP(PE) ?
1075          PE := NEXT(PE) ?
1076      END LOOP ?
1077  END VERIFICA_EXP_S ?
1078  -----
1079  ---
1080  --- FIXED => AS_EXP : EXP ,

```

```

1081 ---      AS_RANGE_VOID : RANGE_VOID ,
1082 ---      SM_SIZE : EXP_VOID ,
1083 ---      SM_ACTUAL_DELTA : RATIONAL ,
1084 ---      SM_BASE_TYPE : TYPE_STRUCT ;
1085 ---
1086 PROCEDURE VERIFICA_FIXED(PNO : PT_TREE) IS
1087 PE , PRV , PTP : PT_TREE ;
1088 BEGIN
1089     PE := AS_EXP(PNO) ;
1090     PRV := AS_RANGE_VOID(PNO) ;
1091     RESOLVE_NOME_E_EXPRESSAO(PE) ;
1092     VERIFICA_EXP(PE) ;
1093     IF NOT EXPRESSAO_ESTATICA(PE) THEN
1094         MSG("ESPERADA EXPRESSAO ESTATICA",PE) ;
1095     ELSIF NOT EXPRESSAO_POSITIVA(PE) THEN
1096         MSG("ESPERADA EXPRESSAO POSITIVA",PE) ;
1097     ELSE
1098         SM_ACTUAL_DELTA(PNO) := VALOR_ESTATICO(PE) ;
1099     END ;
1100     IF APANHA_NOME_NO(PRV) /= DN_VOID THEN
1101         VERIFICA_RANGE(PRV) ;
1102         IF NOT RANGE_REAL(PRV) THEN
1103             MSG("ESPERADO TIPO REAL PARA RANGE",PRV) ;
1104         ELSIF NOT RANGE_ESTATICO(PRV) THEN
1105             MSG("ESPERADO RANGE ESTATICO",PRV) ;
1106         END IF ;
1107     END IF ;
1108     PTP := APANHA_TIPO_PREDEFINIDO(PNO) ;
1109     IF PTP = NIL THEN
1110         MSG("NAO HA TIPO PREDEFINIDO COMPATIVEL COM DECLARACAO",PRV) ;
1111     ELSE
1112         SM_BASE_TYPE(PNO) := PTP ;
1113     END IF ;
1114     SM_SIZE(PNO) := APONTA_VOID ;
1115 END VERIFICA_FIXED ;
1116 -----
1117 ---
1118 --- O DELTA DA DECLARACAO E ESPECIFICADO PRO UMA EXPRESSAO ESTATICA
1119 --- DO TIPO REAL POSITIVA; ESSAS CONDICIONES SAO TESTADAS PELO 1.
1120 --- COMANDO 'IF' DA ROTINA.
1121 --- OS LIMITES DO RANGE DEVEM SER ESTATICOS E TER UM TIPO REAL,
1122 --- EMBORA OS 2 LIMITES NAO NECESSITEM TER O MESMO TIPO
1123 --- (ARM-SC.3.5.9-3); ESSAS CONDICIONES SAO TESTADAS PELO 2. COMANDO
1124 --- 'IF' .
1125 --- A FUNCAO 'APANHA_TIPO_PREDEFINIDO' VERIFICA SE EXISTE UM TIPO
1126 --- PREDEFINIDO COMPATIVEL COM A DECLARACAO, DEVOLVENDO UM PONTEIRO
1127 --- PARA A DEFINICAO DO MESMO OU O VALOR 'NIL'.
1128 ---
1129 --- FLOAT => AS_EXP : EXP ,
1130 ---      AS_RANGE_VOID : RANGE_VOID ,
1131 ---      SM_SIZE : EXP_VOID ,
1132 ---      SM_TYPE_STRUCT : TYPE_SPEC ;
1133 ---
1134 PROCEDURE VERIFICA_FLOAT(PNO : PT_TREE) IS
1135 PE , PRV : PT_TREE ;
1136 BEGIN
1137     PE := AS_EXP(PNO) ;
1138     PRV := AS_RANGE_VOID(PNO) ;
1139     RESOLVE_NOME_E_EXPRESSAO(PNO) ;
1140     VERIFICA_EXP(PNO) ;

```

```

1141 IF NOT EXPRESSAO_ESTATICA(PE) THEN
1142     MSG("ESPERADA EXPRESSAO ESTATICA">PE) #
1143 ELSEIF NOT EXPRESSAO_INTEIRA(PE) THEN
1144     MSG("ESPERADA EXPRESSAO DO TIPO INTEIRO">PE) #
1145 ELSEIF NOT EXPRESSAO_POSITIVA(PE) THEN
1146     MSG("ESPERADA EXPRESSAO POSITIVA">PE) #
1147 END IF #
1148 IF APANHA_NOME_NO(PRV) /= DN_VOID THEN
1149     VERIFICA_RANGE(PRV) #
1150     IF NOT RANGE_REAL(PRV) THEN
1151         MSG("ESPERADO RANGE DE TIPO REAL">PRV) #
1152     ELSEIF NOT RANGE_ESTATICO(PRV) THEN
1153         MSG("ESPERADO RANGE ESTATICO">PRV) #
1154     END IF #
1155 END IF #
1156 END VERIFICA_FLOAT #
1157 -----
1158 ---
1159 --- AS OBSERVACOES DA ULTIMA ROTINA TAMBEM SAO APLICADAS NESTA .
1160 ---
1161 --- FOR => AS_ID : ID , --- ITERATION_ID
1162 ---     AS_DSCRT_RANGE : DSCRT_RANGE #
1163 ---
1164 --- REVERSE => AS_ID : ID , --- ITERATION_ID
1165 ---     AS_DSCRT_RANGE : DSCRT_RANGE #
1166 --- WHILE => AS_EXP : EXP #
1167 ---
1168 --- LOOP => AS_ITERATION : ITERATION ,
1169 ---     AS_STM_S : STM_S #
1170 ---
1171 PROCEDURE VERIFICA_LOOP(PNO : PT_TREE) IS
1172 PIT , PSTMS , PSTM : PT_TREE #
1173 BEGIN
1174     MARCA_LOOP(PNO) #
1175     PIT := AS_ITERATION(PNO) #
1176     PSTMS := AS_STM_S(PNO) #
1177     CASE APANHA_NOME_NO(PID) IS
1178     WHEN DN_FOR =>
1179         VERIFICA_FOR(PID) #
1180     WHEN DN_REVERSE =>
1181         VERIFICA_REVERSE(PID) #
1182     WHEN DN_WHILE =>
1183         VERIFICA_WHILE(PID) #
1184     WHEN OTHERS =>
1185         NULL #
1186     END CASE #
1187     PSTM := AS_LIST(PSTMS) #
1188     WHILE PSTM /= NIL
1189     LOOP
1190         VERIFICA_STM(PSTM) #
1191         PSTM := NEXT(PSTM) #
1192     END LOOP #
1193     DESMARCA_LOOP(PNO) #
1194 END VERIFICA_LOOP #
1195 -----
1196 PROCEDURE VERIFICA_FOR(PNO : PT_TREE) IS
1197 FOR , PTE , PID : PT_TREE #
1198 BEGIN
1199     FOR := AS_DSCRT_RANGE(PNO) #
1200     PID := AS_ID(PNO) #

```

```

1201     VERIFICA_DSORT_RANGE(PDR) ?
1202     PTB := APANHA_TIPO_BASE(PDR) ?
1203     SM_BASE_TYPE(PID) := PTB ?
1204 END VERIFICA_FOR ?
1205 -----
1206 PROCEDURE VERIFICA_REVERSE(PNO : PT_TREE) IS
1207 FOR , PTB , PID : PT_TREE ?
1208 BEGIN
1209     PDR := AS_DSORT_RANGE(PNO) ?
1210     PID := AS_ID(PNO) ?
1211     VERIFICA_DSORT_RANGE(PDR) ?
1212     PTB := APANHA_TIPO_BASE(PDR) ?
1213     SM_BASE_TYPE(PID) := PTB ?
1214 END VERIFICA_REVERSE ?
1215 -----
1216 PROCEDURE VERIFICA_WHILE(PNO : PT_TREE) IS
1217 PE : PT_TREE ?
1218 BEGIN
1219     PE := AS_EXP(PNO) ?
1220     RESOLVE_NOME_E_EXPRESSAO(PE) ?
1221     VERIFICA_EXP(PE) ?
1222     IF NOT TIPO_BOOLEANO(PE) THEN
1223         MSG("ESPERADA EXPRESSAO DE TIPO BOOLEANO",PE) ?
1224     END IF ?
1225 END VERIFICA_WHILE ?
1226 -----
1227 ---
1228 --- NOTE QUE SE O DISCRETE_RANGE DAS SUBARVORES DO 'FOR' OU
1229 --- 'REVERSE' FOR INVALIDO A FUNCAO'APANHA_TIPO_BASE' DEVOLVERA
1230 --- UM PONTEIRO PARA 'TIPO_GERAL'.
1231 ---
1232 ---
1233 ---
1234 ---
1235 --- FUNCTION => AS_PARAMS : PARAMS ,
1236 ---         AS_NAME_VOID : NAME_VOID ?
1237 ---
1238 PROCEDURE VERIFICA_FUNCTION(PNO , PPAI : PT_TREE) IS
1239 BEGIN
1240     PPS := AS_PARAMS(PNO) ?
1241     PNV := AS_NAME_VOID(PNO) ?
1242     CASE APANHA_NOME_NO(PPAI) IS
1243     WHEN DN_SUBPROGRAM_DECL =>
1244         IF DECLARACAO_EQUIVALENTE(PNO,PPAI) THEN
1245             MSG("ESPECIFICACAO DUPLICADA",PNO) ?
1246         END IF ?
1247     WHEN DN_SUBPROGRAM_BODY =>
1248         IF CORPO_EQUIVALENTE(PNO,PPAI) THEN
1249             MSG("CORPO DUPLICADO",PNO) ?
1250         END IF ?
1251     WHEN OTHERS =>
1252         NULL ?
1253     END CASE ?
1254     VERIFICA_PARAMS(PPS) ?
1255     IF APANHA_NOME_NO(PNV) /= DN_VOID THEN
1256         RESOLVE_NOME_E_EXPRESSAO(PNV) ?
1257         VERIFICA_NAME(PNV) ?
1258         POBJ := APANHA_OBJETO(PNV) ?
1259         IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
1260             APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_TYPE_ID THEN

```

```

1261         MSG("ESPERADO IDENTIFICADOR DE TIPO",PN) ;
1262     END IF ;
1263 END IF;
1264 END VERIFICA_FUNCTION ;
1265 -----
1266 ---
1267 --- VER OBSERVACOS SOBRE AS FUNCOES 'DECLARACOES EQUIVALENTES' E
1268 --- 'CORPO EQUIVALENTE' NA ROTINA 'VERIFICA_DEF_OP' .
1269 ---
1270 --- FUNCTION_CALL => AS_NAME : NAME ;
1271 ---                   AS_PARAM_ASSOC_S : PARAM_ASSOC_S ;
1272 ---                   SM_EXP_TYPE : TYPE_SPEC ;
1273 ---                   SM_VALUE : VALUE ;
1274 ---                   SM_NORMALIZED_PARAM_S : EXP_S ;
1275 ---
1276 PROCEDURE VERIFICA_FUNCTION_CALL (PNO:PT_TREE;ESTAT:BOOLEAN:=FALSE) IS
1277 PN , PPAS : PT_TREE ;
1278 BEGIN
1279     PN := AS_NAME(PNO) ;
1280     PPAS := AS_PARAM_ASSOC_S(PNO) ;
1281     IF APANHA_TIPO_NO(PNO) /= TIPO_GERAL THEN
1282         IF ESTAT THEN
1283             IF EXPRESSAO_ESTATICA(PNO) THEN
1284                 MARCA_VALOR_ESTATICO(PNO) ;
1285             ELSE
1286                 MSG("ESPERADA EXPRESSAO ESTATICA",PNO) ;
1287             END IF;
1288         END IF ;
1289         VERIFICA_NAME(PN) ;
1290         IF VERIFICA_PARAM_ASSOC_S(PPAS,PNO) THEN
1291             MONTA_LISTA_NORMALIZADA(PNO) ;
1292         END IF ;
1293     END IF ;
1294 END VERIFICA_FUNCTION_CALL ;
1295 -----
1296 ---
1297 --- NOTE QUE QUANDO ESSA ROTINA E CHAMADA, A SUB_ARVORE
1298 --- CORRESPONDENTE JA SOFREU O PROCESSAMENTO DA ROTINA
1299 --- 'RESOLVE_NOME_E_EXPRESSAO'. DESSA FORMA, SE A CHAMADA DA FUNCAO
1300 --- ESTIVER SEMANTICAMENTE VALIDA, UMA BOA PARTE DOS TESTES
1301 --- NECESSARIOS JA TERA SIDO REALIZADA.
1302 --- O 2. PARAMETRO ('ESTAT') TEM O VALOR VERDADEIRO SE A EXPRESSAO
1303 --- TIVER QUE SER ESTATICA.
1304 ---
1305 --- FUNCTION_ID => SM_SPEC => HEADER ;
1306 ---                   SM_BODY : SUBP_BODY_DESC ;
1307 ---                   SM_LOCATION : LOCATION ;
1308 ---                   SM_STUB : DEF_OCCURRENCE ;
1309 ---                   SM_FIRST : DEF_OCCURRENCE ;
1310 ---
1311 PROCEDURE VERIFICA_FUNCTION_ID(PNO,PPAI : PT_TREE) IS
1312 PSP1 , PSP2 , PSTB , PH : PT_TREE ;
1313 BEGIN
1314     IF APANHA_NOME_NO(PPAI) = DN_SUBPROGRAM_DECL THEN
1315         IF DECLARACAO_EQUIVALENTE(PNO) THEN
1316             MSG("ESPECIFICACAO DUPLICADA",PNO) ;
1317         ELSE
1318             IF AS_SUBPROGRAM_DEF(PPAI) = DN_INSTANTIATION THEN
1319                 PH := CONSTROI_HEADER(PPAI) ;
1320                 SM_SPEC(PNO) := PH ;

```

```

1321         ELSE
1322             SM_SPEC(PNO) := AS_HEADER(PPAI) ;
1323         END IF ;
1324         SM_FIRST(PNO) := PNO ;
1325     END IF ;
1326 ELSE
1327     IF CORPO_EQUIVALENTE(PNO) THEN
1328         MSG("CORPO DUPLICADO",PNO) ;
1329     ELSE
1330         IF AS_SUBPROGRAM_DEF(PPAI) = DN_INSTANTIATION THEN
1331             PH := CONSTROI_HEADER(PPAI) ;
1332             SM_SPEC(PNO) := PH ;
1333         ELSE
1334             SM_SPEC(PNO) := AS_HEADER(PPAI) ;
1335         END IF ;
1336         SM_BODY(PNO) := AS_BLOCK_STUB(PPAI) ;
1337         IF EXISTE_ESPECIFICACAO(PNO) THEN
1338             PESP := APANHA_ESPECIFICACAO(PNO) ;
1339             SM_FIRST(PNO) := PESP ;
1340             SM_BODY(PESP) := PNO ;
1341         ELSE
1342             SM_FIRST(PNO) := PNO ;
1343         END IF ;
1344         IF EXISTE_STUB(PNO) THEN
1345             SM_STUB(PNO) := APANHA_STUB(PNO) ;
1346         END IF ;
1347     END IF ;
1348 END IF ;
1349 END VERIFICA_FUNCTION_ID ;
1350 -----
1351 ---
1352 --- SE O NO 'FUNCTION_ID' OCORRER NA SUB_ARVORE DE UMA INSTANCIACAO
1353 --- E NECESSARIO SE CONSTRUIR UMA SUB_ARVORE QUE REPRESENTA O HEADER
1354 --- DO SUBPROGRAMA GENERICO; O ATRIBUTO 'SM_SPEC' DO NO 'FUNCTION'
1355 --- APONTA PARA A SUB_ARVORE FORMADA.
1356 ---
1357 --- GENERIC => AS_ID ; ID ;
1358 --- AS_GENERIC_PARAMS => GENERIC_PARAMS ;
1359 --- AS_GENERIC_HEADER ; GENERIC_HEADER ;
1360 ---
1361 --- GENERIC_ID => SM_GENERIC_PARAMS ; GENERIC_PARAMS ;
1362 --- SM_SPEC ; GENERIC_HEADER ;
1363 --- SM_BODY ; BODY_STUB_VOID ;
1364 --- SM_FIRST ; DEF_OCCURRENCE ;
1365 --- SM_STUB ; DEF_OCCURRENCE ;
1366 ---
1367 PROCEDURE VERIFICA_GENERIC(PNO : PT_TREE ) IS
1368 PID , POPS , PGH : PT_TREE ;
1369 BEGIN
1370     PID := AS_ID(PNO) ;
1371     POPS := AS_GENERIC_PARAMS(PNO) ;
1372     PGH := AS_GENERIC_HEADER(PNO) ;
1373     VERIFICA_GENERIC_ID(PID , PNO) ;
1374     VERIFICA_GENERIC_PARAMS(POPS) ;
1375     CASE APANHA_NOME_NO(PGH) IS
1376     WHEN DN_PROCEDURE =>
1377         VERIFICA_PROCEDURE(PGH , PNO) ;
1378     WHEN DN_FUNCTION =>
1379         VERIFICA_FUNCTION(PGH , PNO) ;
1380     WHEN DN_PACKAGE_SPEC =>

```

```

1381     VERIFICA_PACKAGE_SPEC(PNO , PNO) ¶
1382     WHEN OTHERS =>
1383         NULL ¶
1384     END CASE ¶
1385 END VERIFICA_GENERIC ¶
1386 -----
1387 PROCEDURE VERIFICA_GENERIC_ID(PNO , PPAT : PT_TREE) IS
1388 BEGIN
1389     IF DECLARADO(PNO) THEN
1390         MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PNO)¶
1391     ELSIF NOT IDENTIFICADOR(PNO) THEN
1392         MSG("ESPERADOR IDENTIFICADOR COMO DESIGNADOR",PNO)¶
1393     ELSE
1394         SM_GENERIC_PARAMS(PNO) := AS_GENERIC_PARAMS(PPAT) ¶
1395         SM_SPEC(PNO) := AS_GENERIC_HEADER(PPAT) ¶
1396         SM_FIRST(PNO) := PNO ¶
1397         SM_BODY(PNO) := APONTA_VOID ¶
1398         SM_STUB(PNO) := APONTA_VOID ¶
1399     END IF ¶
1400 END VERIFICA_GENERIC_ID ¶
1401 -----
1402 ---
1403 --- O DESIGNADOR DE UM SUBPROGRAMA GENERICO DEVE SER UM IDENTIFICADOR
1404 --- (ARM-SC.12.1-4)¶ A FUNCAO 'IDENTIFICADOR' REALIZA ESSE TESTE.
1405 --- OS ATRIBUTOS 'SM_BLOCK' E 'SM_STUB' SAO INICIALIZADOS COM UM
1406 --- PONTEIRO PARA O NO 'VOID'.
1407 ---
1408 --- GENERIC_PARAMS => AS_LIST : SER OF GENERIC_ASSOC ¶
1409 ---
1410 PROCEDURE VERIFICA_GENERIC_PARAMS(PNO : PT_TREE) IS
1411 PGP : PT_TREE ¶
1412 BEGIN
1413     PGP := AS_LIST(PNO)¶
1414     WHILE PGP /= NIL
1415     LOOP
1416         CASE APANNA_NOME_NO(PGP) IS
1417             WHEN DN_IN =>
1418                 VERIFICA_IN(PGP) ¶
1419             WHEN DN_IN_OUT =>
1420                 VERIFICA_IN_OUT(PGP) ¶
1421             WHEN DN_TYPE =>
1422                 VERIFICA_TYPE(PGP) ¶
1423             WHEN DN_SUBPROGRAM_DECL =>
1424                 VERIFICA_SUBPROGRAM_DECL(PGP) ¶
1425             WHEN OTHERS =>
1426                 NULL ¶
1427             END CASE ¶
1428             PGP := NEXT(PGP) ¶
1429         END LOOP ¶
1430 END VERIFICA_GENERIC_PARAMS ¶
1431 -----
1432 ---
1433 --- GOTO => AS_NAME : NAME ¶
1434 ---
1435 PROCEDURE VERIFICA_GOTO(PNO : PT_TREE) IS
1436 BEGIN
1437     ENFILERA_GOTO(PNO) ¶
1438 END VERIFICA_GOTO ¶
1439 -----
1440 ---

```

```

1441 --- A ROTINA 'ENFILERA_BOTO' COLOCA O NO CORRESPONDENTE AO COMANDO
1442 --- SENDO ANALISADO EM UMA FILA? AO SE TERMINAR UM BLOCO, SAO
1443 --- ANALISADOS OS COMANDOS QUE DESVIAM PARA ROTULOS DEFINIDOS NO
1444 --- BLOCO EM QUESTAO.
1445 ---
1446 --- IDS => AS_LIST : SEQ OF IDS ?
1447 ---
1448 --- ESSA SUB_ARVORE E ANALISADA DIRETAMENTE EM SEU CONTEXTO EXTERNO.
1449 ---
1450 --- IF => AS_LIST : SEQ OF COND. CLAUSE ?
1451 ---
1452 PROCEDURE VERIFICA_IF(PNO : PT_TREE) IS
1453 PCC : PT_TREE ?
1454 BEGIN
1455     PCC := AS_LIST(PNO) ?
1456     WHILE PCC /= NIL
1457     LOOP
1458         VERIFICA_COND_CLAUSE(PCC) ?
1459         PCC := NEXT(PCC) ?
1460     END LOOP ?
1461 END VERIFICA_IF ?
1462 -----
1463 ---
1464 --- IN => AS_IDS : IDS ?
1465 ---     AS_NAME : NAME ?
1466 ---     AS_EXP_VOID : EXP_VOID ?
1467 ---
1468 --- IN_ID => SM_OBJ_TYPE : TYPE_SPEC ?
1469 ---     SM_INIT_EXP : EXP_VOID ?
1470 ---     SM_FIRST : DEF_OCCURRENCE ?
1471 ---
1472 PROCEDURE VERIFICA_IN(PNO , PDESTO : PT_TREE) IS
1473 PIS , PN , PE , PID : PT_TREE ?
1474 BEGIN
1475     PIDS := AS_IDS(PNO) ?
1476     PN := AS_NAME(PNO) ?
1477     PE := AS_EXP_VOID(PNO) ?
1478     RESOLVE_NOME_E_EXPRESSAO(PN) ?
1479     VERIFICA_NAME(PN) ?
1480     POBJ := APANHA_OBJETO(PN) ?
1481     IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
1482     TIPO_INVALIDO_PARA_PARAMETRO_IN(SM_DEFN(POBJ)) THEN
1483     MSG("ESPECIFICACAO DE TIPO INVALIDA",PN) ?
1484     ELSE
1485     IF APANHA_NOME_NO(PE) /= DN_VOID THEN
1486     RESOLVE_NOME_E_EXPRESSAO(PE) ?
1487     VALOR_INICIAL_PARAMETRO(PE) ?
1488     IF NOT TIPOS_COMPETIVEIS(PN,PE) THEN
1489     MSG("TIPO EXPRESSAO INICIAL INVALIDO",PE) ?
1490     ELSE
1491     PID := AS_LIST(PIDS) ?
1492     WHILE PID /= NIL
1493     LOOP
1494     VERIFICA_IN_ID(PID , PNO , PDESTO) ?
1495     PID := NEXT(PID) ?
1496     END LOOP ?
1497     END IF ?
1498     ELSE
1499     PID := AS_LIST(PIDS) ?
1500     WHILE PID /= NIL

```



```

1501     LOOP
1502         VERIFICA_IN_ID(PID , PNO , PDESIG) ;
1503         PID := NEXT(PID) ;
1504     END LOOP ;
1505 END IF ;
1506 ELSE
1507     PID := AS_LIST(PIDS) ;
1508     WHILE PID /= NIL
1509     LOOP
1510         VERIFICA_IN_ID(PID , PNO , PDESIG) ;
1511         PID := NEXT(PID) ;
1512     END LOOP ;
1513 END IF ;
1514 END IF ;
1515 END VERIFICA_IN ;
1516 -----
1517 PROCEDURE VERIFICA_IN_ID(PNO , PPAI , PDESIG : PT_TREE) IS
1518 PF : PT_TREE ;
1519 BEGIN
1520     SM_OBJ_TYPE(PNO) := AS_NAME(PPAI) ;
1521     SM_INIT_EXP(PNO) := AS_EXP_VOID(PPAI) ;
1522     IF SM_FIRST(PDESIG) = AS_EXP_VOID(PPAI) THEN
1523         SM_FIRST(PNO) := PNO ;
1524     ELSE
1525         PF := APANHA_FIRST(PDESIG , PNO) ;
1526         SM_FIRST(PNO) := PF ;
1527     END IF ;
1528 END VERIFICA_IN_ID ;
1529 -----
1530 ---
1531 --- O USO DE UM NOME QUE DENOTE UM PARAMETRO FORMAL NAO E PERMITIDO
1532 --- EM UMA EXPRESSAO DEFAULT DE UMA PARTE FORMAL SE A ESPECIFICACAO
1533 --- DO PARAMETRO FOR DADA NA MESMA PARTE FORMAL (ARM-SC.6.1-5) ; A
1534 --- ROTINA 'VALOR_INICIAL_PARAMETRO' VERIFICA SE ESSA CONDICAO FOI
1535 --- SATISFEITA.
1536 --- O 2. ARGUMENTO DA ROTINA _PDESIG_ APONTA PARA O DESIGNADOR DO
1537 --- SUBPROGRAMA QUE ESTA SENDO ANALISADO ; ELE E USADO PARA PREENCHER
1538 --- O ATRIBUTO 'SM_FIRST' DO IDENTIFICADOR.
1539 --- O ATRIBUTO 'SM_DEFN' DO NO 'POBJ' DEVE APONTAR PARA UM NO DO
1540 --- TIPO 'TYPE_ID', 'PRIVATE_TYPE_ID' OU 'L_PRIVATE_TYPE_ID' ; NO
1541 --- CASO DE APONTAR PARA A ESPECIFICACAO DE UM TIPO PRIVADO LIMITADO
1542 --- , E NECESSARIO QUE O PARAMETRO ESTEJA NA ESPECIFICACAO DE UM
1543 --- SUBPROGRAMA QUE ESTEJA NA PARTE VISIVEL DO MESMO PACOTE EM QUE
1544 --- FOI DECLARADO O TIPO PRIVADO ; ESSES TESTES SAO REALIZADOS PELA
1545 --- FUNCAO 'TIPO_INVALIDO_PARAMETRO_IN'.
1546 ---
1547 --- IN_OUT => AS_ID_S : ID_S ,
1548 ---             AS_NAME : NAME ,
1549 ---             AS_EXP_VOID : EXP_VOID ;
1550 ---
1551 --- IN_OUT_ID => SM_OBJ_TYPE : TYPE_SPEC ,
1552 ---              SM_FIRST : DEF_OCCURRENCE ;
1553 ---
1554 PROCEDURE VERIFICA_IN_OUT(PNO , PDESIG : PT_TREE) IS
1555 PIS , FN , PID , PIS , PF : PT_TREE ;
1556 BEGIN
1557     PIS := AS_ID_S(PNO) ;
1558     FN := AS_NAME(PNO) ;
1559     RESOLVE_NOME_E_EXPRESSAO(FN) ;
1560     VERIFICA_NAME(FN) ;

```

```

1561 IF APANHA_NOME_NO(PDESIG) = DN_FUNCTION_ID THEN
1562     MSG("TIPO DE PARAMETRO INVALIDO PARA FUNCAO">PNO) ?
1563 END IF ?
1564 POBJ := APANHA_OBJETO(PN) ?
1565 IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
1566     TIPO_INVALIDO_PARAMETRO_IN_OUT(SM_DEFN(POBJ)) THEN
1567     MSG("ESPECIFICACAO DE TIPO INVALIDA">PN) ?
1568 ELSE
1569     FID := AS_LIST(FIDS) ?
1570     PTS := SM_TYPE_SFEC(SM_DEFN(PNO)) ?
1571     WHILE FID /= NIL
1572     LOOP
1573         SM_OBJ_TYPE(FID) := PTS ?
1574         IF SM_FIRST(PDESIG) = PDESIG THEN
1575             SM_FIRST(PNO) := PNO ?
1576         ELSE
1577             PF := APANHA_FIRST(PDESIG , FID) ?
1578             SM_FIRST(FID) := PF ?
1579         END IF ?
1580         FID := NEXT(FID) ?
1581     END LOOP ?
1582 END IF ?
1583 END VERIFICA_IN_OUT ?
1584 -----
1585 ---
1586 --- O PROCESSAMENTO DO NO 'IN_OUT_ID' E FEITO NA PROPRIA ROTINA
1587 --- 'VERIFICA_IN_OUT_ID'.
1588 --- VER DEMAIS OBSERVACOES NAS ROTINAS 'VERIFICA_IN' E
1589 --- 'VERIFICA_IN_ID'.
1590 ---
1591 ---
1592 --- INDEX => AS_NAME : NAME ?
1593 ---
1594 PROCEDURE VERIFICA_INDEX(PNO : PT_TREE) IS
1595 POBJ : PT_TREE ?
1596 BEGIN
1597     PN := AS_NAME(PNO) ?
1598     RESOLVE_NOME_E_EXPRESSAO(PN) ?
1599     VERIFICA_EXP(PN) ?
1600     POBJ := APANHA_OBJETO(PN) ?
1601     IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
1602     ---
1603     --- INDEX => AS_NAME : NAME ?
1604     ---
1605 PROCEDURE VERIFICA_INDEX(PNO : PT_TREE) IS
1606 POBJ : PT_TREE ?
1607 BEGIN
1608     PN := AS_NAME(PNO) ?
1609     RESOLVE_NOME_E_EXPRESSAO(PN) ?
1610     VERIFICA_EXP(PN) ?
1611     POBJ := APANHA_OBJETO(PN) ?
1612     IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
1613         APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_TYPE_ID THEN
1614         MSG("ESPERADO IDENTIFICADOR DE TIPO">PN) ?
1615     ELSIF NOT TIPO_DISCRETO(SM_DEFN(POBJ)) THEN
1616         MSG("ESPERADOR IDENTIFICADOR DE TIPO">PN) ?
1617     END IF ?
1618 END VERIFICA_INDEX ?
1619 -----
1620 ---

```

```

1621 --- O TIPO DE CADA INDICE DE UM ARRAY DEVE SER DISCRETO (ARM-SC.3.6-1)
1622 --- # A FUNCAO 'TIPO_DISCRETO' VERIFICA SE O NO APONTADO POR SEU
1623 --- PARAMETRO (DO TIPO 'TYPE_ID') REPRESENTA UM TIPO CORRETO.
1624 ---
1625 --- INDEXED => AS_NAME : NAME ;
1626 ---           AS_EXP_S : EXP_S ;
1627 ---           SK_EXP_TYPE : TYPE_SPEC ;
1628 ---
1629 PROCEDURE VERIFICA_INDEXED(PNO : PT_TREE) IS
1630 PN , PES , PE : PT_TREE ;
1631 BEGIN
1632   PN := AS_NAME(PNO) ;
1633   PES := AS_EXP_S(PNO) ;
1634   IF TIPO_DEFINIDO(PNO) THEN
1635     VERIFICA_NAME(PN) ;
1636     PE := AS_LIST(PES) ;
1637     WHILE PE /= NIL
1638     LOOP
1639       VERIFICA_EXP(PE) ;
1640       PE := NEXT(PE) ;
1641     END LOOP ;
1642   END IF ;
1643 END VERIFICA_INDEXED ;
1644 -----
1645 ---
1646 --- O TIPO DE CADA INDICE DE UM ARRAY DEVE SER DISCRETO (ARM-SC.3.6-1)
1647 --- # A FUNCAO 'TIPO_DISCRETO' VERIFICA SE O NO APONTADO POR SEU
1648 --- PARAMETRO (DO TIPO 'TYPE_ID') REPRESENTA UM TIPO CORRETO.
1649 ---
1650 --- INDEXED => AS_NAME : NAME ;
1651 ---           AS_EXP_S : EXP_S ;
1652 ---           SK_EXP_TYPE : TYPE_SPEC ;
1653 ---
1654 PROCEDURE VERIFICA_INDEXED(PNO : PT_TREE) IS
1655 PN , PES , PE : PT_TREE ;
1656 BEGIN
1657   PN := AS_NAME(PNO) ;
1658   PES := AS_EXP_S(PNO) ;
1659   IF TIPO_DEFINIDO(PNO) THEN
1660     VERIFICA_NAME(PN) ;
1661     PE := AS_LIST(PES) ;
1662     WHILE PE /= NIL
1663     LOOP
1664       VERIFICA_EXP(PE) ;
1665       PE := NEXT(PE) ;
1666     END LOOP ;
1667   END IF ;
1668 END VERIFICA_INDEXED ;
1669 -----
1670 ---
1671 --- NOTE QUE A SUB_ARVORE DO NO INDEXED JA FOI PROCESSADA UMA VEZ.
1672 --- A FASE DE RESOLUCAO DE 'OVERLOADING'. DESSA FORMA A ROTINA
1673 --- 'VERIFICA_INDEXED' PODE VERIFICAR SE O TIPO DO NO ESTA DEFINIDO
1674 ---
1675 --- INNER_RECORD => AS_LIST : SEQ OF COMP ;
1676 ---
1677 PROCEDURE VERIFICA_INNER_RECORD(PNO : PT_TREE) IS
1678 ---
1679 --- INDEX => AS_NAME : NAME ;
1680 ---

```

```

1681 PROCEDURE VERIFICA_INDEX(PNO : PT_TREE) IS
1682 FORJ : PT_TREE ;
1683 BEGIN
1684   PN := AS_NAME(PNO) ;
1685   RESOLVE_NOME_E_EXPRESSAO(PN) ;
1686   VERIFICA_EXP(PN) ;
1687   FORJ := APANHA_OBJETO(PN) ;
1688   IF APANHA_NOME_NO(FORJ) /= DN_USED_NAME_ID OR ELSE
1689     APANHA_NOME_NO(SM_DEFN(FORJ)) /= DN_TYPE_ID THEN
1690     MSG("ESPERADO IDENTIFICADOR DE TIPO",PN) ;
1691   ELSIF NOT TIPO_DISCRETO(SM_DEFN(FORJ)) THEN
1692     MSG("ESPERADO IDENTIFICADOR DE TIPO",PN) ;
1693   END IF ;
1694 END VERIFICA_INDEX ;
1695 -----
1696 ---
1697 --- O TIPO DE CADA INDICE DE UM ARRAY DEVE SER DISCRETO(ARM-SC.3.6-1)
1698 --- ; A FUNCAO 'TIPO_DISCRETO' VERIFICA SE O NO APONTADO POR SEU
1699 --- PARAMETRO (DO TIPO 'TYPE_ID') REPRESENTA UM TIPO CORRETO.
1700 ---
1701 --- INDEXED => AS_NAME : NAME ,
1702 ---           AS_EXPLS : EXPLS ,
1703 ---           SM_EXP_TYPE : TYPE_SPEC ;
1704 ---
1705 PROCEDURE VERIFICA_INDEXED(PNO : PT_TREE) IS
1706 PN , PES , PE : PT_TREE ;
1707 BEGIN
1708   PN := AS_NAME(PNO) ;
1709   PES := AS_EXPLS(PNO) ;
1710   IF TIPO_DEFINIDO(PNO) THEN
1711     VERIFICA_NAME(PN) ;
1712     PE := AS_LIST(PES) ;
1713     WHILE PE /= NIL
1714     LOOP
1715       VERIFICA_EXP(PE) ;
1716       PE := NEXT(PE) ;
1717     END LOOP ;
1718   END IF ;
1719 END VERIFICA_INDEXED ;
1720 -----
1721 ---
1722 --- NOTE QUE A SUB_ARVORE DO NO INDEXED JA FOI PROCESSADA UMA VEZ
1723 --- A FASE DE RESOLUCAO DE 'OVERLOADING'. DESSA FORMA A ROTINA
1724 --- 'VERIFICA_INDEXED' PODE VERIFICAR SE O TIPO DO NO ESTA DEFINIDO
1725 ---
1726 --- INNER_RECORD => AS_LIST : SEQ OF COMP ;
1727 ---
1728 PROCEDURE VERIFICA_INNER_RECORD(PNO : PT_TREE) IS
1729
1730
1731 *****
1732
1733
1734
1735 U-
1736 --- NAMED_STM => AS_ID : ID ,      --- NAMED_STM_ID
1737 ---           AS_STMS : STM ;      --- LOOP OU BLOCK
1738 ---
1739 --- NAMED_STM_ID : SM_STM : STM ;  --- NAMED_STM
1740 ---

```

```

1741 PROCEDURE VERIFICA_NAMED_STM(PNO : PT_TREE) IS
1742 PN , PSTM : PT_TREE ;
1743 BEGIN
1744   PN := AS_ID(PNO) ;
1745   PSTM := AS_STM(PNO) ;
1746   IF DECLARADO(PNO) THEN
1747     MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PN) ;
1748   ELSE
1749     SM_STM(PN) := PNO ;
1750   END IF ;
1751   IF APANHA_NOME_NO(PSTM) = DN_LOOP THEN
1752     VERIFICA_LOOP(PSTM) ;
1753   ELSE
1754     VERIFICA_BLOCK(PSTM) ;
1755   END IF ;
1756 END VERIFICA_NAMED_STM ;
1757 -----
1758 ---
1759 --- NO_DEFAULT => ;
1760 ---
1761 --- NOT_IN => ;
1762 ---
1763 --- NAO HA NECESSIDADE DE TESTES SEMANTICOS ASSOCIADOS A ESSES NOS.
1764 ---
1765 --- NULL_ACCESS => SM_EXP_TYPE : TYPE_SPEC ,
1766 ---                   SM_VALUE : VALUE ;
1767 ---
1768 --- O ATRIBUTO 'SM_EXP_TYPE' DO NO JA FOI CALCULADO PELA ROTINA
1769 --- 'VERIFICA_NOME_E_EXPRESSAO' .
1770 --- EMBORA EXISTA O ATRIBUTO 'SM_VALUE' NA NOVE VERSAO DE DIANA,
1771 --- ELE ESTA INCORRETO , UMA VEZ QUE ESSE VALOR NAO PODE SER
1772 --- ESCALAR.
1773 ---
1774 --- NULL_COMP => ;
1775 ---
1776 --- NULL_STM => ;
1777 ---
1778 --- NAO HA NECESSIDADE DE TESTES SEMANTICOS ASSOCIADOS A ESSE NO.
1779 ---
1780 --- NUMBER => AS_IDLS : IDLS ,
1781 ---             AS_EXP : EXP ;
1782 ---
1783 --- NUMBER_ID => SM_OBJ_TYPE : TYPE_SPEC ,
1784 ---              SM_INIT_EXP : EXP ;
1785 ---
1786 PROCEDURE VERIFICA_NUMBER(PNO : PT_TREE) IS
1787 PIDS , FID , FE : PT_TREE ;
1788 BEGIN
1789   PIDS := AS_IDLS(PNO) ;
1790   FE := AS_EXP(PNO) ;
1791   RESOLVE_NOME_E_EXPRESSAO(FE) ;
1792   VERIFICA_EXP(FE) ;
1793   FID := AS_LIST(PIDS) ;
1794   WHILE FID /= NIL
1795     LOOP
1796     IF DECLARADO(FID) THEN
1797       MSG("IDENTIFICADOR JA DECLARADO NESSE ESCOPO",FID) ;
1798     ELSE
1799       SM_OBJ_TYPE(FID) := APANHA_TIPO(FE) ;
1800       SM_INIT_EXP(FID) := FE ;

```

```

1801     END IF ;
1802     PID := NEXT(FID) ;
1803     END LOOP ;
1804 END VERIFICA_NUMBER ;
1805 -----
1806 ---
1807 --- VER COMENTARIOS DAS ROTINAS DOS NOS 'CONSTANT' E 'CONST.ID' .
1808 ---
1809 --- NUMERIC_LITERAL => SM_EXP_TYPE : TYPE_SPEC ,
1810 ---                      SM_VALUE : VALUE ;
1811 ---
1812 PROCEDURE VERIFICA_NUMERIC_LITERAL(PNO : PT_TREE ;
1813                                   ESTAT : BOOLEAN := FALSE) IS
1814 BEGIN
1815     IF SM_EXP(PNO) /= TIPO_GERAL AND THEN
1816         ESTAT THEN
1817         MARCA_VALOR_ESTATICO(PNO) ;
1818     END IF ;
1819 END VERIFICA_NUMERIC_LITERAL ;
1820 -----
1821 ---
1822 --- O TIPO DO LITERAL ('NUMERIC_LITERAL' OU 'NUMERIC_INTEGER') JA FOI
1823 --- CALCULADO PELA ROTINA 'RESOLVE_NOME_E_EXPRESSAO' .
1824 ---
1825 --- OR_ELSE => ;
1826 --- NAO EXISTE ROTINA ASSOCIADA A ESSE NO , UMA VEZ QUE TODA
1827 --- SEMANTICA JA FOI REALIZADA NO NO 'BINARY' .
1828 ---
1829 --- OTHERS => ;
1830 ---
1831 --- NAO EXISTE ROTINA ASSOCIADA A ESSE NO , JA QUE TODA SEMANTICA
1832 --- DESSE NO ( POR EX. ROTINA 'VERIFICA_DESCrpt_Aggregate' ) .
1833 ---
1834 --- OUT => AS_IDS : IDS ,
1835 ---          AS_NAME : NAME ,
1836 ---          AS_EXP_VOID => EXP_VOID ;
1837 ---
1838 --- OUT_ID => SM_OBJ_TYPE : TYPE_SPEC ,
1839 ---          SM_FIRST : DEF_OCCURENCE ;
1840 ---
1841 PROCEDURE VERIFICA_OUT(PNO , PDESIG : PT_TREE) IS
1842 PIS , PID , PN , POBJ , PTF : PT_TREE ;
1843 BEGIN
1844     PIS := AS_IDS(PNO) ;
1845     PN := AS_NAME(PNO) ;
1846     RESOLVE_NOME_E_EXPRESSAO(PN) ;
1847     VERIFICA_NAME(PN) ;
1848     POBJ := APANHA_OBJETO(PN) ;
1849     IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
1850         TIPO_INVALIDO_PARAMETRO_OUT(SM_DEFN(POBJ)) THEN
1851         MSG("ESPECIFICACAO INVALIDA PARA PARAMETRO 'OUT'",PN) ;
1852     ELSE
1853         PID := AS_LIST(PIDS) ;
1854         PTF := SM_TYPE_SPEC(SM_DEFN(POBJ)) ;
1855         WHILE PID /= NIL
1856         LOOP
1857             SM_OBJ_TYPE(PID) := AS_NAME(PNO) ;
1858             IF SM_FIRST(PDESIG) = PDESIG THEN
1859                 SM_FIRST(PNO) := PNO ;
1860             ELSE

```

```

1861         SM_FIRST(PID) := APANHA_FIRST(PDEF1 , PID) ;
1862     END IF ;
1863     PID := NEXT(PID) ;
1864 END LOOP ;
1865 END IF ;
1866 END VERIFICA_OUT ;
1867 -----
1868 ---
1869 --- VER OBSERVACOES DA ROTINA 'IN_OUT' .
1870 ---
1871 --- PACKAGE_ID => SM_SPEC : PACKAGE_SPEC ;
1872 ---         SM_BODY : PACK_BODY_DESC ;
1873 ---         SM_ADDRESS : EXP_VOID ;
1874 ---         SM_STUB : DEF_OCCURRENCE ;
1875 ---
1876 --- PACKAGE_BODY => AS_ID : ID ;
1877 ---         AS_BLOCK_STUB : BLOCK_STUB ;
1878 ---
1879 PROCEDURE VERIFICA_PACKAGE_BODY(PNO : PT_TREE) IS
1880 PID , PBST , PDEF , PDEF1 , PDEF2 : PT_TREE ;
1881 BEGIN
1882     PID := AS_ID(PNO) ;
1883     PBST := AS_BLOCK_STUB(PNO) ;
1884     IF NOT DECLARADO(PID) THEN
1885         MSG("FALTA ESPECIFICACAO DO PACOTE",PID) ;
1886     ELSE
1887         PDEF := SM_DEFN(PID) ;
1888         IF APANHA_NOME_NO(PDEF) /= DN_PACKAGE_ID THEN
1889             MSG("FALTA ESPECIFICACAO DE PACOTE",PID) ;
1890         ELSE
1891             SM_FIRST(PID) := PDEF ;
1892             SM_SPEC(PID) := SM_SPEC(PDEF) ;
1893             IF APANHA_NOME_NO(PBST) = DN_BLOCK THEN
1894                 SM_BODY(PDEF) := PBST ;
1895                 SM_BODY(PID) := PBST ;
1896                 VERIFICA_BLOCK(PBST) ;
1897                 SUBPROGRAMAS_PRESENTES(PNO) ;
1898             ELSE --- DN_STUB
1899                 PDEF2 := APANHA_CORPO(PID) ;
1900                 SM_STUB(PDEF) := PBST ;
1901                 IF APANHA_NOME_NO(PDEF2) /= DN_VOID THEN
1902                     SM_STUB(PDEF2) := PBST ;
1903                 END IF ;
1904             END IF ;
1905         END IF ;
1906     END IF ;
1907 END VERIFICA_PACKAGE_BODY ;
1908 -----
1909 ---
1910 --- TODOS PACOTES POSSUEM UMA ESPECIFICACAO (ARM-SC.7.1-1).
1911 --- SE UMA DECLARACAO DE SUBPROGRAMA, DE PACOTE, DE TASK OU DE UMA
1912 --- UNIDADE GENERICA FOR UM ITEM DA PARTE DECLARATIVA DE CERTO
1913 --- PACOTE, ENTAO O CORPO CORRESPONDENTE DEVE SER UM ITEM DA PARTE
1914 --- DECLARATIVA DO CORPO DO PACOTE EM QUESTAO (ARM-SC.7.1-4) ISSO
1915 --- E VERIFICADO PELA ROTINA 'SUBPROGRAMAS_PRESENTES' .
1916 ---
1917 --- PACKAGE_DECL => AS_ID : ID ;
1918 ---         AS_PACKAGE_DEF : PACKAGE_DEF ;
1919 ---
1920 ---

```

```

1921 PROCEDURE PACKAGE_DECL(PNO : PT_TREE) IS
1922 PID , PPD : PT_TREE ;
1923 BEGIN
1924   PID := AS_ID(PNO) ;
1925   PPD := AS_PACKAGE_DEF(PNO) ;
1926   IF DECLARADO(PID) THEN
1927     MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PID) ;
1928   ELSE
1929     SM_FIRST(PID) := PID ;
1930     SM_SPEC(PID) := PPD ;
1931     CASE APANHA_NOME_NO(PPD) IS
1932       WHEN DN_INSTANTIATION =>
1933         VERIFICA_INSTANTIATION(PPD) ;
1934       WHEN DN_PACKAGE_SPEC =>
1935         VERIFICA_PACKAGE_SPEC(PPD) ;
1936       WHEN DN_RENAME =>
1937         VERIFICA_RENAME(PPD) ;
1938       WHEN OTHERS =>
1939         NULL ;
1940     END CASE ;
1941   END IF ;
1942 END PACKAGE_DEF ;
1943 -----
1944 ---
1945 --- PACKAGE_SPEC => AS_DECL_S1 : DECL_S ;
1946 --- AS_DECL_S2 : DECL_S ;
1947 ---
1948 PROCEDURE VERIFICA_PACKAGE_SPEC(PNO : PT_TREE) IS
1949 PDS1 , PDS2 , PD1 , PD2 : PT_TREE ;
1950 BEGIN
1951   PDS1 := AS_DECL_S1(PNO) ;
1952   PDS2 := AS_DECL_S2(PNO) ;
1953   PD1 := AS_LIST(PDS1) ;
1954   PD2 := AS_LIST(PDS2) ;
1955   WHILE PD1 /= NIL
1956   LOOP
1957     VERIFICA_DECL(PD1) ;
1958     PD1 := NEXT(PD1) ;
1959   END LOOP ;
1960   WHILE PD2 /= NIL
1961   LOOP
1962     VERIFICA_DECL(PD2) ;
1963     PD2 := NEXT(PD2) ;
1964   END LOOP ;
1965   TIPOS_PRIVADOS_COMPLETOS(PNO) ;
1966 END VERIFICA_PACKAGE_SPEC ;
1967 -----
1968 --- SE UMA DECLARACAO DE TIPO PRIVADO E DADA NA PARTE VISIVEL DE UM
1969 --- PACOTE, ENTAO A DECLARACAO COMPLETA CORRESPONDENTE DO TIPO DEVE
1970 --- APARECER NA PARTE DECLARATIVA DA PARTE PRIVADA DO PACOTE. ESSA
1971 --- DECLARACAO DEVE SER UMA DECLARACAO DE TIPO COMPLETO OU DE TASK
1972 --- (ARM-SC.7.4) O MESMO SE APLICA PARA CONSTANTES POSTERGADAS
1973 --- (ARM-SC.7.4.3-1) A ROTINA 'TIPOS_PRIVADOS_COMPLETOS' REALIZA
1974 --- ESSES TESTES.
1975 ---
1976 --- PARAM_ASSOC_S => AS_LIST : SEQ OF PARAM_ASSOC ;
1977 ---
1978 FUNCTION VERIFICA_PARAM_ASSOC_S(PNO,PPAI : PT_TREE) RETURN BOOLEAN IS
1979 PLO , PPAR , FN , FORBJ : PT_TREE ;
1980 ERRO , CHAVE_ASSOC : BOOLEAN := FALSE ;

```



```

1981 BEGIN
1982   PLO := AS_LIST(PNO) ;
1983   POSJ := APANHA_OBJETO(AS_NAME(PFAL)) ;
1984   PPAR := APANHA_PARAMETRO(POSJ) ;
1985   WHILE PLO /= NIL
1986     LOOP
1987       IF APANHA_NOME_NO(PLO) /= DN_ASSOC THEN
1988         IF CHAVE_ASSOC THEN
1989           MSG("NAO ERA ESPERADA ASSOCIACAO POR POSICAO",PLO) ;
1990           ERRO := TRUE ;
1991         ELSE
1992           PPAR := PROXIMO_PARAMETRO(PPAR) ;
1993           RESOLVE_NOME_E_EXPRESSAO(PLO) ;
1994           VERIFICA_EXP(PLO) ;
1995           IF NOT TIPOS_COMPATIVELIS(PLO,PPAR) THEN
1996             MSG("TIPO EXPRESSAO INVALIDO PARA ASSOCIACAO",PLO) ;
1997             ERRO := TRUE ;
1998           ELSE
1999             MARCA_PARAMETRO(PPAR) ;
2000           END IF ;
2001         END IF ;
2002       ELSE -- DN_ASSOC
2003         CHAVE_ASSOC := TRUE ;
2004         PEXP := AS_EXP(PLO) ;
2005         PN := AS_NAME(PLO) ;
2006         RESOLVE_NOME_E_EXPRESSAO(PEXP) ;
2007         VERIFICA_EXP(PEXP) ;
2008         PPAR := APANHA_PARAMETRO(PN) ;
2009         IF NOT PARAMETRO_VALIDO(PN) THEN
2010           MSG("PARAMETRO INVALIDO PARA FUNCAO",PN) ;
2011           ERRO := TRUE ;
2012         ELSIF NOT TIPOS_COMPATIVELIS(PEXP , PPAR) THEN
2013           MSG("TIPO DA EXPRESSAO INVALIDA",PEXP) ;
2014           ERRO := TRUE ;
2015         ELSIF JA_DEFINIDO(PN , PPAR) THEN
2016           MSG("PARAMETRO JA DEFINIDO",PN) ;
2017           ERRO := TRUE ;
2018         ELSE
2019           MARCA_PARAMETRO(PPAR) ;
2020         END IF ;
2021       END IF ;
2022       PLO := NEXT(PLO) ;
2023     END LOOP ;
2024   IF NOT LISTA_COMPLETA(PNO) THEN
2025     MSG("LISTA DE PARAMETROS INCOMPLETA",PNO) ;
2026   END IF ;
2027   RETURN ERRO ;
2028 END VERIFICA_PARAM_ASSOC_S ;
2029 -----
2030 ---
2031 --- ESSE NO E TRATADO COMO UMA FUNCAO DE MODO A PERMITIR QUE A LISTA
2032 --- DE PARAMETROS NORMALIZADA CORRESPONDENTE SO SEJA CONSTRUIDA CASO
2033 --- A ASSOCIACAO ESTEJA CORRETA .
2034 --- VER DEMAIS OBSERVACOES DA ROTINA 'VERIFICA_USCRMT_AGGREGATE', QUE
2035 --- POSSUI UMA SEMANTICA BASTANTE PARECIDA.
2036 --- PARA CADA PARAMETRO FORMAL DE UM SUBPROGRAMA, UMA CHAMADA DEVE
2037 --- ESPECIFICAR EXATAMENTE UM PARAMETRO REAL, EXPLICITAMENTE OU POR
2038 --- UMA EXPRESSAO DEFAULT.
2039 --- A FUNCAO 'LISTA_COMPLETA' VERIFICA SE TODOS OS PARAMETROS FORMAIS
2040 --- FORAM ESPECIFICADOS.

```

```

2041 ---
2042 --- PARAM_S => AS_LIST : SEQ OF PARAM ;
2043 ---
2044 PROCEDURE VERIFICA_PARAM_S(PNO : PT_TREE) IS
2045 PPAR : PT_TREE ;
2046 BEGIN
2047   PPAR := AS_LIST(PNO) ;
2048   WHILE PPAR /= NIL
2049     LOOP
2050       CASE AFANHA_NOME_NO(PPAR) IS
2051         WHEN DN_IN =>
2052           VERIFICA_IN(PPAR,PDESIG) ;
2053         WHEN DN_IN_OUT =>
2054           VERIFICA_IN_OUT(PPAR,PDESIG) ;
2055         WHEN DN_OUT =>
2056           VERIFICA_OUT(PPAR,PDESIG) ;
2057         WHEN OTHERS =>
2058           NULL ;
2059       END CASE ;
2060       PPAR := NEXT(PPAR) ;
2061     END LOOP ;
2062 END VERIFICA_PARAM_S ;
2063 -----
2064 ---
2065 --- PARENTHESIZED => AS_EXP : EXP ,
2066 ---                               SM_EXP_TYPE : TYPE_SPEC ,
2067 ---                               SM_VALUE : VALUE ;
2068 ---
2069 PROCEDURE VERIFICA_PARENTHESESIZED(PNO : PT_TREE ,
2070                                     ESTAT : BOOLEAN := FALSE) IS
2071 PEXP : PT_TREE ;
2072 BEGIN
2073   PEXP := AS_EXP(PNO) ;
2074   IF SM_EXP_TYPE(PNO) /= TIPO_GERAL THEN
2075     IF ESTAT THEN
2076       IF EXPRESSAO_ESTATICA(PNO) THEN
2077         MARCA_VALOR_ESTATICO(PNO) ;
2078       ELSE
2079         MSG("ESPERADA EXPRESSAO ESTATICA",PNO) ;
2080       END IF ;
2081     END IF ;
2082     VERIFICA_EXP(PEXP) ;
2083   END IF ;
2084 END VERIFICA_PARENTHESESIZED ;
2085 -----
2086 ---
2087 ---
2088 --- PRAGMA => AS_ID : ID ,
2089 ---                               AS_PARAM_ASSOC_S : PARAM_ASSOC_S ;
2090 ---
2091 PROCEDURE VERIFICA_PRAGMA(PNO : PT_TREE) IS
2092 PID , FPAS : PT_TREE ;
2093 BEGIN
2094   PID := AS_ID(PNO) ;
2095   FPAS := AS_PARAM_S(PNO) ;
2096   IF NOT DECLARADO(PID) OR ELSE
2097     AFANHA_NOME_NO(SM_DEFN(PID)) /= ON_PRAGMA_ID THEN
2098     AVISA("ESPERADO NOME DE PRAGMA",PID) ;
2099   ELSE
2100     RESOLVE_PRAGMA(PID , FPAS) ;

```

```

2101     END IF ?
2102 END VERIFICA_PRAGMA ?
2103 -----
2104 ---
2105 ---  EMBORA NAO EXISTA A PRODUCAO DE PRAGMA NA GRAMATICA DE ADA, O
2106 ---  ANALISADOR SINTACTICO E O RESPONSAVEL POR TESTAR SE OS PRAGMAS
2107 ---  APARECEM NOS LUGARES CORRETOS.
2108 ---  VER APENDICE I DO MANUAL DE DIANA, SOBRE O AMBIENTE PREDEFINIDO
2109 ---  DA LINGUAGEM, AS OBSERVACOES SOBRE O NO 'PRAGMA_ID'.
2110 ---  COMO CADA IMPLEMENTACAO PODE DEFINIR SEUS PROPRIOS PRAGMAS, O
2111 ---  APARECIMENTO DE UM PRAGMA DESCONHECIDO PARA UMA IMPLEMENTACAO
2112 ---  NAO CONFIGURA NECESSARIAMENTE UMA SITUACAO DE ERRO, DE MODO QUE
2113 ---  NESSAS SITUACOES UMA ROTINA QUE EMITE UMA MENSAGEM DE ADVERTENCIA
2114 ---  CADA PRAGMA POSSUI UMA SEMANTICA ESPECIFICA ASSOCIADA? OS TESTES
2115 ---  NECESSARIOS SAO REALIZADOS PELA ROTINA 'RESOLVE_PRAGMA' .
2116 ---
2117 --- PRIVATE => SM_DISCRIMINANTS : DSCRNT_VARS ?
2118 ---
2119 --- PRIVATE_TYPE_ID => SM_TYPE_SPEC : TYPE_SPEC ?
2120 ---
2121 PROCEDURE VERIFICA_PRIVATE(PNO , PPAI : PT_TREE) IS
2122 BEGIN
2123     IF NOT CONTEXTO_TIPO_PRIVADO(PNO) THEN
2124         MSG("CONTEXTO INVALIDO PARA TIPO PRIVADO",PNO) ?
2125     ELSE
2126         SM_DISCRIMINANTS(PNO) := AS_DSCRNT_VARS(PPAI) ?
2127     END IF ?
2128 END VERIFICA_PRIVATE ?
2129 -----
2130 PROCEDURE VERIFICA_PRIVATE_TYPE_ID(PNO : PT_TREE) IS
2131 BEGIN
2132     IF DECLARADO(PNO) THEN
2133         MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PNO) ?
2134     END IF ?
2135 END VERIFICA_PRIVATE_TYPE_ID ?
2136 -----
2137 ---
2138 ---  VERIFICAR OBSERVACOES DAS ROTINAS 'VERIFICA_LL_PRIVATE' E
2139 ---  'VERIFICA_LL_PRIVATE_TYPE_ID' .
2140 ---
2141 --- PROCEDURE => AS_PARAM_S : PARAM_S ?
2142 ---
2143 PROCEDURE VERIFICA_PROCEDURE(PNO , PPAI : PT_TREE) IS
2144 PPS : PT_TREE ?
2145 BEGIN
2146     PPS := AS_PARAM_S(PNO) ?
2147     CASE APANHA_NOME_NO(PPAI) IS
2148     WHEN DN_SUBPROGRAM_DECL =>
2149         IF DECLARACAO_EQUIVALENTE(PNO) THEN
2150             MSG("ESPECIFICACAO DUPLICADA",PNO) ?
2151         END IF ?
2152     WHEN DN_SUBPROGRAM_BODY =>
2153         IF CORPO_EQUIVALENTE(PNO) THEN
2154             MSG("CORPO DUPLICADO",PNO) ?
2155         END IF ?
2156     WHEN OTHERS =>
2157         NULL ?
2158     END CASE ?
2159     VERIFICA_PARAM_S(PPS) ?
2160 END VERIFICA_PROCEDURE ?

```

```

2161 -----
2162 ---
2163 --- VER OBSERVAÇÕES SOBRE AS FUNÇÕES 'DECLARACAO_EQUIVALENTE' E
2164 --- 'CORPO_EQUIVALENTE' NA ROTINA 'VERIFICA_DEF_DP' .
2165 ---
2166 --- PROCEDURE_CALL => AS_NAME : NAME ;
2167 --- AS_PARAM_ASSOC_S : PARAM_ASSOC_S ;
2168 ---
2169 PROCEDURE VERIFICA_PROCEDURE_CALL(PNO , PPAI : PT_TREE) IS
2170 PN , PPAS , PPAR : PT_TREE ;
2171 BEGIN
2172   PN := AS_NAME(PNO) ;
2173   PPAS := AS_PARAM_ASSOC_S(PNO) ;
2174   RESOLVE_NOME_E_EXPRESSAO(PN) ;
2175   VERIFICA_NAME(PN) ;
2176   IF VERIFICA_PARAM_ASSOC_S(PPAS,PNO) THEN
2177     MONTA_LISTA_NORMALIZADA(PNO) ;
2178   END IF ;
2179 END VERIFICA_PROCEDURE_CALL ;
2180 -----
2181 ---
2182 --- A ROTINA DE SOLICACAO DE NOME E EXPRESSAO, QUE JA FOI USADA NA
2183 --- MESMA SUBARVORE, GARANTE QUE ESTA REPRESENTA REALMENTE UMA CHAMADA
2184 --- DE ROTINA, DESSA FORMA NAO PRECISO FAZER NENHUM TESTE COM SEU
2185 --- IDENTIFICADOR.
2186 --- A FUNCAO 'VERIFICA_PARAM_ASSOC_S' DEVOLVE O VALOR VERDADEIRO SE A
2187 --- LISTA DE PARAMETROS FOR VALIDA PARA O SUBPROGRAMA EM QUESTAO.
2188 ---
2189 --- PROC_ID => SM_SPEC : HEADER ,
2190 --- SM_BODY : SUBP_BODY_DESC ,
2191 --- SM_LOCATION : LOCATION ,
2192 --- SM_STUB : DEF_OCCURRENCE ,
2193 --- SM_FIRST : DEF_OCCURRENCE ;
2194 ---
2195 PROCEDURE VERIFICA_PROC_ID(PNO , PPAI : PT_TREE) IS
2196 PH , PESF : PT_TREE ;
2197 BEGIN
2198   IF APANHA_NOME_NO(PPAI) = DN_SUBPROGRAM_DECL THEN
2199     IF DECLARACAO_EQUIVALENTE(PNO) THEN
2200       MSG("ESPECIFICACAO DUPLICADA",PNO) ;
2201     ELSE
2202       IF AS_SUBPROGRAM_DEF(PPAI) = DN_INSTANTIATION THEN
2203         PH := CONSTROI_HEADER(PPAI) ;
2204         SM_SPEC(PNO) := PH ;
2205       ELSE
2206         SM_SPEC(PNO) := AS_HEADER(PPAI) ;
2207       END IF ;
2208       SM_FIRST(PNO) := PNO ;
2209     END IF ;
2210   ELSE -- DN_SUBPROGRAM_BODY
2211     IF CORPO_EQUIVALENTE(PNO) THEN
2212       MSG("CORPO DUPLICADO",PNO) ;
2213     ELSE
2214       IF AS_SUBPROGRAM_DEF(PPAI) = DN_INSTANTIATION THEN
2215         PH := CONSTROI_HEADER(PPAI) ;
2216         SM_SPEC(PNO) := PH ;
2217       ELSE
2218         SM_SPEC(PNO) := PNO ;
2219       END IF ;
2220       SM_BODY(PNO) := AS_BLOCK_STUB(PPAI) ;

```

```

2221     IF EXISTE_ESPECIFICACAO(PNO) THEN
2222         PESP := APANHA_ESPECIFICACAO(PNO) ;
2223         SM_FIRST(PNO) := PESP ;
2224         SM_BODY(PESP) := PNO ;
2225     ELSE
2226         SM_FIRST(PNO) := PNO ;
2227     END IF ;
2228     IF EXISTE_STUB(PNO) THEN
2229         SM_STUB(PNO) := APANHA_STUB(PNO) ;
2230     END IF ;
2231     END IF ;
2232 END IF ;
2233 END VERIFICA_PROCLID ;
2234 -----
2235 ---
2236 --- QUALIFIED => AS_NAME : NAME ,
2237 ---          AS_EXP : EXP ,
2238 ---          SM_EXP_TYPE : TYPE_SPEC ,
2239 ---          SM_VALUE : VALUE ;
2240 ---
2241 PROCEDURE VERIFICA_QUALIFIED( PNO : PT_TREE ;
2242                               ESTAT : BOOLEAN := FALSE) IS
2243 PN , PE : PT_TREE ;
2244 BEGIN
2245     PN := AS_NAME(PNO) ;
2246     PE := AS_EXP(PNO) ;
2247     IF SM_EXP_TYPE(PNO) /= TIPO_GERAL THEN
2248         RESOLVE_NOME_E_EXPRESSAO(PN) ;
2249         VERIFICA_NOME_E_EXPRESSAO(PE) ;
2250         RESOLVE_NOME_E_EXPRESSAO(PE) ;
2251         VERIFICA_EXP(PE) ;
2252     IF ESTAT THEN
2253         IF EXPRESSAO_ESTATICA(PNO) THEN
2254             MARCA_VALOR_ESTATICO(PNO) ;
2255         ELSE
2256             MSG("ESPERADA EXPRESSAO ESTATICA">PE) ;
2257         END IF ;
2258     END IF ;
2259 END IF ;
2260 END VERIFICA_QUALIFIED ;
2261 -----
2262 ---
2263 --- RAISE => AS_NAME_VOID : NAME_VOID ;
2264 ---
2265 PROCEDURE VERIFICA_RAISE(PNO : PT_TREE) IS
2266 PN : PT_TREE ;
2267 BEGIN
2268     PN := AS_NAME(PNO) ;
2269     IF APANHA_NOME_NO(PN) = ON_VOID THEN
2270         IF NOT CONTEXTO_RAISE(PNO) THEN
2271             MSG("ESPERADO NOME DE EXCECAO">PNO) ;
2272         END IF ;
2273     ELSE
2274         RESOLVE_NOME_E_EXPRESSAO(PN) ;
2275         VERIFICA_NAME(PN) ;
2276         FORBJ := APANHA_OBJETO(PN) ;
2277         IF APANHA_NOME_NO(SM_DEFN(FORBJ)) /= ON_EXCEPTION_ID THEN
2278             MSG("ESPERADO NOME DE EXCECAO">PNO) ;
2279         END IF ;
2280     END IF ;

```

```

2281 END VERIFIVA_RAISE ;
2282 -----
2283 ---
2284 --- UM COMANDO 'RAISE' SEM UM NOME DE EXCECAO SO E PERMITIDO EM UM
2285 --- 'EXCEPTION-HANDLER' (MAS NAO EM UMA SEQUENCIA DE COMANDOS DE UM
2286 --- SUBPROGRAMA, FACOTE, UNIDADE TASK, UNIDADE GNERICA INTERNA AO
2287 --- HANDLER - ARM.SC.11.3-3) A FUNCAO 'CONTEXTO_RAISE' REALIZA ESSE
2288 --- TESTE.
2289 ---
2290 ---
2291 --- RANGE => AS_EXP1 : EXP ,
2292 ---          AS_EXP2 : EXP ,
2293 ---          SM_BASE_TYPE : TYPE_SPEC ;
2294 PROCEDURE VERIFICA_RANGE(PNO , PPAI : PT_TREE) IS
2295 PE1 , PE2 : PT_TREE ;
2296 BEGIN
2297     PE1 := AS_EXP1(PNO) ;
2298     PE2 := AS_EXP2(PNO) ;
2299     RESOLVE_NOME_E_EXPRESSAO(PE1) ;
2300     VERIFICA_EXP(PE1) ;
2301     RESOLVE_NOME_E_EXPRESSAO(PE2) ;
2302     VERIFICA_EXP(PE2) ;
2303 END VERIFICA_RANGE ;
2304 -----
2305 ---
2306 --- OS DEMAIS TESTES QUE DEVEM SER REALIZADOS NESSA ARVORE (POR EX.,
2307 --- TIPO DE EXPRESSOES) SAO REALIZADOS DIRETAMENTE PELA ROTINA QUE
2308 --- CHAMOU 'VERIFICA_RANGE', DEVIDO AS DIFERENTES SITUACOES
2309 --- SEMANTICAS EM QUE ESSE NAO APARECE.
2310 ---
2311 --- RECORD => AS_LIST : SEQ OF COMP ,
2312 ---          SM_SIZE : EXP_VOID ,
2313 ---          SM_DISCRIMINANTES : DISCRNT_VARS ,
2314 ---          SM_PACKING : BOOLEAN ,
2315 ---          SM_RECORD_SPEC : REP_VOID ;
2316 ---
2317 ---
2318 PROCEDURE VERIFICA_RECORD(PNO , PPAI : PT_TREE) IS
2319 PCOM : PT_TREE ;
2320 BEGIN
2321     SM_SIZE(PNO) := APONTA_VOID ;
2322     SM_DISCRIMINANTS(PNO) := AS_DISCRNT_VARS(PPAI) ;
2323     SM_PACKING(PNO) := FALSE ;
2324     SM_RECORD_SPEC(PNO) := APONTA_VOID ;
2325     PCOM := AS_LIST(PNO) ;
2326     WHILE PCOM /= NIL
2327     LOOP
2328         CASE AFANHA_NOME_NO(PCOM) IS
2329             WHEN DN_PRAGMA =>
2330                 VERIFICA_PRAGMA(PCOM) ;
2331             WHEN DN_VAR =>
2332                 VERIFICA_CHAR(PCOM) ;
2333             WHEN DN_VARIANT_PART =>
2334                 VERIFICA_VARIANT_PART(PCOM,PNO) ;
2335             WHEN OTHERS =>      -- DN_NULL_COM=
2336                 NULL ;
2337             END CASE ;
2338             PCOM := NEXT(PCOM) ;
2339     END LOOP ;
2340     TORNA_IDENTIFICADOR_VISIVEL(PNO) ;      -- MODIFICA ATRIBUTO ACESSO

```

```

2341 END VERIFICA_RECORD ;
2342 -----
2343 ---
2344 --- RECORD_REF => AS_NAME : NAME ;
2345 --- AS_ALIGNMENT : ALIGNMENT ;
2346 --- AS_COMP_REFS : COMP_REFS ;
2347 ---
2348 PROCEDURE VERIFICA_RECORD_REF(PNO : PT_TREE) IS
2349 FN,PAL,PCRS,POBJ,PREC: PT_TREE ;
2350 BEGIN
2351 FN := AS_NAME(PNO) ;
2352 PREC := APONTA_VOID ;
2353 PAL := AS_ALIGNMENT(PNO) ;
2354 PCRS := AS_COMP_REFS(PNO) ;
2355 RESOLVE_NOME_E_EXPRESSAO(FN) ;
2356 VERIFICA_NAME(FN) ;
2357 POBJ := APANHA_OBJETO(FN) ;
2358 IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID THEN
2359 MSG("ESPERADO DESIGNADOR DE TIPO DE RECORD",POBJ) ;
2360 ELSE
2361 CASE APANHA_NOME_NO(SM_DEFN(POBJ)) IS
2362 WHEN DN_RECORD =>
2363 PREC := SM_DEFN(POBJ) ;
2364 WHEN DN_DERIVED =>
2365 IF SM_TYPE_STRUCT(SM_DEFN(POBJ)) /= DN_RECORD THEN
2366 MSG("ESPERADO DESIGNADOR DE TIPO DE RECORD",POBJ) ;
2367 ELSE
2368 PREC := SM_TYPE_STRUCT(SM_DEFN(POBJ)) ;
2369 END IF ;
2370 WHEN OTHERS =>
2371 MSG("ESPERADO DESIGNADOR DE TIPO DE RECORD",POBJ) ;
2372 END CASE ;
2373 END IF ;
2374 IF APANHA_NOME_NO(PREC) /= DN_VOID THEN
2375 IF SM_RECORD_SPEC(PREC) /= DN_VOID THEN
2376 MSG("TIPO JA RECEDEU CLAUSULA DE REPRESENTACAO",POBJ) ;
2377 ELSIF NOT MESMA PARTE DECLARATIVA(PNO,PREC) THEN
2378 MSG("DEFINICAO OCORREU EM OUTRA PARTE DECLARATIVA",PNO) ;
2379 ELSIF JA_USADO(PREC) THEN
2380 MSG("TIPO JA USADO-CLAUSULA INVALIDA",PNO) ;
2381 ELSE
2382 VERIFICA_ALIGNMENT(PAL,PREC) ;
2383 VERIFICA_COMP_REFS(PCRS,PREC) ;
2384 SM_RECORD_SPEC(PREC) := PNO ;
2385 END IF ;
2386 END IF ;
2387 END VERIFICA_RECORD_REF ;
2388 -----
2389 ---
2390 --- UMA CLAUSULA DE REPRESENTACAO DE TIPO E VALIDA SOMENTE PARA UM
2391 --- TIPO, NAO PARA SUBTIPOS (ARM-SC.13.1-3) NO CASO A CLAUSULA SO E
2392 --- VALIDA RECORDS OU TIPOS DERIVADOS DE RECORDS.
2393 --- NO MAXIMO UMA CLAUSULA DE REPRESENTACAO DE RECORD E PERMITIDA PARA
2394 --- O MESMO TIPO(ARM-SC.13.1-3) O ATRIBUTO 'SM_RECORD_SPEC' DO TIPO
2395 --- QUE ESTA SENDO ESPECIFICADO E USADO PARA ESTE TESTE.
2396 --- A CLAUSULA DE ESPECIFICACAO E A DECLARACAO DE TIPO CORRESPONDENTE
2397 --- DEVEM OCORRER NA MESMA PARTE DECLARATIVA (ARM-SC.13.1-5) ESSE
2398 --- FATO E TESTADO PELA FUNCAO 'MESMA PARTE DECLARATIVA'.
2399 --- NAO E PERMITIDO NENHUM USO DO TIPO ANTES DA CLAUSULA DE
2400 --- REPRESENTACAO CORRESPONDENTE, O QUE E TESTADO PELA FUNCAO

```

```

2401 --- 'JA_USADD'.
2402 ---
2403 PROCEDURE VERIFICA_RENAME(PNO , PPAT : PT_TREE) IS
2404 PN , POBJ , PID , PPROC , PROC : PT_TREE ?
2405 BEGIN
2406   PN := AS_NAME(PNO) ?
2407   RESOLVE_NOME_E_EXPRESSAO(PN) ?
2408   VERIFICA_NAME(PN) ?
2409   POBJ := APANHA_OBJETO(PNO) ?
2410   IF DECLARADO(POBJ) THEN
2411     CASE APANHA_NOME_NO(PPAT) IS
2412     WHEN DN_CONSTANT | DN_VAR =>
2413       PID := AS_LIST(AS_IDS(PPAT)) ?
2414       IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID THEN
2415         MSG("ESPERADO IDENTIFICADOR DE OBJETO",POBJ) ?
2416       ELSE
2417         IF APANHA_TIPO_BASE(PID) /= APANHA_TIPO_BASE(PN) THEN
2418           MSG("TIPOS INCOMPATIVELS EM CLAUSULA RENAME",POBJ) ?
2419         ELSIF APANHA_NOME_NO(PID) = DN_CONST_ID THEN
2420           SM_FIRST(PID) := PID ?
2421         ELSE
2422           IF SUBCOMPONENTE(POBJ) THEN
2423             RENAME_SUBCOMPONENTE(POBJ) ?
2424           END IF ?
2425           IF PARAMETRO_FORMAL(POBJ) THEN
2426             RENAME_PARAMETRO(POBJ) ?
2427           END IF ?
2428           SM_ADDRESS(PID) := APONTA_VOID ?
2429           SM_OBJ_TYPE(PID) := AS_TYPE_SPEC(PPAT) ?
2430           SM_OBJ_DEF(PID) := SM_OBJ_DEF(SM_DEFN(POBJ)) ?
2431         END IF ?
2432       END IF ?
2433     WHEN DN_EXCEPTION =>
2434       IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
2435         APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_EXCEPTION_ID THEN
2436         MSG("ESPERADO IDENTIFICADOR DE EXCECAO",POBJ) ?
2437       ELSE
2438         PID := AS_LIST(AS_IDS(PPAT)) ?
2439         SM_EXCEPTION_DEF(PID) := PNO ?
2440       END IF ?
2441     WHEN DN_PACKAGE_DECL =>
2442       IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
2443         APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_PACKAGE_ID THEN
2444         MSG("ESPERADO IDENTIFICADOR DE PACKAGE",POBJ) ?
2445       ELSE
2446         PID := AS_ID(PPAT) ?
2447         SM_BODY(PID) := PNO ?
2448         SM_SPEC(PID) := SM_SPEC(SM_DEFN(POBJ)) ?
2449         SM_ADDRESS(PID) := SM_ADDRESS(SM_DEFN(POBJ)) ?
2450         SM_FIRST(PID) := PID ?
2451       END IF ?
2452     WHEN DN_SUBPROGRAM_DECL =>
2453       IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
2454         RENAMECAD_SUBPROGRAMA_INVALIDA(POBJ) THEN
2455         MSG("ESPERADO IDENTIFICADOR DE PROCEDIMENTO",POBJ) ?
2456       ELSE
2457         PPROC := COMPARA_HEADER(PPAT , POBJ) ?
2458         IF APANHA_NOME_NO(PPAT) = DN_VOID THEN
2459           MSG("NAO EXISTE PROCEDIMENTO COM HEADER COMPATIVEL")
2460         ELSE

```



```

2461 CASE APANHA_NOME_NO(SM_DEFN(POBJ)) IS
2462 WHEN DN_PROC_ID =>
2463     IF APANHA_NOME_NO(AS_DESIGNATOR(POBJ)) /= DN_PROC_ID
2464     MSG("ESPERADOR IDENTIFICADOR DE PROCEDURE",POBJ)
2465     ELSE
2466     TESTA_RENAME_PROCEDURE(POBJ)
2467     END IF
2468 WHEN DN_FUNCTION_ID =>
2469     IF APANHA_NOME_NO(AS_DESIGNATOR(PPAT)) = DN_DEF_OF TI
2470     TESTA_RENAME_OP(POBJ)
2471     ELSIF APANHA_NOME_NO(PDESIG) = DN_FUNCTION_ID THEN
2472     TESTA_RENAME_FUNCTION(POBJ)
2473     END IF
2474 WHEN DN_ENUM_ID =>
2475     IF APANHA_NOME_NO(PDESIG) /= DN_FUNCTION_ID THEN
2476     MSG("ESPERADO IDENTIFICADOR DE FUNCAO",POBJ)
2477     ELSE
2478     TESTA_RENAME_ENUM_LITERAL(POBJ)
2479     END IF
2480 WHEN DN_ENTRY_ID =>
2481     IF APANHA_NOME_NO(PDESIG) /= DN_PROC_ID THEN
2482     MSG("ESPERADO IDENTIFICADOR DE PROCEDIMENTO",POBJ)
2483     ELSE
2484     TESTA_RENAME_ENTRY(POBJ)
2485     END IF
2486     END CASE
2487     END IF
2488     END IF
2489     END CASE
2490     END IF
2491     END VERIFICA_RENAME
2492     -----
2493     ---
2494     --- O MANUAL COLOCA ALGUMAS RESTRICOES A RENOMEACAO DE SUBCOMPONENTES
2495     --- E PARAMETROS FORMAIS (ARM-SC.8.5-8) AS ROTINAS 'SUBCOMPONENTE' E
2496     --- 'PARAMETRO_FORMAL' VERIFICAM ESSAS RESTRICOES.
2497     --- A SEMANTICA PARA RENOMEACAO DE SUB_PROGRAMAS SE ENCONTRA DEFINIDA
2498     --- NO MANUAL, SECOES 8.5-8, 8.5-9.
2499     ---
2500     --- RETURN => AS_EXP_VOID => EXP_VOID
2501     ---
2502     PROCEDURE VERIFICA_RETURN(PNO : PT_TREE) IS
2503     PEXP : PT_TREE
2504     BEGIN
2505     PEXP := AS_EXP_VOID(PNO)
2506     IF NOT CONTEXTO_RETURN(PNO) THEN
2507     MSG("CONTEXTO INVALIDO PARA COMANDO RETURN",PNO)
2508     ELSIF APANHA_NOME_NO(PEXP) = DN_VOID THEN
2509     IF NOT ACCEPT_PROCEDURE(PNO) THEN
2510     MSG("ESPERADO EM COMANDO ACCEPT OU CORDE DE ROTINA",PNO)
2511     END IF
2512     ELSE
2513     IF NOT FUNCTION(PNO) THEN
2514     MSG("ESPERADO EM FUNCTION",PNO)
2515     ELSE
2516     RESOLVE_NOME_E_EXPRESSAO(PEXP)
2517     VERIFICA_EXP(PEXP)
2518     IF NOT TIPO_VALIDO_PARA_FUNCAO(PEXP) THEN
2519     MSG("TIPO INVALIDO PARA FUNCAO",PEXP)
2520     END IF

```

```

2521     END IF ;
2522 END IF ;
2523 END VERIFICA_RETURN ;
2524 -----
2525 ---
2526 --- ESSE COMANDO SO E PERMITIDO NO CORPO DE UM SUBPROGRAMA (GENERTICO
2527 --- OU NAO) OU EM UM COMANDO 'ACCEPT', MAS NAO EM UMA TASK OU PACOTE
2528 --- ENGLOBADO POR TAI CONSTRUICOES (ARM-SC.5.8-3) A FUNCAO
2529 --- 'CONTEXTO_RETURN' VERIFICA ESSAS CONDICICOES.
2530 --- UM COMANDO 'RETURN' PARA UM COMANDO 'ACCEPT' OU PROCEDURE NAO DEVE
2531 --- CONTER UMA EXPRESSAO; PARA UMA FUNCAO DEVE CONTER UMA EXPRESSAO
2532 --- QUE TENHA O TIPO DA FUNCAO (ARM-SC.5.8-4,5.8-5) ;
2533 ---
2534 --- SELECT => AS_SELECT_CLAUSE_S : SELECT_CLAUSE_S ;
2535 ---           AS_STM_S : STM_S ; -- ELSE
2536 ---
2537 --- SELECT_CLAUSE => AS_EXP_VOID : EXP_VOID ;
2538 ---           AS_STM_S : STM_S ;
2539 ---
2540 --- SELECT_CLAUSE_S => AS_LIST : SEQ OF SELECT_CLAUSE ;
2541 ---
2542 PROCEDURE VERIFICA_SELECT(PNO : PT_TREE) IS
2543 PSCS , PELSE , PSC , PSTM : PT_TREE ;
2544 ALT_ACCEPT : BOOLEAN := FALSE ;
2545 TYPE OPCOES IS (TERMINATE_OPC, DELAY_OPC, ELSE_OPC) ;
2546 OPT : OPCOES ;
2547 BEGIN
2548   PSCS := AS_SELECT_CLAUSE_S(PNO) ;
2549   PELSE := AS_STM(PNO) ;
2550   PSC := AS_LIST(PSCS) ;
2551   WHILE PSC /= NIL
2552   LOOP
2553     PSTMS := AS_STM_S(PSC) ;
2554     PEXP := AS_EXP_VOID(PSC) ;
2555     IF AFANHA_NOME_NO(PEXP) /= NIL THEN
2556       RESOLVE_NOME_E_EXPRESSAO(PEXP) ;
2557       VERIFICA_EXP(PEXP) ;
2558       IF NOT TIPO_BOOLEANO(PEXP) THEN
2559         MSG("ESPERADA EXPRESSAO BOOLEANA">PEXP) ;
2560       END IF ;
2561     END IF ;
2562     PSTM := AS_LIST(PSTMS) ;
2563     CASE AFANHA_NOME_NO(PSTM) IS
2564     WHEN DN_ACCEPT =>
2565       ALT_ACCEPT := TRUE ;
2566     WHEN DN_TERMINATE ...>
2567       IF OPT = TERMINATE_OPC THEN
2568         MSG("NAO ERA ESPERADA ALTERNATIVA TERMINATE">PSTM) ;
2569       ELSIF OPT = DELAY_OPC THEN
2570         MSG("NAO ERA ESPERADA ALTERNATIVA TERMINATE">PSTM) ;
2571       ELSE
2572         OPT := TERMINATE_OPC ;
2573       END IF ;
2574     WHEN DN_DELAY =>
2575       IF OPT = TERMINATE_OPC THEN
2576         MSG("NAO ERA ESPERADA ALTERNATIVA DELAY">PSTM) ;
2577       ELSE
2578         OPT := TERMINATE_OPC ;
2579       END IF ;
2580     WHEN OTHERS => NULL

```

```

2591     END CASE ;
2592     WHILE PSTM /= NIL
2593     LOOP
2594         VERIFICA_STM(PSTM) ;
2595         PSTM := NEXT(PSTM) ;
2596     END LOOP ;
2597     PSC := NEXT(PSC) ;
2598 END LOOP ;
2599 IF NOT ALT_ACCEPT THEN
2600     MSG("ERA ESPERADA PELO MENOS UMA ALTERNATIVA ACCEPT",PNO) ;
2601 END IF ;
2602 IF APANHA_NOME_NO(PELSE) /= DN_VOID THEN
2603     IF TERMINATE_OPC OR DELAY_OPC THEN
2604         MSG("NAO ERA ESPERADA ALTERNATIVA ELSE",PELSE) ;
2605     END IF ;
2606     WHILE PELSE /= NIL
2607     LOOP
2608         VERIFICA_STM(PELSE) ;
2609         PELSE := NEXT(PELSE) ;
2610     END LOOP ;
2611 END IF ;
2612 END VERIFICA_SELECT ;
2613 -----
2614 ---
2615 --- UMA ESPERA SELETIVA DEVE CONTER PELO MENOS PELO MENOS UMA
2616 --- ALTERNATIVA 'ACCEPT'. POR OUTRO LADO, PODE CONTER UMA E SO UMA
2617 --- ALTERNATIVA 'TERMINATE', UMA OU MAIS ALTERNATIVAS 'DELAY' OU
2618 --- UMA PARTE 'ELSE' ; ESSAS 3 POSSIBILIDADES SAO MUTUAMENTE
2619 --- EXCLUSIVAS.
2620 ---
2621 ---
2622 --- TASK_DECL => AS_ID : ID ,      -- VAR_ID
2623 ---                AS_TASK_DEF : TASK_DEF ;
2624 ---
2625 --- TASK_SPEC => AS_DECL_S : DECL_S ,
2626 ---                SM_BODY : BODY_STUB_VOID ,
2627 ---                SM_ADDRESS : EXP_VOID ,
2628 ---                SM_STORAGE_SIZE : EXP_VOID ;
2629 ---
2630 PROCEDURE VERIFICA_TASK_DECL(PNO : PT_TREE) IS
2631     FID , FTK_SPEC : PT_TREE ;
2632 BEGIN
2633     PID := AS_ID(PNO) ;
2634     PSTK_SPEC := AS_TASK_DEF(PNO) ;
2635     IF DECLARADO(FID) THEN
2636         MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PID) ;
2637     ELSE
2638         SM_OBJ_TYPE(PID) := FTK_SPEC ;
2639         SM_OBJ_DEF(PID) := APONTA_VOID ;
2640         SM_ADDRESS(PID) := APONTA_VOID ;
2641         VERIFICA_TASK_SPEC(PSTK_SPEC) ;
2642     END IF ;
2643 END VERIFICA_TASK_DECL ;
2644 -----
2645 PROCEDURE VERIFICA_TASK_SPEC(PNO : PT_TREE) IS
2646     PDCLS , PDCL : PT_TREE ;
2647 BEGIN
2648     PDCLS := AS_DECL_S(PNO) ;
2649     PDCL := AS_LIST(PDCLS) ;

```

```

2641 WHILE PDCL /= NIL
2642 LOOP
2643     CASE APANHA_NOME_NO(PDCL) IS
2644         WHEN SUBPROGRAM_DECL =>
2645             VERIFICA_SUBPROGRAM_DECL(PDCL) ¶
2646         WHEN DN_SIMPLE_REF =>
2647             VERIFICA_SIMPLE_REF(PDCL) ¶
2648         WHEN DN_ADDRESS =>
2649             VERIFICA_ADDRESS(PDCL) ¶
2650         WHEN DN_RECORD_REF =>
2651             VERIFICA_RECORD_REF(PDCL) ¶
2652     END CASE ¶
2653     PDCL := NEXT(PDCL) ¶
2654 END LOOP ¶
2655 END VERIFICA_TASK_SPEC ¶
2656 -----
2657 ---
2658 --- TASK_BODY => AS_ID : ID ,      --- TASK_BODY_ID
2659 --- AS_BLOCK_STUB : BLOCK_STUB ¶
2660 ---
2661 --- TASK_BODY_ID => SM_TYPE_SPEC : TYPE_SPEC ,
2662 --- SM_BODY : BLOCK_STUB_VOID ,
2663 --- SM_FIRST : DEF_OCCURRENCE ,
2664 --- SM_STUB : DEF_OCCURRENCE ¶
2665 ---
2666 PROCEDURE VERIFICA_TASK_BODY(PNO : PT_TREE) IS
2667 PTID , PBLK , PSPEC : PT_TREE ¶
2668 BEGIN
2669     PTID := AS_ID(PNO) ¶
2670     PBLK := AS_BLOCK_STUB(PNO) ¶
2671     IF NOT DECLARADO(PTID) THEN
2672         MSG("NAO EXISTE ESPECIFICACAO PARA TASK",PTID) ¶
2673     ELSE
2674         PDEF := APANHA_DEFINICAO(PTID) ¶
2675         IF APANHA_NOME_NO(PDEF) = DN_VAR_ID THEN
2676             IF APANHA_NOME_NO(SM_OBJ_TYPE(PDEF)) /= DN_TASK_SPEC THEN
2677                 MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PTID) ¶
2678             ELSE
2679                 PSPEC := SM_OBJ_TYPE(PDEF) ¶      --- DN_TASK_SPEC
2680                 SM_BODY(PSPEC) := PBLK ¶
2681                 SM_TYPE_SPEC(PTID) := PSPEC ¶
2682                 SM_FIRST(PTID) := PDEF ¶
2683                 IF APANHA_NOME_NO(PBLK) = DN_BLOCK THEN
2684                     SM_BODY(PTID) := PBLK ¶
2685                     SM_STUB(PTID) := APONTA_VOID ¶
2686                 ELSE
2687                     SM_BODY(PTID) := APONTA_VOID ¶
2688                     SM_STUB(PTID) := PBLK ¶
2689                 END IF ¶
2690             END IF ¶
2691         ELSIF APANHA_NOME_NO(PDEF) = DN_TYPE_ID THEN
2692             IF APANHA_NOME_NO(SM_TYPE_SPEC(PDEF)) /= DN_TASK_SPEC THEN
2693                 MSG("IDENTIFICADOR JA USADO NO MESMO ESCOPO",PTID) ¶
2694             ELSE
2695                 PSPEC := SM_TYPE_SPEC(PDEF) ¶
2696                 SM_BODY(PSPEC) := PBLK ¶
2697                 SM_TYPE_SPEC(PTID) := PSPEC ¶
2698                 SM_FIRST(PTID) := PDEF ¶
2699                 IF APANHA_NOME_NO(PBLK) = DN_BLOCK THEN
2700                     SM_BODY(PTID) := PBLK ¶

```

```

2701         SM_STUB(PTID) := APONTA_VOID ;
2702     ELSE
2703         SM_BODY(PTID) := APONTA_VOID ;
2704         SM_STUB(PTID) := PBLK ;
2705     END IF ;
2706 END IF ;
2707 ELSE
2708     MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PTID) ;
2709 END IF ;
2710 END IF ;
2711 END VERIFICA_TASK_BODY ;
2712 -----
2713 ---
2714 --- TIMED_ENTRY => AS_STM_S1 : STM_S ,
2715 ---             AS_STM_S2 : STM_S ;
2716 ---
2717 PROCEDURE VERIFICA_TIMED_ENTRY(PNO : PT_TREE) IS
2718 P$1 , P$2 : PT_TREE ;
2719 BEGIN
2720     P$1 := AS_STM_S1(PNO) ;
2721     P$2 := AS_STM_S2(PNO) ;
2722     VERIFICA_STM_S(P$1) ;
2723     VERIFICA_STM_S(P$2) ;
2724 END VERIFICA_TIMED_ENTRY ;
2725 -----
2726 ---
2727 --- TYPE => AS_ID : ID ,      --- TYPE_ID
2728 ---             AS_DSCRNT_VAR_S : DSCRNT_VAR_S ,
2729 ---             AS_TYPE_SPEC : TYPE_SPEC ;
2730 ---
2731 --- TYPE_ID => SM_TYPE_SPEC : TYPE_SPEC ,
2732 ---             SM_FIRST : DEF_OCCURENCE ;
2733 ---
2734 PROCEDURE VERIFICA_TYPE(PNO : PT_TREE) IS
2735 PID,P$VS,P$SPEC,P$DEF,P$DSC,P$FIRST:PT_TREE ;
2736 BEGIN
2737     PID := AS_ID(PNO) ;
2738     P$FIRST := PID ;
2739     P$VS := AS_DSCRNT_VAR_S(PNO) ;
2740     P$DSC := AS_LIST(P$VS) ;
2741     P$SPEC := AS_TYPE_SPEC(PNO) ;
2742     IF DECLARADO(PID) THEN
2743         P$DEF := APANHA_DEFINICAO(PID) ;
2744         IF APANHA_NOME_NO(P$DEF) = DN_TYPE_ID AND THEN
2745             APANHA_NOME_NO(SM_TYPE_SPEC(P$DEF)) = DN_VOID THEN
2746                 SM_TYPE_SPEC(P$DEF) := P$SPEC ;
2747                 P$FIRST := P$DEF ;
2748                 COMPARA_DISCRIMINANTES(PID,P$DEF) ;
2749                 IF APANHA_NOME_NO(P$SPEC) /= DN_VOID OR ELSE
2750                     APANHA_NOME_NO(P$SPEC) /= DN_PRIVATE OR ELSE
2751                     APANHA_NOME_NO(P$SPEC) /= DN_LL_PRIVATE THEN
2752                     MSG("ESPERADA DECLARACAO DE TIPO COMPLETA",P$SPEC) ;
2753                     RETURN ;
2754                 END IF ;
2755             ELSE
2756                 MSG("IDENTIFICADOR JA USADO NO MESMO ESCOPO",PID) ;
2757                 RETURN ;
2758             END IF ;
2759         END IF ;
2760     SM_FIRST(PID) := P$FIRST ;

```

```

2761 SM_TYPE_SPEC(PID) := PTSPEC ;
2762 IF PUSC /= NIL THEN
2763   IF APANHA_NOME_NO(PTSPEC) /= DN_RECORD OR
2764     APANHA_NOME_NO(PTSPEC) /= DN_PRIVATE OR
2765     APANHA_NOME_NO(PTSPEC) /= DN_VOID THEN
2766     MSG("NAO ERA ESPERADO DISCRIMINANTE">PUSC) ;
2767   ELSE
2768     VERIFICA_DSCRMT_VAR_S(PUSC) ;
2769   END IF ;
2770 END IF ;
2771 TORNA_INVISIVEL(PID) ;
2772 VERIFICA_TYPE_SPEC(PTSPEC) ;
2773 TORNA_VISIVEL(PID) ;
2774 END VERIFICA_TYPE ;
2775 -----
2776 ---
2777 --- SEQ IDENTIFICADOR DO TIPO JA TIVER SIDO USADO EM OUTRA DECLARACAO
2778 --- DEVE-SE SE SUA 1. DECLARACAO E UMA DECLARACAO DE TIPO INCOMPLETA?
2779 --- VER MANUAL DE DIANA (DM-SC.3.5.1.1) PARA A ESTRUTURA DE
2780 --- DECLARACAO DE TIPO INCOMPLETA.
2781 --- UMA PARTE DISCRIMINANTE DEVE SER FORNECIDA EM UMA DECLARACAO DE
2782 --- TIPO COMPLETAMENTE SE FOI FORNECIDA NA DECLARACAO INCOMPLETA
2783 --- CORRESPONDENTE? NESSE CASO, OS DISCRIMINANTES DEVEM SER
2784 --- COMPATIVELIS(ARM-SC.3.8.1-4)? A ROTINA 'COMPARA_DISCRIMINANTES'
2785 --- REALIZA ESSE TESTE.
2786 --- UMA DECLARACAO DE TIPO INCOMPLETA NAO PODE REPRESENTAR UMA
2787 --- DECLARACAO DE TIPO PRIVADO.
2788 --- UMA PARTE DISCRIMINANTE SO PODE SER FORNECIDA PARA UMA DECLARACAO
2789 --- DE RECORD OU DE TIPO PRIVADO.
2790 --- UM DADO TIPO NAO PODE TER UMA SUBCOMPONENTE CUJO SEJA O PROPRIO
2791 --- TIPO DADO(ARM-SC.3.3-8)? ISSO E OBTIDO PELAS ROTINAS
2792 --- 'TORNA_INVISIVEL' E 'TORNA_VISIVEL' .
2793 ---
2794 --- USE => AS_LIST : SEQ OF NAME ;
2795 ---
2796 PROCEDURE VERIFICA_USE(PNO : PT_TREE) IS
2797 PN , POBJ : PT_TREE ;
2798 BEGIN
2799   PN := AS_LIST(PNO) ;
2800   WHILE PN /= NIL
2801     LOOP
2802       RESOLVE_NOME_E_EXPRESSAO(PN) ;
2803       VERIFICA_NAME(PN) ;
2804       POBJ := APANHA_OBJETO(PN) ;
2805       IF APANHA_NOME_NO(POBJ) /= DN_USED_NAME_ID OR ELSE
2806         APANHA_NOME_NO(SM_DEFN(POBJ)) /= DN_PACKAGE_ID THEN
2807         MSG("ESPERADO IDENTIFICADOR DE PACKAGE">POBJ) ;
2808       ELSIF NOT EK_CLAUSULA_WITH(POBJ) THEN
2809         MSG("IDENTIFICADOR NAO ESPECIFICADO EM CLAUSULA WITH">POBJ) ;
2810       ELSE
2811         TORNA_DECLARACAOES_VISIVEIS(POBJ) ;
2812       END IF ;
2813       PN := NEXT(PN) ;
2814     END LOOP ;
2815 END VERIFICA_USE ;
2816 -----
2817 ---
2818 --- OS NOMES QUE APARECEM NA CLAUSULA DEVEM SER NOMES DE PACOTES
2819 --- (ARM-SC.8.4-1).
2820 --- ESSES NOMES JA DEVEM TER SIDO MENCIONADOS EM CLAUSULAS 'WITH'

```

```

2821 --- ANTERIORES (ARM-SC.10.1.1-3).
2822 --- A SECÃO 8.4 DO MANUAL DE AIA ESPECIFICA QUAIS AS DECLARAÇÕES
2823 --- QUE SE TORNAM VISÍVEIS PELA CLAUSULA 'USE' E A ROTINA
2824 --- 'TORNA_DECLARAÇÕES_VISÍVEIS' IMPLEMENTA ESSA VISIBILIDADE.
2825 ---
2826 --- USED_BLTN_ID => SM_OPERATOR : OPERATOR ;
2827 ---
2828 --- USED_BLTN_OP => SM_OPERATOR : OPERATOR ;
2829 ---
2830 PROCEDURE VERIFICA_USED_BLTN_ID(PNO : PT_TREE) IS
2831   IF BUILD_IN(PNO) THEN
2832     SM_OPERATOR(PNO) := APANHA_DEFINICAO(PNO) ;
2833   END IF ;
2834 ---
2835 PROCEDURE VERIFICA_USED_BLTN_OP(PNO : PT_TREE) IS
2836   SM_OPERATOR(PNO) := APANHA_DEFINICAO(PNO) ;
2837 ---
2838 --- O ATRIBUTO 'SM_OPERATOR' DESSES NÓS INDICA UMA ENTIDADE
2839 --- 'PRE_CONSTRUIDA' ESSE ATRIBUTO É UM TIPO PRIVADO E
2840 --- DEPENDENTE DE IMPLEMENTAÇÃO.
2841 ---
2842 ---
2843 ---
2844 --- USED_CHAR => SM_DEFN : DEF_OCCURRENCE ,
2845 ---           SM_EXP_TYPE : TYPE_SPEC ,
2846 ---           SM_VALUE : VALUE ;
2847 ---
2848 PROCEDURE VERIFICA_USED_CHAR(PNO : PT_TREE; ESTAT : BOOLEAN; FALSE) IS
2849 PELS, PEL : PT_TREE ;
2850 ACHOU : BOOLEAN := FALSE ;
2851 BEGIN
2852   IF SM_EXP_TYPE(PNO) /= TIPO_GERAL THEN
2853     PELS := SM_EXP_TYPE(PNO) ; --- ENUM_LITERAL_S
2854     PEL := AS_LIST(PELS) ;
2855     WHILE NOT ACHOU
2856     LOOP
2857       IF MESMO_LITERAL(PNO, PEL) THEN
2858         ACHOU := TRUE ;
2859         SM_DEFN(PNO) := PEL ;
2860       ELSE
2861         PEL := NEXT(PEL) ;
2862       END IF ;
2863     END LOOP ;
2864   IF ESTAT THEN
2865     MARCA_LITERAL_ESTATICO(PNO) ;
2866   END IF ;
2867 END IF ;
2868 END VERIFICA_USED_CHAR ;
2869 -----
2870 ---
2871 --- A ROTINA 'RESOLVE_NOME_E_EXPRESSAO' JÁ FOI CHAMADA PARA ANALISAR
2872 --- ESSE NÓ DESSA FORMA, SE O TIPO DO MESMO ESTIVER DEFINIDO (/=
2873 --- TIPO_GERAL), SEU ATRIBUTO 'SM_EXP_TYPE' APONTA PARA UMA SUB-ÁRVORE
2874 --- DO TIPO 'ENUM_LITERAL_S' QUE CONTEM A DEFINIÇÃO DO LITERAL DE
2875 --- ENUMERAÇÃO EM QUESTÃO (NOTE QUE ISSO OCORRE DEPOIS DA RESOLUÇÃO DE
2876 --- OVERLOADING).
2877 ---
2878 --- USED_NAME_ID => SM_DEFN : DEF_OCCURRENCE ;
2879 ---
2880 --- USED_OBJECT_ID => SM_EXP_TYPE : TYPE_SPEC ;

```

```

2881 ---          SM_DEFN : DEF_OCCURRENCE ;
2882 ---          SM_VALUE : VALUE ?
2883 ---
2884 PROCEDURE VERIFICA_USED_NAME_ID(PNO : PT_TREE) IS
2885     NULL
2886 ---
2887 PROCEDURE VERIFICA_USED_OBJECT_ID(PNO : PT_TREE) IS
2888     IF ESTAT THEN
2889         MARCA_VALOR_OBJETO(PNO) ?
2890     END IF ?
2891 ---
2892 ---  AS ROTINAS 'TRANSFORMA_USED_OBJECT_ID' E 'TRANSFORMA_USED_NAME_ID'
2893 ---  CHAMADAS NA FASE DE RESOLUCAO DE NOME E EXPRESSAO, SAO
2894 ---  PELO PREENCHIMENTO DOS ATRIBUTOS SEMANTICOS DOS NÓS? DESSA FORMA
2895 ---  SO E NECESSARIO SE VERIFICAR SE E ESPERADO UM VALOR ESTATICO PARA
2896 ---  UM NÓ DO TIPO 'USED_OBJECT_ID'.
2897 ---
2898 ---  VAR => AS_IDS : IDS ;
2899 ---          AS_TYPE_SPEC : TYPE_SPEC ;
2900 ---          AS_OBJECT_TYPE : OBJECT_DEF ?
2901 ---
2902 ---  VAR_ID => SM_OBJ_TYPE : TYPE_SPEC ;
2903 ---          SM_ADDRESS : EXP_VOID ;
2904 ---          SM_OBJ_DEF : OBJECT_DEF ?
2905 ---
2906 PROCEDURE VERIFICA_VAR(PNO : PT_TREE) IS
2907 PIDS , PID , PISPEC , PODEF : PT_TREE ?
2908 BEGIN
2909     PIDS := AS_IDS(PNO) ?
2910     PISPEC := AS_TYPE_SPEC(PNO) ?
2911     PODEF := AS_OBJECT_DEF(PNO) ?
2912     PID := AS_LIST(PIDS) ?
2913     WHILE PID /= NIL
2914     LOOP
2915         IF DECLARADO(PID) THEN
2916             MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO" , PID) ?
2917         ELSE
2918             SM_OBJ_TYPE(PID) := PISPEC ?
2919             SM_ADDRESS(PID) := APONTA_VOID ?
2920             SM_OBJ_DEF(PID) := PODEF ?
2921         END IF ?
2922         PID := NEXT(PID) ?
2923     END LOOP ?
2924     IF APANHA_NOME_NO(PISPEC) = DN_CONSTRAINT THEN
2925         VERIFICA_CONSTRAINED(PISPEC) ?
2926         IF NOT TIPO_DECLARADO(PISPEC) THEN
2927             MSG("IDENTIFICADOR DE TIPO NAO DECLARADO" , PISPEC) ?
2928         END IF ?
2929     ELSE --- DN_ARRAY
2930         VERIFICA_ARRAY(PISPEC) ?
2931     END IF ?
2932     IF APANHA_NOME_NO(PODEF) /= DN_VOID THEN
2933         RESOLVE_NOME_E_EXPRESSAO(PODEF) ?
2934         VERIFICA_EXP(PODEF) ?
2935     END IF ?
2936     TORNA_VISIVEL(PID) ?
2937 END VERIFICA_VAR ?
2938 -----
2939 ---
2940 ---  A ROTINA 'VERIFICA_CONSTRAINED' NAO TESTA SE O IDENTIFICADOR QUE

```



```

2941 --- APARECER NA INDICACAO DE SUBTIPO JA FOI DECLARADO OU NAO? ESSE
2942 --- TESTE E FEITO PELA ROTINA 'TIPO_DECLARADO'.
2943 --- A INDICACAO DE SUBTIPO E USADA PELA ROTINA DE RESOLUCAO DE
2944 --- EXPRESSOES; DESSA FORMA NAO PRECISO TESTAR SE O TIPO DA EXPRESSAO
2945 --- INICIAL ESTA DE ACORDO COM O SUBTIPO INDICADO.
2946 ---
2947 --- VARIANT_PART => AS_NAME : NAME ;
2948 --- AS_VARIANT_S : VARIANT_S ;
2949 ---
2950 --- VARIANT_S => AS_LIST : SEQ OF VARIANT ;
2951 ---
2952 --- VARIANT => AS_CHOICE_S : CHOICE_S ;
2953 ---
2954 PROCEDURE VERIFICA_VARIANT_PART(PNO , PREC : PT_TREE) IS
2955 PN,PVS,PV,PCS,PIN_REC,PC : PT_TREE ;
2956 BEGIN
2957 PN := AS_NAME(PNO) ;
2958 PVS := AS_VARIANT_S(PNO) ;
2959 RESOLVE_NOME_E_EXPRESSAO(PN) ;
2960 VERIFICA_EXP(PN) ;
2961 IF NOT DISCRIMINANTE_VALIDO(PN,PREC) THEN
2962 MSG("NOME INVALIDO PARA DISCRIMINANTE",PN) ;
2963 ELSE
2964 IF ESTRUTURA_VARIANT_S(PVS) THEN
2965 PV := AS_LIST(PVS) ;
2966 WHILE PV /= NIL
2967 LOOP
2968 PCS := AS_CHOICE_S(PV) ;
2969 PIN_REC := AS_RECORD(PV) ;
2970 PC := AS_LIST(PCS) ;
2971 WHILE PC /= NIL
2972 LOOP
2973 CASE APANHA_NOME_NO(PC) IS
2974 WHEN DN_RANGE =>
2975 VERIFICA_RANGE(PC) ;
2976 IF NOT TIPOS_COMPATIVELIS(PN,PC) THEN
2977 MSG("TIPO INCOMPATIVEL COM DISCRIMINANTE",PC) ;
2978 ELSIF NOT RANGE_ESTATICO(PC) THEN
2979 MSG("ESPERADO RANGE ESTATICO",PC) ;
2980 END IF ;
2981 WHEN DN_CONSTRAINED =>
2982 VERIFICA_CONSTRAINED(PC) ;
2983 IF NOT TIPO_DECLARADO(PC) THEN
2984 MSG("IDENTIFICADOR DE SUBTIPO NAO DECLARADO",PC) ;
2985 ELSIF NOT TIPOS_COMPATIVELIS(PN,PC) THEN
2986 MSG("SUBTIPO NAO E COMPATIVEL",PC) ;
2987 ELSIF NOT RESTRICAO_ESTATICA(PC) THEN
2988 MSG("ESPERADO RANGE ESTATICO",PC) ;
2989 END IF ;
2990 WHEN DN_OTHERS =>
2991 NULL ;
2992 WHEN OTHERS => -- EXPRESSOES
2993 RESOLVE_NOME_E_EXPRESSAO(PC) ;
2994 VERIFICA_EXP(PC) ;
2995 IF NOT TIPOS_COMPATIVELIS(PN,PC) THEN
2996 MSG("TIPO DA EXPRESSAO INCOMPATIVEL",PC) ;
2997 ELSIF NOT RANGE_ESTATICO(PN,PC) THEN
2998 MSG("ESPERADA EXPRESSAO ESTATICA",PC) ;
2999 ELSIF NOT COMPONENTE(PC,PREC) THEN
3000 MSG("NAO ERA ESPERADA COMPONENTE DE RECORD")

```

```

3001         END IF ;
3002         END CASE ;
3003         PC := NEXT(PC) ;
3004     END LOOP ;
3005     PV := NEXT(PV) ;
3006     END LOOP ;
3007     END IF ;
3008     END IF ;
3009     END VERIFICA_VARIANT_PART ;
3010     -----
3011     ---
3012     --- SIMPLE_REP => AS_NAME : NAME ,
3013     --- AS_EXP : EXP ;
3014     ---
3015     ---
3016     PROCEDURE VERIFICA_SIMPLE_REP(PNO : PT_TREE) IS
3017     PN , PE , PORJ , PID , PTP : PT_TREE ;
3018     BEGIN
3019     PN := AS_NAME(PNO) ;
3020     PE := AS_EXP(PNO) ;
3021     IF APANHA_NOME_NO(PN) = DN_ATTRIBUTE THEN      --- LENGTH CLAUSE
3022     PID := AS_ID(PN) ;      --- ATRIBUTO
3023     PTP := AS_NAME(PN) ;      --- PREFIXO
3024     RESOLVE_NOME_E_EXPRESSAO(PTP) ;
3025     VERIFICA_NAME(PTP) ;
3026     PORJ := APANHA_OBJETO(PN) ;
3027     IF APANHA_NOME_NO(PORJ) /= DN_USED_NAME_ID OR ELSE
3028     APANHA_NOME_NO(SM_DEFN(PORJ)) /= DN_TYPE_ID THEN
3029     MSG("PREFIXO DO ATRIBUTO DEVE DENOTAR TIPO",PORJ) ;
3030     ELSE
3031     RESOLVE_NOME_E_EXPRESSAO(PE) ;
3032     VERIFICA_EXP(PE) ;
3033     CASE APANHA_REPRESENTACAO(PID) IS
3034     WHEN REP_SIZE =>
3035     IF NOT TIPO_INTEIRO(PE) THEN
3036     MSG("ESPERADA EXPRESSAO DE TIPO INTEIRO",PE) ;
3037     ELSIF NOT EXPRESSAO_ESTATICA(PE) THEN
3038     MSG("ESPERADA EXPRESSAO ESTATICA",PE) ;
3039     ELSIF NOT TAMANHO_SUFICIENTE(PE,PORJ) THEN
3040     MSG("TAMANHO INSUFICIENTE PARA OBJETO",PE) ;
3041     ELSIF NOT TIPO_VALIDO_LENGTH_CLAUSE(PORJ) THEN
3042     MSG("TIPO INVALIDO PARA CLAUSULA",PORJ) ;
3043     ELSE
3044     VERIFICA_ATTRIBUTE(PN) ;
3045     END IF ;
3046     WHEN REP_STORAGE_SIZE =>
3047     IF NOT TIPO_INTEIRO(PE) THEN
3048     MSG("ESPARADA EXPRESSAO DE TIPO INTEIRO",PE) ;
3049     ELSIF APANHA_NOME_NO(SM_DEFN(PORJ)) = DN_DERIVED THEN
3050     MSG("NAO ERA ESPERADO TIPO DERIVADO",PORJ) ;
3051     ELSE
3052     VERIFICA_ATTRIBUTE(PN) ;
3053     END IF ;
3054     WHEN REP_SMALL =>
3055     IF NOT TIPO_REAL(PE) THEN
3056     MSG("ESPERADA EXPRESSAO DE TIPO REAL",PE) ;
3057     ELSIF NOT EXPRESSAO_ESTATICA(PE) THEN
3058     MSG("ESPERADA EXPRESSAO ESTATICA",PE) ;
3059     ELSE
3060     VERIFICA_ATTRIBUTE(PN) ;

```

```

3061         END IF ;
3062     WHEN OTHERS =>
3063         MSG("ATRIBUTO INVALIDO PARA CLAUSULA LENGTH",F10) ;
3064     END CASE ;
3065     END IF ;
3066 ELSE    --    ENUMERATION REPRESENTATION CLAUSE
3067     IF NOT TIPO_BASE_ENUMERACAO(POBJ) THEN
3068         MSG("ESPERADO TIPO DE ENUMERACAO",POBJ) ;
3069     ELSE
3070         SM_EXP_TYPE(PE) := SM_TYPE_SPEC(POBJ) ;
3071         VERIFICA_AGGREGATE(PE) ;
3072         IF NOT CODIGOS_DISTINTOS(PE) THEN
3073             MSG("CODIGO REPETIDO PARA VARIOS LITERAIS DE ENUMERACAO") ;
3074         ELSIF NOT CODIGOS_ESTATICOS(PE) THEN
3075             MSG("ESPERADA EXPRESSAO ESTATICA") ;
3076         ELSIF NOT CODIGOS_ORDENADOS(PE) THEN
3077             MSG("CODIGOS DEVEM SATISFAZER ORDENACAO DO TIPO") ;
3078         END IF ;
3079     END IF ;
3080 END IF ;
3081 END VERIFICA_SIMPLE_REP ;
3082 -----
3083 ---
3084 ---  ESSA SUBARVORE PODE REPRESENTAR TANTO UMA CLAUSULA 'LENGTH' QUANDO
3085 ---  UMA CLAUSULA DE REPRESENTACAO DE TIPO DE ENUMERACAO? NO 1. CASO
3086 ---  O ATRIBUTO 'AS_NAME' VAI APONTAR PARA UM NO DO TIPO 'ATTRIBUTE'.
3087 ---
3088 ---  SLICE => AS_NAME : NAME ,
3089 ---           AS_DSCRT_RANGE : DSCRT_RANGE ,
3090 ---           SM_EXP_TYPE : TYPE_SPEC ,
3091 ---           SM_CONSTRAINT : CONSTRAINT ;
3092 ---
3093 PROCEDURE VERIFICA_SLICE(PNO : PT_TREE) IS
3094     PN , PDR : PT_TREE ;
3095 BEGIN
3096     PN := AS_NAME(PNO) ;
3097     PDR := AS_DSCRT_RANGE(PNO) ;
3098     IF SM_EXP_TYPE(PNO) /= TIPO_GERAL THEN
3099         RESOLVE_NOME_E_EXPRESSAO(PN) ;
3100         VERIFICA_NAME(PN) ;
3101         IF APANHA_NOME_NO(PDR) = DN_RANGE THEN
3102             VERIFICA_RANGE(PDR) ;
3103             SM_CONSTRAINT(PNO) := PDR ;
3104         ELSE    --    DN_CONSTRAINT
3105             VERIFICA_CONSTRAINT(PNO) ;
3106             SM_CONSTRAINT(PNO) := SM_CONSTRAINT(PDR) ;
3107         END IF ;
3108     END IF ;
3109 END VERIFICA_SLICE ;
3110 -----
3111 ---
3112 ---  NOTE QUE QUANDO A ROTINA 'VERIFICA_SLICE' E CHAMADA, A ROTINA DE
3113 ---  RESOLUCAO DE 'OVERLOADING' JA REALIZOU UMA GRANDE PARTE DOS TESTES
3114 ---  SEMANTICOS (POR EX., JA FOI VERIFICADO SE O NOME REPRESENTA UM
3115 ---  OBJETO DO TIPO ARRAY v ...).
3116 ---
3117 ---  STM_S => AS_LIST : SEQ IF STM ;
3118 ---
3119 PROCEDURE VERIFICA_STM_S(PNO : PT_TREE) IS
3120     FS : PT_TREE ;

```

```

3121 BEGIN
3122     PS := AS_LIST(PNO) ;
3123     WHILE PS /= NIL
3124     LOOP
3125         CASE APANHA_NOME_NO(PS) IS
3126             WHEN DN_IF =>
3127                 VERIFICA_IF(PS) ;
3128             WHEN DN_CASE =>
3129                 VERIFICA_CASE(PS) ;
3130             WHEN DN_NAMED_STM =>
3131                 VERIFICA_NAMED_STM(PS) ;
3132             WHEN DN_BLOCK =>
3133                 VERIFICA_BLOCK(PS) ;
3134             WHEN DN_ACCEPT =>
3135                 VERIFICA_ACCEPT(PS) ;
3136             WHEN DN_SELECT =>
3137                 VERIFICA_SELECT(PS) ;
3138             WHEN DN_COND_ENTRY =>
3139                 VERIFICA_COND_ENTRY(PS) ;
3140             WHEN DN_TIMED_ENTRY =>
3141                 VERIFICA_TIMED_ENTRY(PS) ;
3142             WHEN DN_LABELLED =>
3143                 VERIFICA_LABELLED(PS) ;
3144             WHEN DN_NULL_STM =>
3145                 NULL ;
3146             WHEN DN_ASSIGN =>
3147                 VERIFICA_ASSIGN(PS) ;
3148             WHEN DN_PROCEDURE_CALL =>
3149                 VERIFICA_PROCEDURE_CALL(PS) ;
3150             WHEN DN_EXIT =>
3151                 VERIFICA_EXIT(PS) ;
3152             WHEN DN_EXIT =>
3153                 VERIFICA_EXIT(PS) ;
3154             WHEN DN_RETURN =>
3155                 VERIFICA_RETURN(PS) ;
3156             WHEN DN_GOTO =>
3157                 VERIFICA_GOTO(PS) ;
3158             WHEN DN_ENTRY_CALL =>
3159                 VERIFICA_ENTRY_CALL(PS) ;
3160             WHEN DN_DELAY =>
3161                 VERIFICA_DELAY(PS) ;
3162             WHEN DN_ABORT =>
3163                 VERIFICA_ABORT(PS) ;
3164             WHEN DN_RAISE =>
3165                 VERIFICA_RAISE(PS) ;
3166             WHEN DN_CODE =>
3167                 VERIFICA_CODE(PS) ;
3168             WHEN DN_PRAGMA =>
3169                 VERIFICA_PRAGMA(PS) ;
3170             WHEN DN_TERMINATE =>
3171                 VERIFICA_TERMINATE(PS) ;
3172         END CASE ;
3173     END LOOP ;
3174 END VERIFICA_STM_S ;
3175
3176 ---
3177 --- STRING_LITERAL => SM_EXP_TYPE : TYPE_SPEC ,
3178 ---                               SM_CONSTRAINT : CONSTRAINT ;
3179 ---
3180 PROCEDURE VERIFICA_STRING_LITERAL(PNO , PPAI : PL_TRFE) IS

```

```

3181  PREST : PT_TREE ;
3182  BEGIN
3183      IF APANHA_NOME_NO(PPAI) = DN.CONSTANT THEN
3184          PREST := CONSTRUI_RESTRICAO(PNO) ;
3185          SM_CONSTRAINT(PNO) := PREST ;
3186      ELSE
3187          SM_CONSTRAINT(PNO) := APONTA_VOID ;
3188      END IF ;
3189  END VERIFICA_STRING_LITERAL ;
3190  -----
3191  ---
3192  --- COMO O MANUAL DE DIANA DEFINE QUE O ATRIBUTO 'SM_VALUE' NAO DEVE
3193  --- SER USADO PARA GUARDAR O VALOR DE UMA EXPRESSAO QUE NAO SEJA
3194  --- ESTATICA(DRM-SC.1.3.1), ELIMINOU-SE ESSE ATRIBUTO DESSE NO.
3195  --- UMA LITERAL DE CARACTERES PODE INTRODUIR UM SUBTIPO ANONIMO,
3196  --- QUE PODE SER USADO PARA RESTRINGIR UM OBJETO SEM A LITERAL
3197  --- APARECER COMO VALOR INICIAL DE UMA CONSTANTE(ARM-SC.3.6.1), NOTE
3198  --- QUE A ESTRUTURA DA RESTRICAO NA APARECE IMPLICITA NA ARVORE
3199  --- SINTATICA ABSTRATA, DE FORMA QUE ELA DEVE SER CRIADA, FUNCAO
3200  --- DA ROTINA 'CONSTRUI_RESTRICAO'.
3201  ---
3202  --- SUB =>      ;
3203  ---
3204  PROCEDURE VERIFICA_STUB(PNO : PT_TREE) IS
3205  NULL ;
3206  ---
3207  --- SUBPROGRAM_BODY => AS_DESIGNATOR : DESIGNATOR ,
3208  ---                   AS_HEADER : HEADER ,
3209  ---                   AS_BLOCK_STUB : BLOCK_STUB ;
3210  ---
3211  --- SUBPROGRAM_DECL => AS_DESIGNATOR : DESIGNATOR ,
3212  ---                   AS_HEADER : HEADER ,
3213  ---                   AS_SUBPROGRAM_DEF : SUBPROGRAM_DEF ;
3214  ---
3215  PROCEDURE VERIFICA_SUBPROGRAM_BODY(PNO : PT_TREE) IS
3216  PDESIG , PHEADER , PBLOCK : PT_TREE ;
3217  BEGIN
3218      PDESIG := AS_DESIGNATOR(PNO) ;    --- PROC_ID , FUNCTION_ID , DEF_OP
3219      PHEADER := AS_HEADER(PNO) ;    --- ENTRY , FUNCTION , PROCEDURE
3220      PBLOCK := AS_BLOCK_STUB(PNO) ;
3221      VERIFICA_DESIGNATOR(PDESIG , PNO) ;
3222      VERIFICA_HEADER(PHEADER , PNO) ;
3223      VERIFICA_BLOCK_STUB(PBLOCK , PNO) ;
3224      IF INSERCAO_CODIGO_INVALIDA THEN
3225          MSG("SUBPROGRAMA COM INSERCAO DE CODIGO DE MAQUINA INVALIDA",PNO)
3226      END VERIFICA_SUBPROGRAM_BODY ;
3227  -----
3228  PROCEDURE VERIFICA_SUBPROGRAM_DECL(PNO : PT_TREE) IS
3229  PDESIG , PHEADER , PSUBDEF : PT_TREE ;
3230  BEGIN
3231      PDESIG := AS_DESIGNATOR(PNO) ;
3232      PHEADER := AS_HEADER(PNO) ;
3233      PBLOCK := AS_BLOCK_STUB(PNO) ;
3234      VERIFICA_DESIGNATOR(PDESIG , PNO) ;
3235      VERIFICA_HEADER(PHEADER , PNO) ;
3236      VERIFICA_SUBPROGRAM_DEF(PSUBDEF , PNO) ;
3237  END VERIFICA_SUBPROGRAM_DECL ;
3238  -----
3239  ---
3240  --- SUBTYPE => AS_ID : ID ,    --- SUBTYPE_ID

```

```

3241 --- AS_CONSTRAINED : CONSTRAINT ?
3242 ---
3243 --- SUBTYPE_ID => SM_TYPE_SPEC : CONSTRAINED ?
3244 ---
3245 PROCEDURE VERIFICA_SUBTYPE (PNO : PT_TREE) IS
3246 PID , PCONST : PT_TREE ?
3247 BEGIN
3248   PID := AS_ID(PNO) ?
3249   PCONST := AS_CONSTRAINED(PNO) ?
3250   IF DECLARADO(PID) THEN
3251     MSG("IDENTIFICADOR JA DECLARADO NO MESMO ESCOPO",PID) ?
3252   ELSE
3253     SM_TYPE_SPEC(PID) := PCONST ?
3254     VERIFICA_CONSTRAINED(PCONST) ?
3255   END IF ?
3256 END VERIFICA_SUBTYPE ?
3257 -----
3258 ---
3259 --- SUBUNIT => AS_NAME : NAME ?
3260 --- AS_SUBUNIT_BODY : SUBUNIT_BODY ?
3261 ---
3262 --- ESSA ESTRUTURA E PASSADA DIRETAMENTE PARA O TANGORETE, QUE REALIZA
3263 --- OS TESTES NECESSARIOS, E, SE TUDO ESTIVER CORRETO, MONTA O
3264 --- PARA ANALISE SEMANTICA DA UNIDADE.
3265 ---
3266
3267
**** FIN DO ARQUIVO.

```