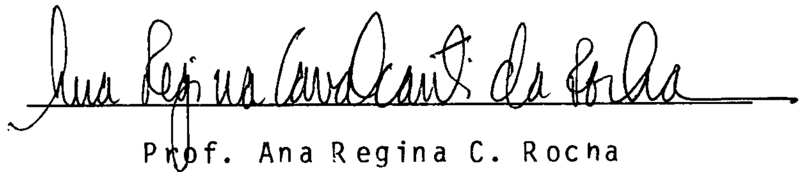
# UM ANALISADOR ESTÁTICO DE PROGRAMAS
# ESCRITOS EM C

JORGE BIDARRA

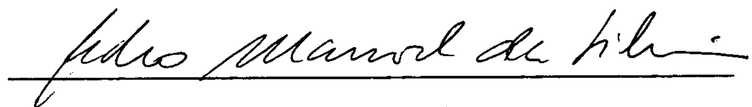
TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.) EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

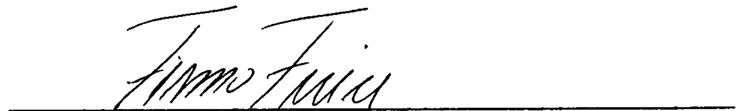


Prof. Ana Regina C. Rocha

(Presidente)



Prof. Pedro Manoel da Silveira



Dr. Firmo Freire

RIO DE JANEIRO, RJ - BRASIL

ABRIL DE 1988

BIDARRA, JORGE

Um Analisador Estático de Programas Escritos em C (Rio de Janeiro) 1988.

xi, 157 p. 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1988)

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1. Engenharia de Software

I.COPPE/UFRJ II. Título (série)

À minha mulher,

Zelimar S. Bidarra

A realização deste trabalho se deveu a muito esforço e a muitas dificuldades, principalmente decorrentes do pouquíssimo tempo disponível para a sua conclusão. Para que isto fosse possível, contei com o incentivo e a colaboração de muitos colegas que acabaram se envolvendo no processo. Assim sendo, gostaria de expressar os meus agradecimentos às seguintes pessoas:

A Ana Regina C. da Rocha pela oportuna orientação. Agradeço-a principalmente pela atenção e paciência dispensadas;

A Firmo Freire pelo apoio;

A César José dos Santos pelas discussões e sugestões no decorrer do desenvolvimento do texto da tese;

A Luís Paulo Almeida Nogueira pela enorme contribuição na implementação e testes do produto;

A Valeria Bastos Chaves e a Mauro Barbosa d' Oliveira pelas contribuições na implementação do Analisador Léxico e Sintático da linguagem C;

A Alcebíades Duarte de S. C. Neto e Maeli Cibeli Frederico Botelho pelo excelente trabalho de edição de texto;

A George W. B. de Oliveira pela confecção dos desenhos;

A todos os colegas da COBRA que direta ou indiretamente estiveram comigo nesta árdua tarefa e

A minha mulher, Zelimar, pelo valioso incentivo que só me impulsionou para a frente.

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

UM ANALISADOR ESTÁTICO DE PROGRAMAS ESCRITOS EM C.

Jorge Bidarra

Abril de 1988

Orientador : Ana Regina C. Rocha

Programa : Engenharia de Sistemas e Computação

Os problemas associados com o desenvolvimento de software têm evidenciado a necessidade de usar o próprio computador para fornecer um apoio automatizado às atividades de desenvolvimento.

Este trabalho descreve um ambiente de apoio automatizado para desenvolvimento de software básico de grande escala, com ênfase em uma de suas ferramentas: AEP - Um Analisador Estático de Programas Escritos em C.

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

A STATIC ANALYZER OF PROGRAMS WRITTEN IN C

Jorge Bidarra

April, 1988

Chairman : Ana Regina C. Rocha

Department : Systems Engineering and Computation

The problems associated with software development have shown the necessity of using computers to provide automated support for development activities.

This paper describes an automated environment for software development, emphasizing one of its tools: AEP - A Static Analyser of Programs written in C.

# ÍNDICE

I - INTRODUÇÃO .......................................... 1

I.1 O Processo de Desenvolvimento de Software .......... 3

I.2 A Qualidade de Software ............................ 5

I.2.1 Métodos de Avaliação da Qualidade de Software ... 8

I.3 Um Ambiente de Apoio Automatizado .................. 11

I.4. Objetivo e Organização deste Trabalho ............. 16

II - AVALIAÇÃO DA QUALIDADE DE PROGRAMAS ................. 17

II.1 Introdução ........................................ 17

II.2 Um Método para Avaliação da Qualidade de Software .. 17

II.2.1 Qualidade de Programas ......................... 21

II.2.1.1 O Fator Legibilidade ......................... 24

II.2.1.1.1 Simplicidade ............................... 27

II.2.1.1.2 Concisão ................................... 34

II.2.1.1.3 Estilo ..................................... 35

II.2.1.1.4 Modularidade ............................... 37

II.3 Técnicas de Avaliação da Qualidade de Software ..... 45

II.4 Analisadores Estáticos de Programas ............... 55

II.5 Um Analisador Estático de Programas escritos em C .. 56

III - AEP: O ANALISADOR ESTÁTICO ......................... 57

III.1 Introdução ....................................... 57

III.2 O AEP e suas Características ..................... 57

III.3 A Especificação de Requisitos do AEP ............. 60

III.3.1 Introdução ..................................... 60

III.3.1.1 Escopo ....................................... 60

III.3.1.2 Glossário .................................... 60

III.3.1.3 Referências .............................. 61

III.3.2 Descrição Geral ................................ 61

III.3.2.1 Atributos do Produto ........................ 61

III.3.2.2 Características do Usuário ................ 62

III.3.2.3 Ambiente do Produto ........................ 62

III.3.2.4 Restrições Gerais ........................... 63

III.3.2.5 Funções do Produto .......................... 63

III.3.3 Requisitos Específicos ........................ 65

III.3.3.1 Requisitos de Atributos .................... 65

III.3.3.2 Requisitos de Interfaces Externas ......... 65

III.3.3.3 Restrições de Desenho ...................... 66

III.3.3.4 Requisitos Funcionais ...................... 66

III.3.4 Definição de Interfaces Externas ............. 67

III.3.4.1 Interface de Usuário ....................... 67

III.3.4.2 Interface com o Ambiente .................. 74

IV - O PROJETO AEP

IV.1 Introdução ......................................... 76

IV.2 Especificação de Desenho de Software do AEP ........ 76

IV.2.1 Introdução ..................................... 76

IV.2.1.1 Escopo ....................................... 76

IV.2.1.2 Definições e Normas de Batismo dos Arquivos 77

IV.2.1.3 Referências/Bibliografia .................. 79

IV.2.2 Descrição Geral da Arquitetura ................ 80

IV.2.3 Especificações de Interfaces .................. 88

IV.2.4 Descrição Detalhada das Estruturas de Dados .... 117

V - CONCLUSÕES .............................................. 130

REFERÊNCIAS BIBLIOGRÁFICAS .................................. 132

ANEXOS:

Anexo 1: Manual do Usuário

Anexo 2: Exemplo da execução da Ferramenta

# ÍNDICE DAS FIGURAS

I.1 Ciclo de Vida de Software, Modelo "Cascata" ......... 6

I.2 Modelos de Avaliação da Qualidade de Software ....... 10

I.3 Ambiente de Apoio Automatizado para Desenvovimento de Software Básico .................................. 12

II.1 Estrutura do Método ................................ 20

II.2 Características de Qualidade de Programas ........... 23

II.3 Critérios de Avaliação do Fator Legibilidade ........ 26

II.4 Cálculo da "Complexidade Ciclomática de Mccabe" ..... 30

II.5 Atributos para Avaliação da Modularidade de Programas 40

II.6 Grafo de Fluxo ...................................... 52

II.7 Grafo de Chamada .................................... 52

II.8 Diagrama de estados das Variáveis ................... 54

# CAPÍTULO I - INTRODUÇÃO

Programar significa, a partir de uma necessidade bem definida, representar em código, que o computador consiga entender, todos os passos necessários para solucionar um problema. A estes passos denomina-se "Algoritmo".

No princípio, programar era uma tarefa muito complicada. De posse do algoritmo a tradução para o código do computador era penosa, posto que se utilizava de uma linguagem composta por sequências de zeros e uns que, organizados segundo uma prévia definição, significavam as instruções para o computador.

Com o tempo, o homem vem tentando desenvolver uma comunicação com a máquina mais amigável, criando linguagens cada vez mais próximas da linguagem natural; neste momento, surgem as linguagens de programação. Atualmente são várias as linguagens existentes: umas de fácil uso, outras mais complexas; umas de uso científico, outras de uso comercial.

Hoje em dia, escrever programas passou a ser uma tarefa mais simples. Entretanto, o disseminado uso do computador nos mais diversos segmentos da sociedade, tais como: hospitais, aeroportos, forças armadas, etc., tornou imperativa a necessidade da construção de programas de qualidade. Os programas de computador não são mais escritos por uma pessoa para resolver um problema individual; eles são desenvolvidos em ambientes complexos. Nestes ambientes, são vá-

rios os problemas encontrados para o seu desenvolvimento, RAMAMOORTHY (1):

. a existência de requisitos vagos e incompletos;
. a imposição de uma metodologia aos desenvolvedores;
. a obtenção do grau de confiabilidade e desempenho desejados;
. o prazo limitado para a conclusão dos programas.

Para evitar que estes problemas propiciassem a geração de programas não utilizáveis, os especialistas em computação passaram a pensar num processo já bastante usado em ambientes mais tradicionais - a Garantia da Qualidade.

Mas qualidade não pode ser definida universalmente; sempre está associada a alguma coisa, como por exemplo: qualidade de especificações, qualidade de projetos, qualidade de programas, etc. Em particular, a qualidade de programas deve ser buscada ao longo do processo de desenvolvimento. Para isso, é necessário que sejam especificados os atributos que a determinam. Vários estudos são realizados neste sentido, BOEHM (2), DUNN (3), CHOW (4), e deles originam-se os modelos de avaliação de qualidade.

Entretanto, a necessidade de apoiar os usuários de computadores na construção e verificação de seus produtos, visando a qualidade, levou à automatização destes modelos, hoje conhecida como Ambientes de Apoio Automatizado. Estes ambientes se caracterizam pelo fornecimento de ferramentas de

apoio à produção de software.

Nos últimos anos, o desenvolvimento de ferramentas de apoio aos programadores tem sido considerável. Neste conjunto encontramos os Editores de Programas, os Depuradores, os Analisadores, etc. A maioria destas ferramentas, no entanto, são contruídas para linguagens específicas. Este trabalho também propõe uma destas ferramentas. Trata-se de um protótipo de Analisador Estático destinado a programas escritos em C, denominado AEP. O objetivo desta ferramenta é auxiliar os programadores da linguagem C a construírem programas de qualidade.

## I.1 O Processo de Desenvolvimento de Software

Usualmente o desenvolvimento de software é efetuado através de refinamentos sucessivos que parte de uma especificação de requisitos e vai-se acrescentando detalhes à medida que se progride no desenvolvimento. Este processo, no entanto, não deve se dar livremente. Assim, para maior controle, é necessário que sejam definidas fases para o desenvolvimento do software. Cada uma destas fases corresponde a um período de tempo em que as atividades acontecem, com início e fim bem determinados. A este conjunto de fases denomina-se Ciclo de Vida do Software. Um dos modelos para a representação do ciclo de vida do software é conhecido por modelo em "cascata", conforme mostrado na figura (I.1). Em geral, estas fases são as seguintes:

- Especificação de Requisitos

  Esta fase deve produzir uma especificação completa das funções e características de desempenho do software, sem se ater a detalhes de implementação;

- Desenho de Software

  Esta fase deve especificar a configuração do sistema, os módulos que compõem o produto e suas comunicações, as estruturas de controle e de dados. A linguagem de programação a ser usada também deve ser especificada nesta fase;

- Desenho Detalhado

  Esta fase deve fornecer mais detalhes de especificação dos módulos como por exemplo: os algoritmos usados, as comunicações entre os módulos, etc.;

- Programação e Codificação

  Nesta fase os módulos são codificados na linguagem de programação especificada na fase de Desenho do Produto;

- Integração do Sistema

  Esta fase é responsável pela integração de todos os módulos do produto. Nela, o produto deve ser exaustivamente testado para garantir que os requisitos funcionais estão sendo alcançados;

- Instalação e Aceitação

  É a fase em que o produto será liberado para a organização para que sejam realizados os testes finais. As docu-

mentações e o manual do usuário devem ser liberados nesta fase;

- **Manutenção**

  É a fase em que devem ser feitas as correções de possíveis erros, alterações em código e manuais, acréscimo de novas funções e eliminação de funções inúteis.

## I.2 A Qualidade de Software

A aplicação de softwares é muito extensa. Vai da solução de problemas simples à solução de problemas complexos. Entretanto, para que o software atenda completamente aos propósitos a que se destina, é importante que a sua qualidade seja assegurada.

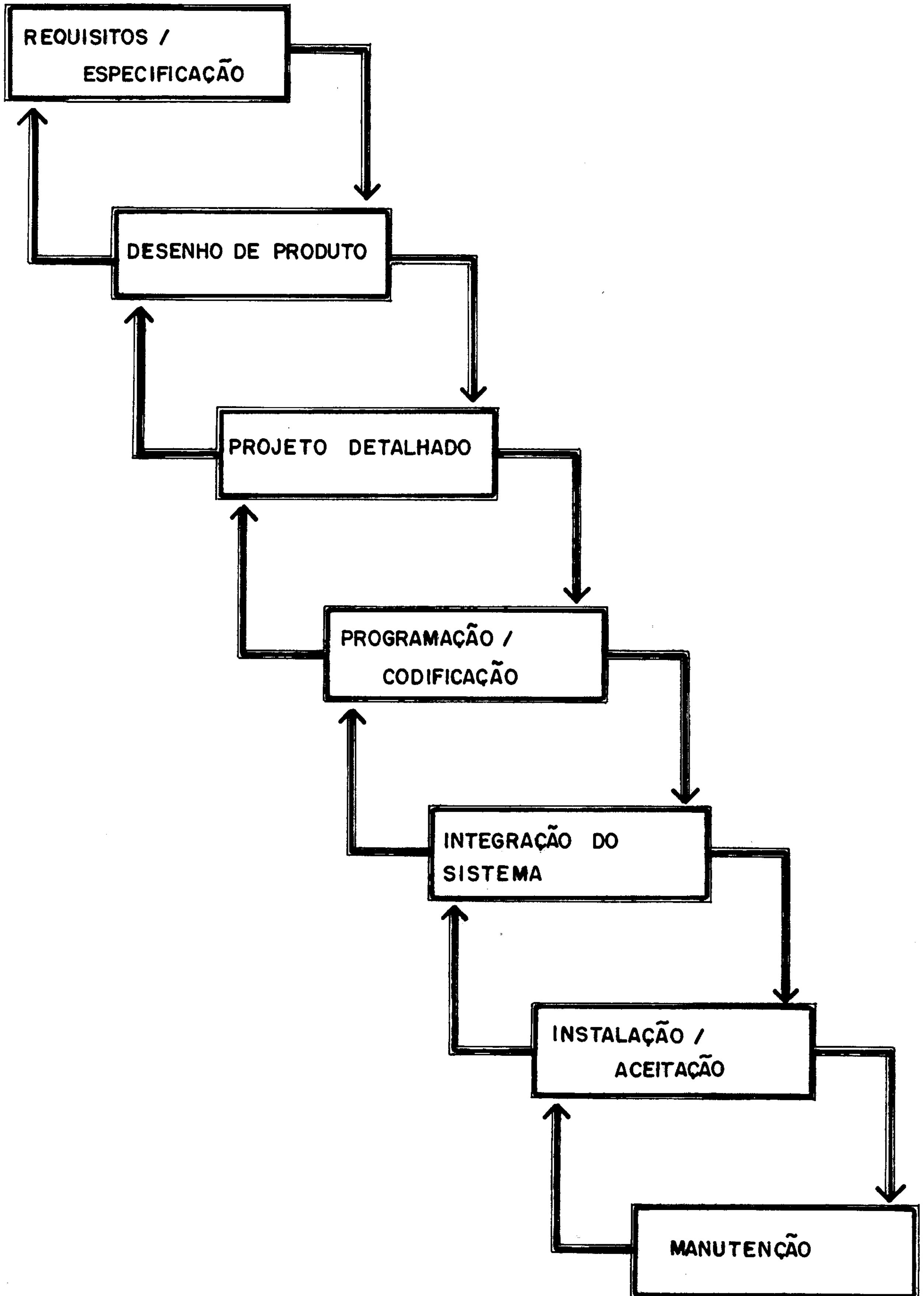



FIGURA I.1 - CICLO DE VIDA DE SOFTWARE, MODELO "CASCATA"
Fonte: CONTE (5)

Segundo Hans (6), a qualidade de um software é obtida se forem considerados os seguintes aspectos:

. O custo para se manter o software

O custo é tão mais alto quanto houver de erro em seus estágios iniciais e as dificuldades para se introduzirem acertos a ele posteriormente.

. Campos de aplicação do software

Há um crescente número de campos de aplicação para computadores cuja confiabilidade é essencial. A sua ausência se torna um risco.

Entretanto, um software não deve ser somente correto e confiável. Deve possuir outros atributos, como por exemplo:

. Ser Amigável

Por amigável entende-se aquele software cuja comunicação com o usuário seja clara e de fácil entendimento.

. Ser Modificável

Por modificável entende-se aquele software que, evidenciada uma anomalia, seja passível de alteração sem efeito colateral e, o que é mais importante, que o objetivo desejado não seja alterado.

. Ser Robusto

Por robusto entende-se aquele software que ao se

deparar com um problema seja capaz de tratá-lo de forma precisa e bem definida.

Embora haja uma grande variedade de atributos que determinam a qualidade de um software, eles são considerados conforme as necessidades de cada aplicação. Assim, a natureza de cada aplicação é quem define quais atributos devem estar presentes no software que a representa.

### I.2.1 Métodos de Avaliação da Qualidade de Software

A preocupação de se garantir um padrão para a qualidade de software levou os pesquisadores a proporem métodos de avaliação. Destacam-se entre eles os desenvolvidos por BOEHM et alli (2) e McCALL et alli (7). Tais modelos se assemelham no conjunto, diferindo apenas em algumas características propostas. Ambos foram desenvolvidos a partir de agrupamento de características de software, conforme pode ser visto na figura (I.2). Em última análise, as construções primitivas de Boehm e os critérios de McCall podem ser reduzidos a um conjunto de métricas que permitam a avaliação do software de forma quantitativa.

Porém, dada a complexidade dos ambientes de desenvolvimento de software, estes modelos, por si só, não são suficientes. Os problemas associados com o desenvolvimento de software têm mostrado a necessidade de um apoio automatizado para melhorar a produção técnica e gerencial.

Criar e evoluir software envolve muitos compromissos técnicos e administrativos, tais como: capacidade da equipe de desenvolvimento, planejamento, práticas comerciais e obrigações contratuais, facilidades computacionais. O ambiente no qual o software é construído reflete estes compromissos e deve fornecer ferramentas e métodos para a realização de todas as atividades do ciclo de vida do software.

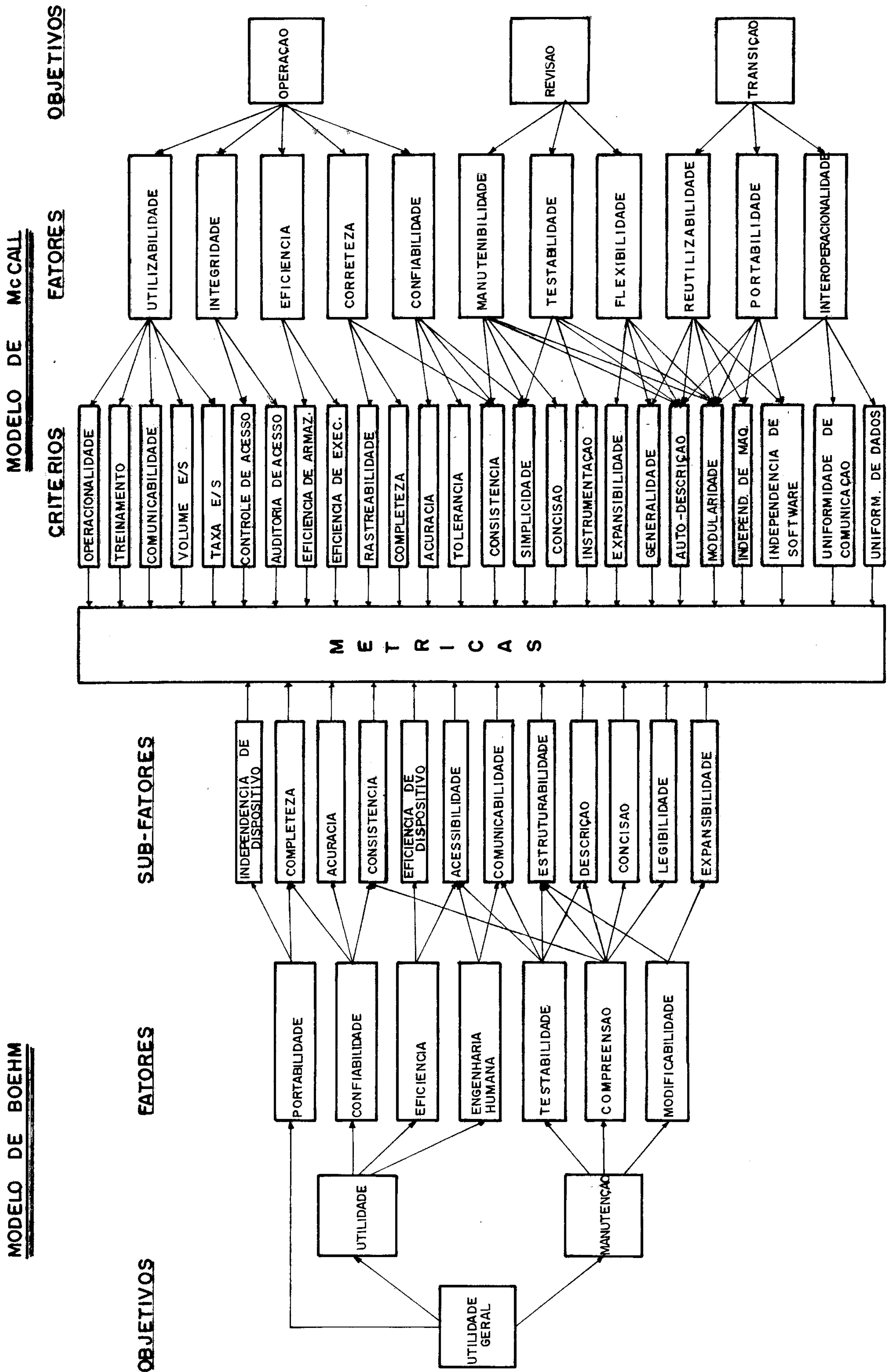



FIGURA I.2 - MODELO DE AVALIAÇAO DA QUALIDADE DE SOFTWARE.
Fonte: PERLIS (8).

Na seção seguinte descreveremos, com mais detalhes, um destes ambientes de apoio automatizados.

## I.3 Um Ambiente de Apoio Automatizado

O objetivo de um ambiente de apoio automatizado é fornecer um conjunto de ferramentas que automatize ou forneça apoio automatizado para as tarefas do processo de desenvolvimento de software. SANTOS (9), em seu trabalho, propõe um destes ambientes que será tomado como base para o desenvolvimento desta tese. A figura (I.3) dá uma visão geral do ambiente. Este ambiente é composto de dois níveis:

- Nível 0

  Constitue o núcleo do ambiente. Nele se encontram o hardware e o Software hospedeiro.

- Nível 1

  O nível 1 é constituído de ambientes mínimos. Cada um atende a um tipo de necessidade específica, tais como: apoio automatizado a gerenciamento, a metodologias, ou a áreas de aplicação. Para este nível foram identificados pelo menos três destes ambientes:

.AMAEP - ambiente mínimo de apoio à Engenharia do Produto.

.AMAGT - ambiente mínimo de apoio à Gerência Técnica.

.AMAVV - ambiente mínimo de apoio à Verificação e Validação.

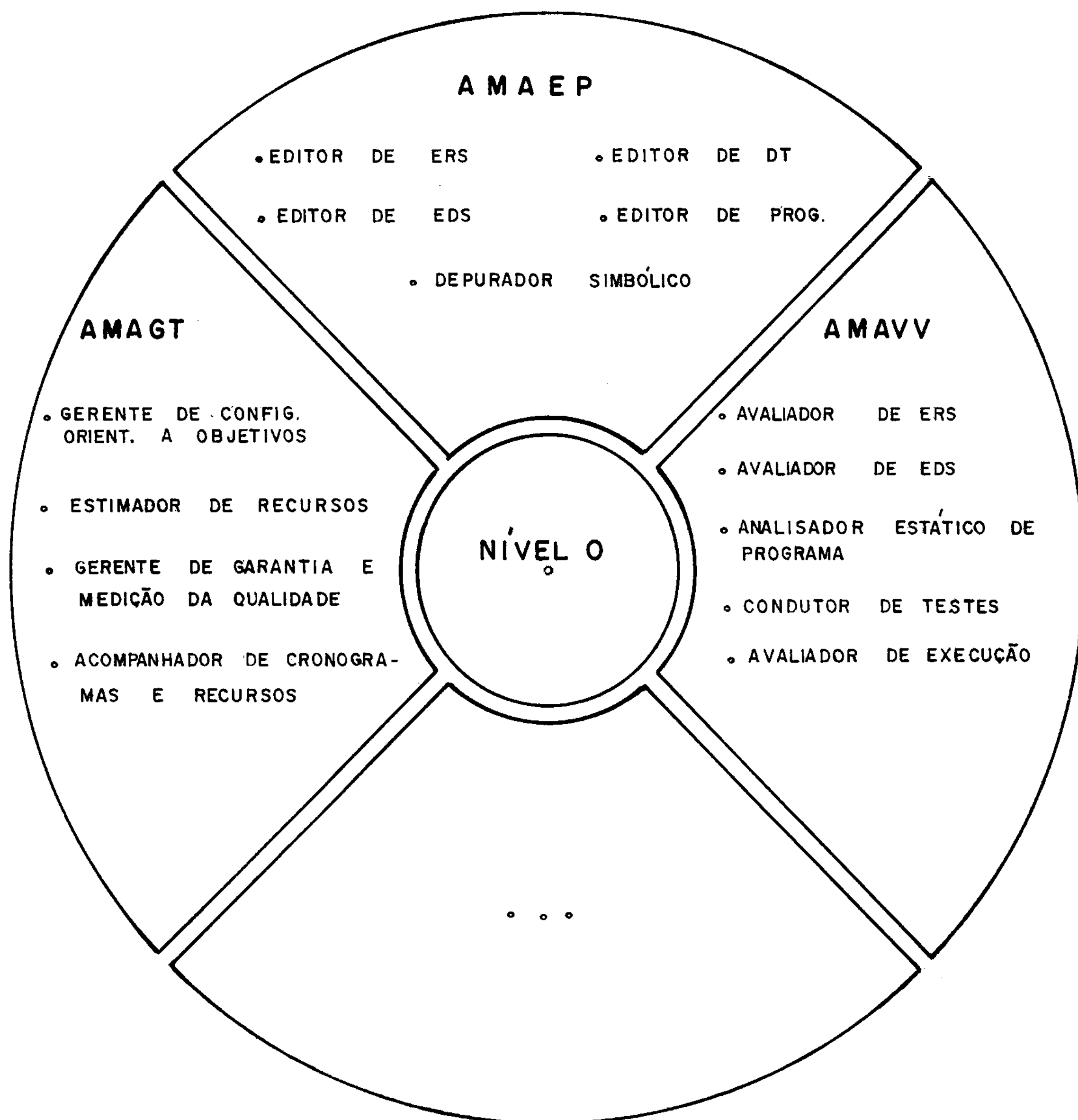



FIGURA I.3 — AMBIENTE DE APOIO AUTOMATIZADO PARA DESENVOLVIMENTO DE SOFTWARE BÁSICO.

Fonte: SANTOS (9).

A seguir serão descritas brevemente cada uma das ferramentas do ambiente:

- **Gerente de Configuração Orientado a Objetivos**

  O Gerente de Configuração Orientado a Objetivos fornece uma ferramenta que estabelece uma disciplina para o processo de desenvolvimento de software e provê uma maior visibilidade deste processo.

- **Estimador de Recursos**

  O Estimador de Recursos fornece uma ferramenta para estimativa dos custos de desenvolvimento, o que é de fundamental importância para o planejamento do projeto e avaliação da viabilidade.

- **Gerente de Garantia e Medição da Qualidade de Software**

  O Gerente de Garantia e Medição da Qualidade de Software fornece funções para apoiar no estabelecimento de medidas de qualidade e gráficos de acompanhamento da qualidade durante o ciclo de desenvolvimento de software.

- **Acompanhador de Cronograma e Custos**

  O Acompanhador de Cronograma e Custos é uma ferramenta que auxilia no Controle do projeto.

- **Editor de Especificação de Requisitos**

  O Editor de Especificação de Requisitos fornece uma ferramenta de apoio à produção deste documento. Esta ferramenta é um processador de textos especializado para este

tipo de documento.

- **Editor de Especificação de Desenho**

  O Editor de Especificação de Desenho fornece uma ferramenta de apoio à produção deste documento.

- **Editor de Documentação de Testes**

  O Editor de Documentação de Testes fornece uma ferramenta de apoio à produção dos documentos de testes: Planos de Testes, Procedimentos de Testes, e Relatórios de Testes ANSI/IEEE(10).

- **Editor de Programa Sensível a Linguagens**

  O Editor de Programa Sensível a Linguagens fornece, inicialmente, uma ferramenta de edição específica para programas-fonte em linguagem C. Futuramente outras linguagens poderão ser reconhecidas por este editor.

- **Depurador Simbólico**

  O Depurador Simbólico provê uma ferramenta de apoio ao processo de depuração em alto nível, isto é, uma ferramenta que permite ao usuário monitorar a execução de um programa sem ser obrigado a pensar em termos de linguagem de máquina.

- **Avaliador de Especificação de Requisitos**

  O Avaliador de Especificação de Requisitos fornece uma ferramenta para apoiar na avaliação da qualidade das especificações de requisitos.

- **Avaliador de Especificação de Desenho**

  O Avaliador de Especificação de Desenho fornece uma ferramenta para apoiar na avaliação da qualidade das especificações de desenho.

- **Condutor de Testes**

  O Condutor de Testes fornece uma ferramenta de auxílio à aplicação de testes e verificação de resultados.

- **Avaliador de Execução**

  O Avaliador de Execução fornece uma ferramenta para análise do comportamento do programa durante a sua execução.

- **Analisador Estático de Programas**

  O Analisador Estático de Programas, inicialmente, tem por objetivo analisar o código fonte de um programa escrito em C. Nos próximos capítulos descrevemos, com mais detalhe, esta ferramenta.

## I.4 Objetivo e Organização deste Trabalho

O objetivo deste trabalho é definir uma ferramenta para avaliar a qualidade de programas escritos na linguagem "C". Esta ferramenta é um elemento do conjunto AMAVV. Abaixo são descritos brevemente os conteúdos dos demais capítulos.

O capítulo II discorrerá sobre a qualidade de programas e proporá um modelo para o analisador de programas escritos em "C".

O capítulo III, a partir do modelo descrito no capítulo anterior, especificará o analisador, usando como manual de especificação o proposto no ambiente automatizado referido no item (I.3).

O capítulo IV descreverá o projeto e a implementação da ferramenta apresentada.

O capítulo V é destinado à conclusão do trabalho.

Além destes capítulos, esta tese terá dois anexos:

Anexo 1: Conterá o Manual de Utilização da Ferramanta.

Anexo 2: Conterá um exemplo de programa avaliado pela ferramenta.

# CAPÍTULO II - AVALIAÇÃO DA QUALIDADE DE PROGRAMAS

## II.1 Introdução

Qualidade de software não se atinge por si só, mas deve ser cuidadosamente buscada ao longo do seu desenvolvimento. Para isto, é necessário conhecer que atributos determinam a qualidade e que técnicas de avaliação adotar no sentido de controlar o processo de desenvolvimento para se atingir o nível de qualidade desejado.

Este capítulo estuda um método e técnicas para avaliar a qualidade de software. O método é apresentado na seção II.2 com uma particularização para permitir a avaliação da qualidade de programas. A seção II.3 descreve as técnicas mais usuais para a avaliação da qualidade de software. Dentre as técnicas de avaliação da qualidade, a análise estática é abordada em detalhes. Na seção II.4 são apresentadas as ferramentas automatizadas de apoio à análise estática: os analisadores estáticos de programas; e na seção II.5 é proposto um analisador estático para programas escritos na linguagem C.

## II.2 Um Método para Avaliação da Qualidade de Software

Desenvolver software de qualidade não é uma tarefa simples. O grau de formalidade e o tempo despendido nas várias etapas que envolvem a sua construção variam de acordo com a complexidade e o tamanho do produto. Não adianta se colocar

um produto em uso num curto espaço de tempo sem que sejam observadas as características que garantam a sua qualidade. Esta conduta pode levá-lo a uma sub-utilização e/ou gerar um alto custo de manutenção pelos reparos que se tornarão necessários ao longo do seu uso. Produzir softwares de qualidade tem sido um dos principais objetivos da pesquisa em engenharia de software. No trabalho desenvolvido por ROCHA (11) encontramos a definição de um método para avaliação de qualidade de software, sintetizado na figura (II.1). Este método se baseia nos seguintes conceitos:

. Objetivo de Qualidade
São propriedades gerais que um produto deve possuir;

. Fatores de Qualidade
São os determinantes da qualidade do produto, considerando-se a visão dos diferentes usuários do produto;

. Critérios
São atributos primitivos que podem ser efetivamente avaliados;

. Processos de avaliação
Determinam o processo e os instrumentos a serem utilizados para a medição do grau de presença, no produto, de um determinado critério;

. Medidas

São os resultados obtidos ao se avaliarem os critérios;

. Medidas Agregadas

Indicam o grau de presença de um determinado fator;

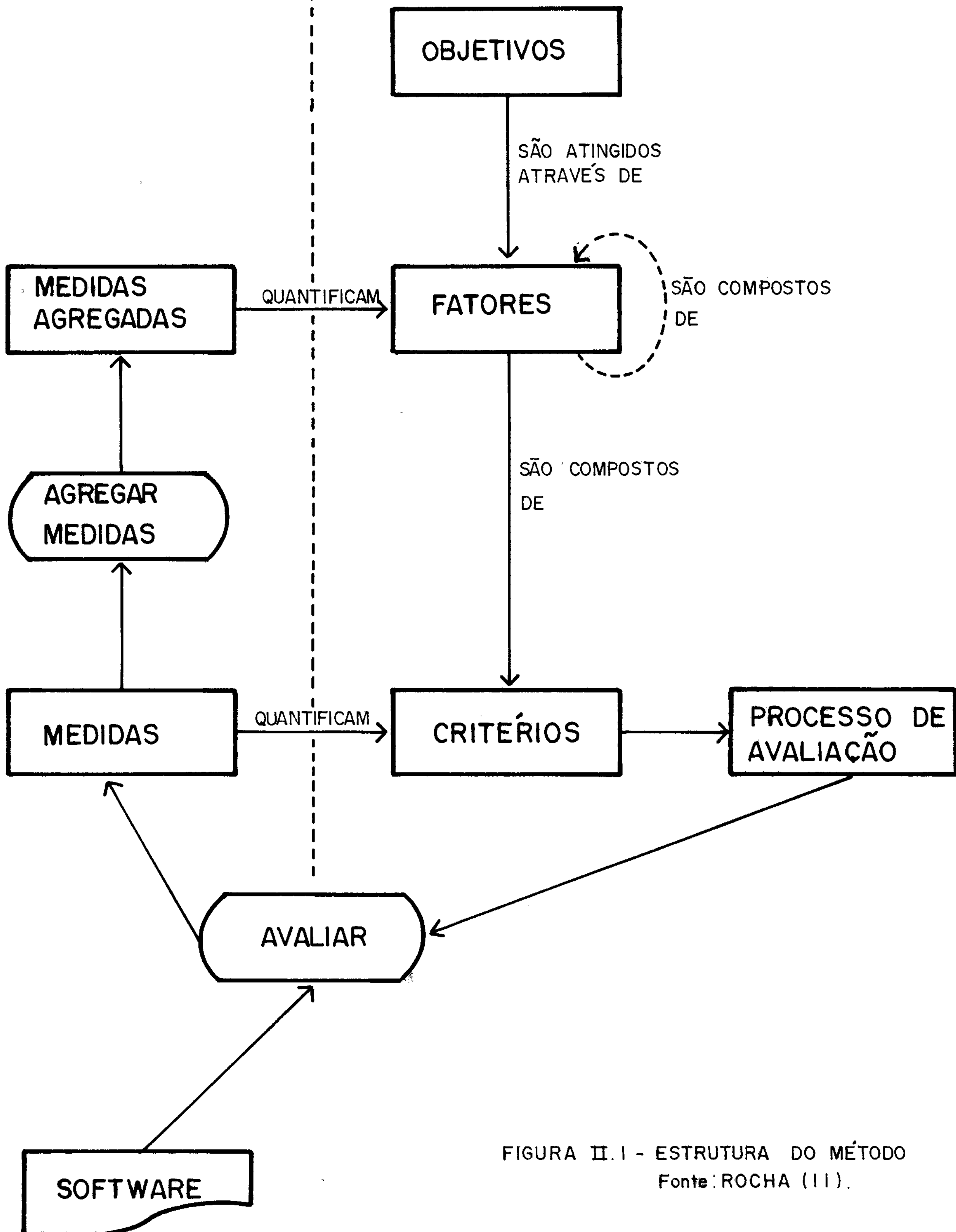



FIGURA II.1 - ESTRUTURA DO MÉTODO
Fonte: ROCHA (11).

Este método é uma tentativa de estabelecer meios que permitam a quantificação dos atributos de qualidade de software, através do desenvolvimento de métricas que avaliem os vários fatores que os envolvem. Para tanto, partiu-se dos objetivos de qualidade de software que foram sendo detalhados até se alcançarem os processos de medição. Neste processo de refinamento, para que os objetivos de qualidade fossem atingidos, evidenciou-se a necessidade da identificação de fatores que influenciam a qualidade. Os fatores envolvem desde o aspecto pessoal, tal como a habilidade de quem desenvolverá o software, até o tempo disponível para o seu desenvolvimento. Mas objetivos e fatores não são mensuráveis. É necessário se definir critérios para qua possam ser avaliados. Os critérios são atributos primitivos, independentes uns dos outros, que não têm significado isoladamente. No entanto, em conjunto, descrevem completamente um determinado fator que, por sua vez, também em conjunto, descreve os objetivos.

### II.2.1 Qualidade de Programas

Para permitir a avaliação da qualidade de programas, o método foi particularizado no trabalho desenvolvido por FREITAS et alli (12), onde foram identificados três objetivos fundamentais para a avaliação da qualidade de programas:

. **Utilizabilidade**

Um programa ao ser liberado deve ser utilizável. A

utilização de um programa acontece de diferentes maneiras. Ele pode ser usado na fase de operação, seus módulos podem ser reutilizados por outros programas, etc.

. **Confiabilidade Conceitual**

Quando escrito, um programa deve satisfazer às necessidades e aos requisitos que motivaram a sua construção.

. **Confiabilidade da Representação**

Ao se escrever um programa, deve-se ter em mente que ele será alvo de leitura e até de manipulação de seu código por parte de terceiros que, na maioria das vezes, não participaram da sua construção.

A figura (II.2) dá uma visão geral dos objetivos, fatores e sub-fatores identificados para avaliação de programas.

Tendo em vista que o objetivo deste trabalho é fornecer uma ferramenta de apoio ao programador, visando a construção de programas legíveis, foge ao seu escopo definir todos os fatores e critérios relacionados a cada objetivo. Portanto, a seguir discutiremos com mais detalhes o fator Legibilidade.

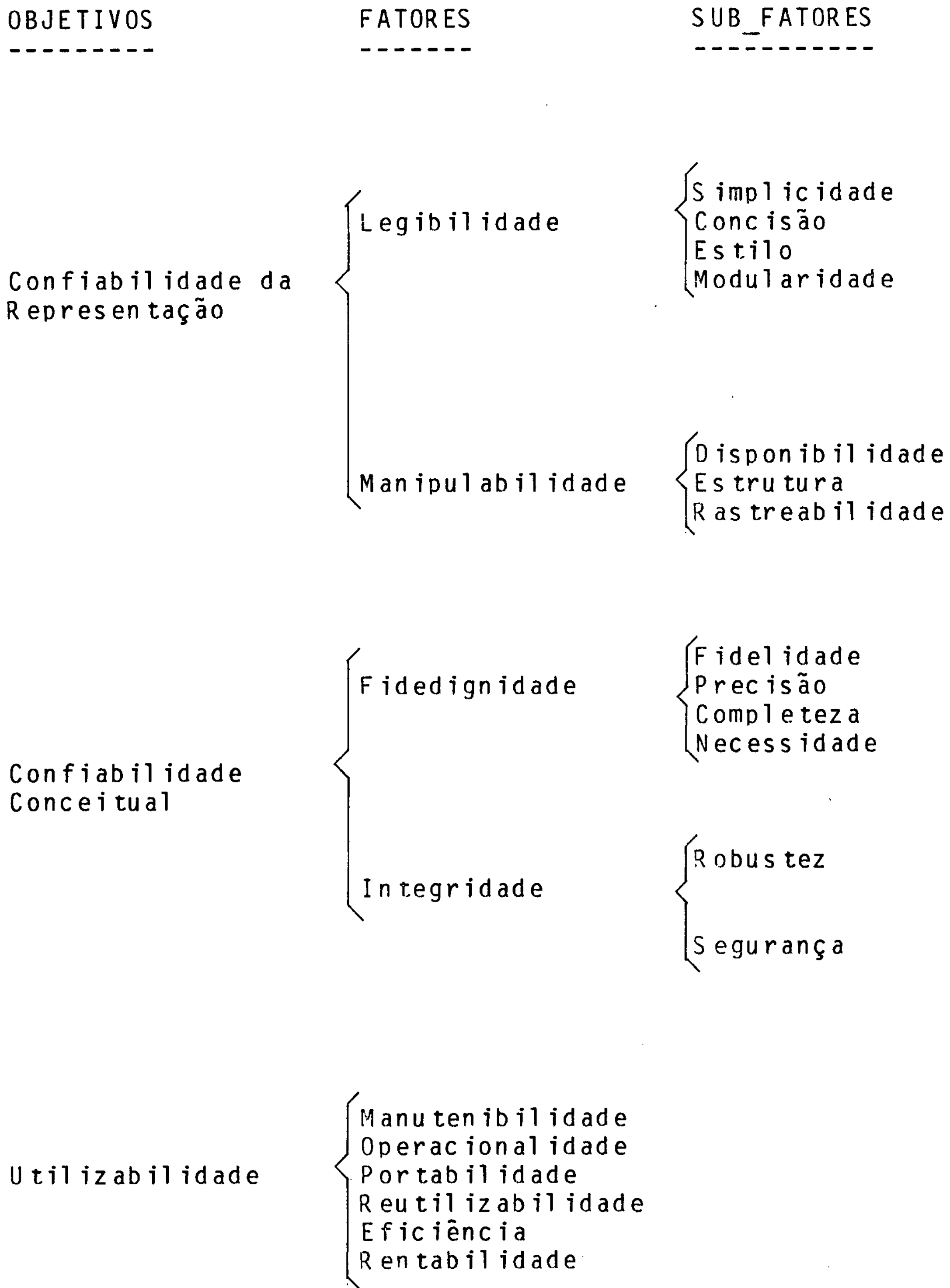



Figura II.2 - Características de Qualidade de Programas
Fonte: FREITAS (12)

### II.2.1.1 O Fator Legibilidade

Escrever programas é uma tarefa que requer um cuidado especial com a forma da representação do código. Os programas desenvolvidos por uma pessoa ou por uma equipe não devem ser encarados como uma propriedade particular, tornando-se inacessível a outras pessoas. Em grandes ambientes de processamento de dados, é muito comum os programas serem escritos por determinados programadores e manipulados por outros, que consequentemente precisam entendê-los. Assim sendo, uma das maiores preocupações nestes ambientes é quanto à construção de programas legíveis. Um programa deve ser legível o suficiente para que outras pessoas que necessitem entender o seu código não se percam nas dificuldades nele introduzidas pelo programador.

Legibilidade é um atributo de qualidade que se refere às características da representação do programa que afetam a sua compreensão por pessoas.

A legibilidade de um programa pode ser obtida se conseguirmos dar a ele as seguintes características:

. Simplicidade

Um programa deve ser simples na forma de apresentação. A complexidade de código deve ser evitada tanto quanto possível, porque através dela o entendimento é prejudicado.

. Concisão

Um programa deve ser conciso. O excesso de código, de um modo geral, torna o programa pouco objetivo e suscetível a maior ocorrência de erros.

. Estilo

A maneira como um programa é apresentado influencia fortemente a sua legibilidade. O uso de identação, comentários adequados e a padronização de identificadores, por exemplo, ajudam na compreensão do código fonte.

. Modularidade

Um programa deve ter as funções a ele inerentes organizadas de forma que a estrutura de módulos as tornem independentes. A dependência entre módulos é prejudicial na medida em que qualquer alteração ocorrida em algum deles tende a afetar os demais.

Entretanto, para que tais características sejam verificadas nos programas, é necessário definir critérios. Nos sub-ítens seguintes serão estudados alguns dos possíveis critérios que possibilitam esta verificação. A figura (II.3) dá uma visão geral destes critérios.

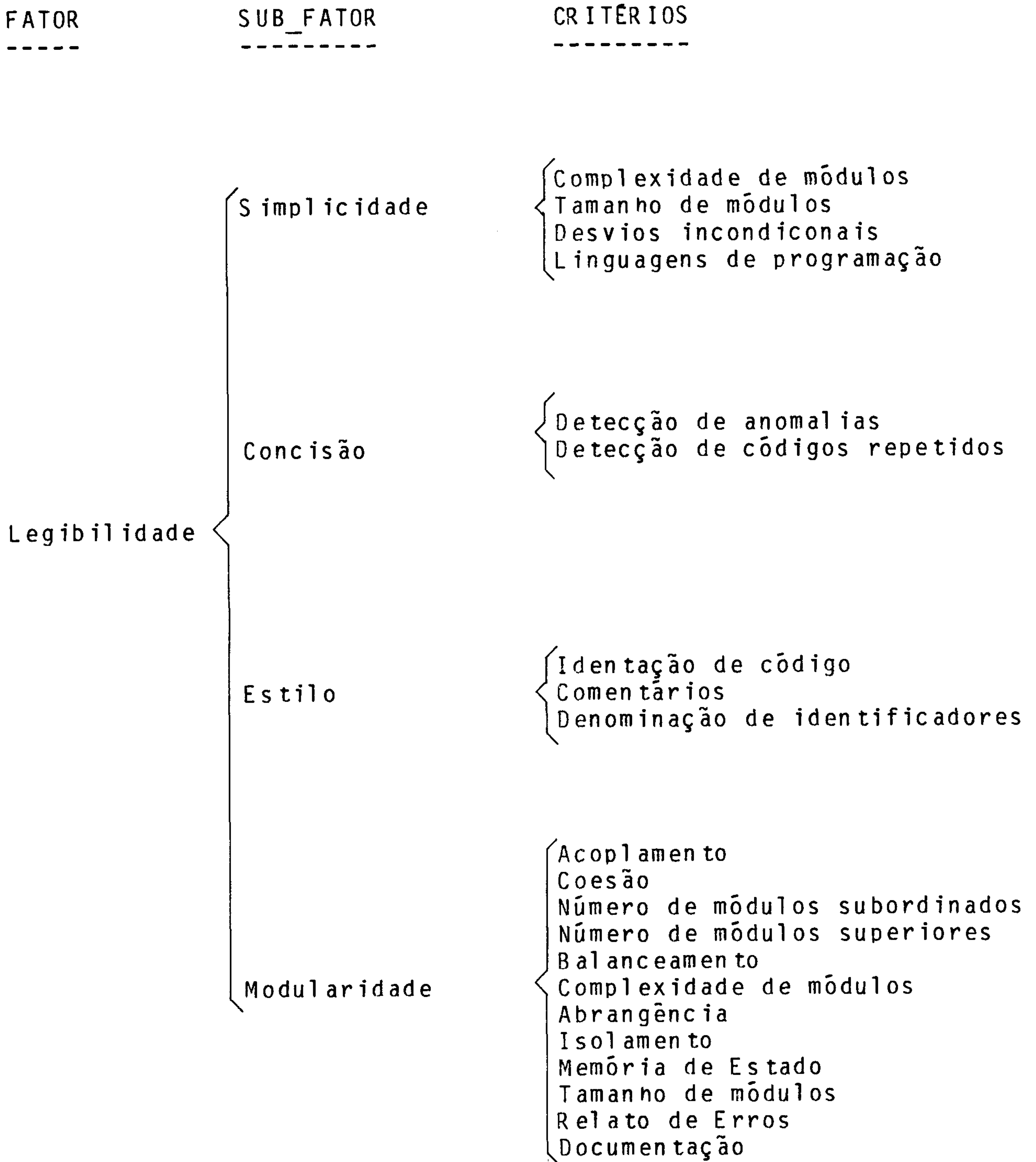



Figura II.3 - Critérios de Avaliação do Fator Legibilidade

### II.2.1.1.1 Simplicidade

Por definição, FERREIRA (13), simplicidade é a qualidade do que não apresenta dificuldade ou obstáculo. Em programação esta definição também se aplica porque é extremamente necessário que um programa, ao ser escrito, tenha uma codificação tão clara quanto possível, isento da complicação que é da tendência humana, uma vez que poderá ser manipulado por terceiros. O que se espera obter com a simplicidade de programas é que seus códigos sejam facilmente verificados, depurados, testados e modificados.

Ao contrário do que se pensa, não é imediato se alcançar a simplicidade. Para tanto, são necessários raciocínio e alguma experiência.

Estudos têm sido realizados para se estabelecer os critérios relevantes para que a simplicidade seja obtida. Os resultados obtidos mostraram que a simplicidade está diretamente relacionada à:

- Complexidade de módulos;
- Tamanho do programa;
- Existência de comandos de desvios incondicionais no programa;
- Linguagem de programação.

- **Complexidade**

Por Complexidade entende-se a dificuldade que uma pessoa

terá para compreender como um sistema, um programa ou como um módulo trabalha, LOWELL (14). Tão mais simples será um programa, quanto menor for a sua complexidade.

A complexidade de um módulo depende de suas estruturas de decisão e do número de caminhos que contém. O excessivo número de caminhos num módulo afeta sua testabilidade e manutenibilidade.

Existem alguns modelos matemáticos que estimam a complexidade de um programa. Entre eles destaca-se a teoria desenvolvida por McCABE (15).

A teoria de McCabe, conhecida por "Complexidade Ciclomática de McCabe", analisa as decisões encontradas num programa, partindo da proposição de que ele seja visto como um grafo que tenha uma única entrada e uma única saída. Valendo-se da teoria de grafos, McCabe obteve a seguinte relação matemática:

$$V(G) = e - n + 2p$$

onde: $e$ = número de ligações no grafo

$n$ = número de nós no grafo

$p$ = número de componentes conexos no grafo

Após vários testes, McCabe descobriu que o número de complexidade ciclomática, V(G), menor ou igual a 10 é mais adequado. Segundo ele, módulos com V(G) maior do que 10

precisam ser analisados e posteriormente reduzidos.

LOWELL (14) fornece outra forma de se calcular a complexidade ciclomática:

V(G) = SOMA (Decisões, ANDs, ORs, NOTs) + 1

A figura (II.4) mostra um exemplo de cálculo da complexidade através da teoria desenvolvida por McCabe.

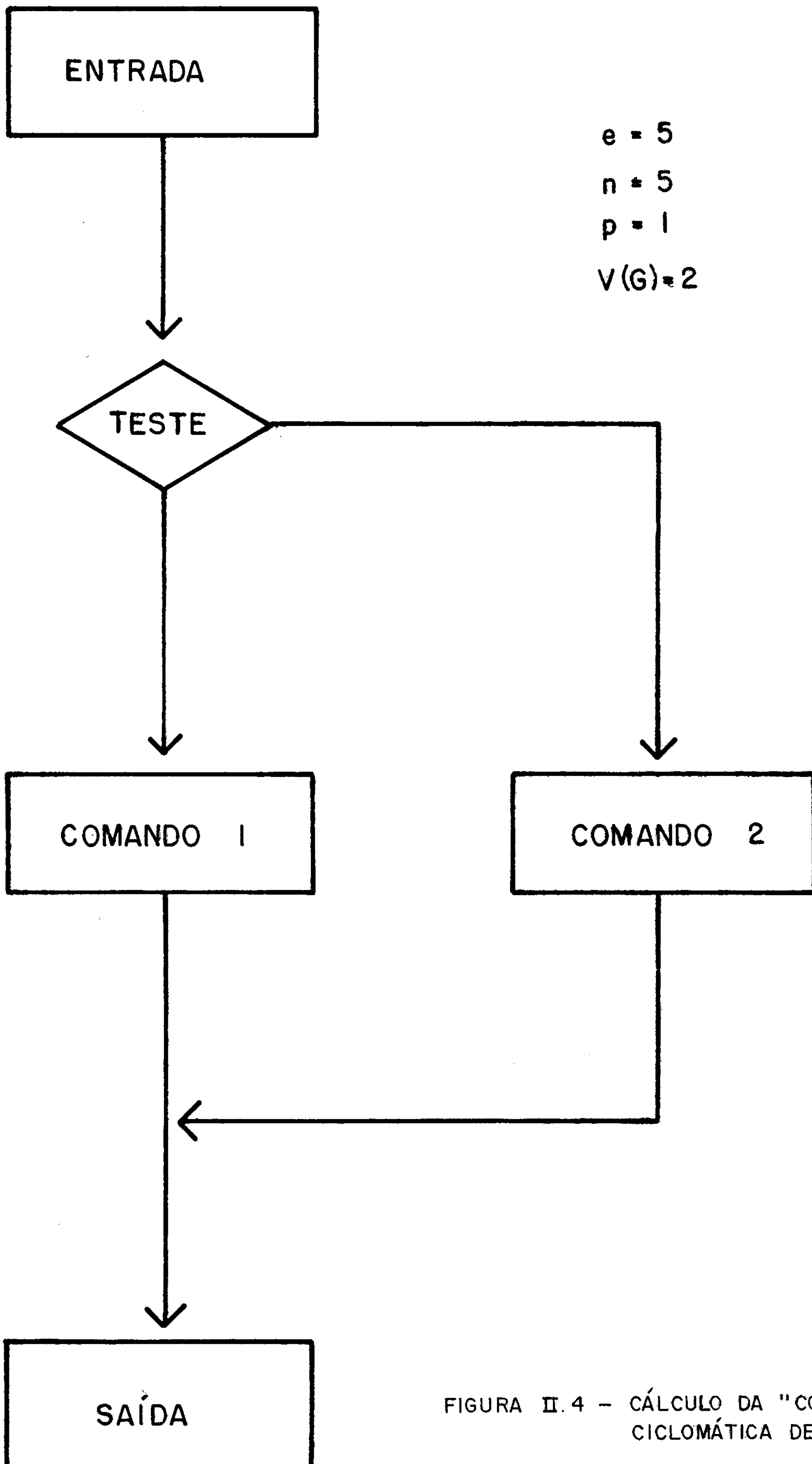



FIGURA II.4 – CÁLCULO DA "COMPLEXIDADE CICLOMÁTICA DE McCABE"

- **Tamanho do Programa**

Por tamanho de programa entende-se, em princípio, o número de linhas que ele possui. ELSHOFF (16) afirma que programas muito grandes induzem a erros em virtude do volume de informações a serem absorvidas para a sua compreensão.

Mas na realidade, tamanho de programa pouco significa em termos de simplicidade. Um programa pode ser considerado muito grande e no entanto ser simples. Logo, medir o tamanho de programa não é um processo consistente. Todavia, podemos restringir este conceito se considerarmos cada módulo independente do programa.

Alguns pesquisadores sugerem processos de avaliação para o tamanho. HALSTEAD (17), p.ex., calcula o tamanho de um módulo totalizando o número de tokens usados para a sua codificação. YOURDON (18) avalia o tamanho de um módulo pelo número de comandos que ele possui. Segundo Yourdon um módulo deve ter no máximo cinquenta comandos. NASCIMENTO (19) também desenvolveu um estudo a respeito, tendo chegado a duas conclusões:

. Módulos maiores do que duas páginas devem ser verificados quanto à possibilidade de serem separados, desde que a funcionalidade não seja comprometida, e,
. Módulos com menos de cinco linhas devem ser examinados para uma possível junção com seus módulos superiores.

De um modo geral, ainda não existe uma forma de provar que este ou aquele tamanho de módulo é mais adequado. Entretanto, medir o tamanho de um módulo é um critério que não deve ser desconsiderado. Devemos ter em mente que, via de regra, módulos menores são mais fáceis de serem entendidos e consequentemente testados e modificados por qualquer pessoa.

- **Existência de Comandos de Desvios Incondicionais (GOTO´s)**

  É mais fácil se compreender o funcionamento de um programa se os sucessivos comandos de seu código corresponderem a sucessivas ações no tempo, CONTE(5). A presença dos comandos de desvios incondicionais num código rompe tal relação, tornando a sua estrutura mais complexa e consequentemente menos simples.

  Mas nem sempre é possível se evitar o uso deste tipo de comando. Algumas linguagens de programação, como é o caso do FORTRAN, obrigam o programador a fazer uso dele. Nestas linguagens, o comando de desvio incondicinal serve para simular comandos de controle não disponíveis na estrutura da linguagem.

  Embora o comando em si possa ser usado de forma positiva, na maioria das vezes, é utilizado de forma abusiva, tornando a estrutura do programa muito complexa e sujeita a maior incidência de erros.

  Em linguagens de programação que forneçam estruturas de controle mais elaboradas, os comandos de desvios incondi-

cionais devem ser detectados e os trechos de código onde ocorrem devem ter a sua lógica refeita a fim de eliminá-los. A remoção de GOTO's tende a tornar a lógica de decisão mais clara, produzindo módulos mais legíveis.

- **Linguagem de Programação**

A linguagem de programação tem uma forte influência sobre a simplicidade do código de um programa. Um programa escrito em linguagem de alto nível tende a ser mais legível do que um programa escrito em linguagem de baixo nível. Quanto mais a linguagem de programação se aproxima da linguagem natural, mais simples é a codificação e a leitura de um programa.

As linguagens de programação mais modernas fornecem uma variedade de construções que facilitam o desenvolvimento de programas mais simples. Estas construções incluem: definição de módulos que podem ser compilados em separado, inclusão de arquivos em programas, definição de tipos de dados do usuário, criação de macros, etc.

Mas nem sempre a escolha da linguagem em ambientes de processamento de dados é possível. É uma questão de disponibilidade e de característica do ambiente. Neste caso, o programador deve ser capaz de utilizar a linguagem disponível, buscando a forma mais simplificada de codificação e explorando as facilidades que elas fornecem, comentando os trechos do programa mais complexos, de modo a obter maior legibilidade nos programas produzidos.

II.2.1.1.2 Concisão

Um programa conciso é aquele implementado com o mínimo de código possível. A objetividade durante a codificação de um programa é um fator muito importante para se alcançar a concisão.

É importante ressaltar que programas concisos produzem códigos mais simples, mas isto não deve significar que a concisão seja buscada sem critérios. A concisão não deve dificultar a compreensão do código, produzindo um efeito contrário ao esperado. Normalmente, a concisão de programas é obtida através das facilidades oferecidas pelas linguagens de programação.

Avaliar a concisão de programas significa, por exemplo, considerar os seguintes critérios:

- **Escolha da Linguagem de Programação**

  Quanto mais a linguagem de programação se aproxima da linguagem natural, mais concisa é a forma de se escrever os comandos para o computador.

  Uma linguagem de alto nível, oferece mais recursos para se obter a concisão de programas do que uma linguagem de baixo nível ("assembly").

  Um comando da linguagem de alto nível, na maioria das vezes, representa o equivalente a várias linhas de

código da linguagem de baixo nível.

- **Detecção de Anomalias de Código**

  A ocorrência de práticas de programação não recomendáveis, como, por exemplo, aparecimento de variáveis declaradas e não utilizadas e trechos de programas nunca alcançados, se não geram erros, pelo menos geram informações desnecessárias. O código de um programa deve ser percorrido no sentido de se detectarem as possíveis anomalias existentes. A eliminação das anomalias num código, além de evitar possibilidade de erros, permite se reduzir o código sensivelmente.

  Uma técnica bastante usada no processo de detecção de anomalias de código é através da análise estática do programa. Esta técnica será discutida com mais detalhes na seção II.3.

- **Verificação de Códigos Repetidos**

  É muito comum em programas aparecerem trechos que se repetem. Algumas vezes eles diferem por pequenas variações. Não há uma forma de se detectarem tais ocorrências se não através da observação do próprio código. Tanto quanto possível estes trechos devem ser detectados e reduzidos a um único trecho. Quanto menos ocorrência deste tipo houver, mais conciso se tornará o programa.

#### II.2.1.1.3 - Estilo

Estilo é a maneira como um programador usa a sua criatividade para dar ao programa um código claro, confiável, manutenível e eficiente. Portanto, estilo é uma questão de criatividade pessoal.

Não há leis ou regras que controlem este processo. Entretanto, recomenda-se aos programadores a utilização de algumas técnicas sugeridas na literatura ou o uso de normas estabelecidas pela empresa onde trabalham. O respeito a algum critério de programação garante homogeneidade aos produtos.

Os processos mais usuais entre os programadores que lhes conferem um bom estilo são os seguintes:

- **Identação de códigos e inserção de linhas em branco**

  É a técnica usada para permitir a separação de código em diversas partes, destacando as seções, as rotinas, as declarações e os comentários.

- **Uso de Comentários**

  Um programa deve ser bem comentado, principalmente os que apresentam certa complexidade. O uso do comentário é importante para relacionar o programa às entidades do mundo real que estão sendo modeladas por ele.

  Apesar das linguagens de programação fornecerem meios de se comentar um programa, certas dúvidas podem ser

levantadas:

. Quantos comentários são suficientes num programa?
. Onde devem ser colocados os comentários efetivamente?
. Os comentários podem confundir os leitores do programa?
. Os comentários são manuteníveis?

Para a maioria das perguntas acima, não há respostas. Mas, não há dúvida de que comentários são necessários, desde que, ao comentarmos qualquer trecho de programa, tenhamos disciplina e o cuidado de não gerarmos informações inconsistentes, pois um mau comentário é pior de que não tê-los.

- **Denominação de Identificadores**

É comum a criação de normas para a denominação de identificadores. Uma prática muito explorada é o uso de prefixos ou sufixos nos identificadores, imputando-lhes uma determinada categoria, como por exemplo, identificadores iniciados pela letra "D" especificam um dado elementar. Além disso, os nomes dos identificadores devem ser auto-explicativos, ou seja, através deles devemos ser capazes de descobrir a sua natureza, o seu escopo e outros tipos de informação.

II.2.1.1.4 - **Modularidade**

Moduralidade é a característica de um programa implementar as funções a ele pertinentes com uma estrutura de módulos altamente independentes.

Módulos são trechos de programa independentes e endereçáveis aos quais são dados nomes. Em geral, estes trechos constituem os chamados sub-programas ou rotinas. Uma definição mais formal para estas entidades pode ser:

. Módulos contêm instruções, lógica de processamento e estrutura de dados;

. Módulos podem ser compilados separadamente e armazenados em biblioteca;

. Módulos podem ser incluídos num programa;

. Módulos podem ser chamados por outros módulos e vice-versa.

O objetivo da moduralidade é separar um programa complexo em sub-programas menores e mais simples. Com isto fica mais fácil a realização de verificações, manutenções e reutilizações das partes.

Há vários critérios para a obtenção da modularidade de um programa. Dependendo do critério usado, diferentes estruturas de dados podem ser obtidas. Alguns destes critérios estão relacionados a seguir, FAIRLEY (20):

. Critério Convencional

É o critério onde um módulo e seus sub-módulos correspondem a um passo de processamento na sequência de execução;

. Critério de se esconder informações

Neste caso cada módulo esconde dos demais as dificuldades e as decisões de alteração do projeto;

. Critério da abstração de dados

O módulo esconde os detalhes de representação das maiores estruturas de dados;

. Níveis de abstração

É o critério no qual uma coleção de módulos fornece um conjunto hierárquico de tarefas complexas.

Na prática a modularização de um programa pode ser obtida usando-se um único critério ou misturando aspectos de vários critérios. O importante é que o processo usado seja bem definido.

Buscar uma forma de avaliação do grau de modularidade de um programa tem sido objeto de estudo. Em NASCIMENTO (19) encontramos o desenvolvimento de um trabalho que tem este objetivo. A figura (II.5) mostra o conjunto de atributos identificado.

Neste trabalho a modularidade foi classificada considerando-se três aspectos:

. Estrutural;

. Lógico;

. Psicológico

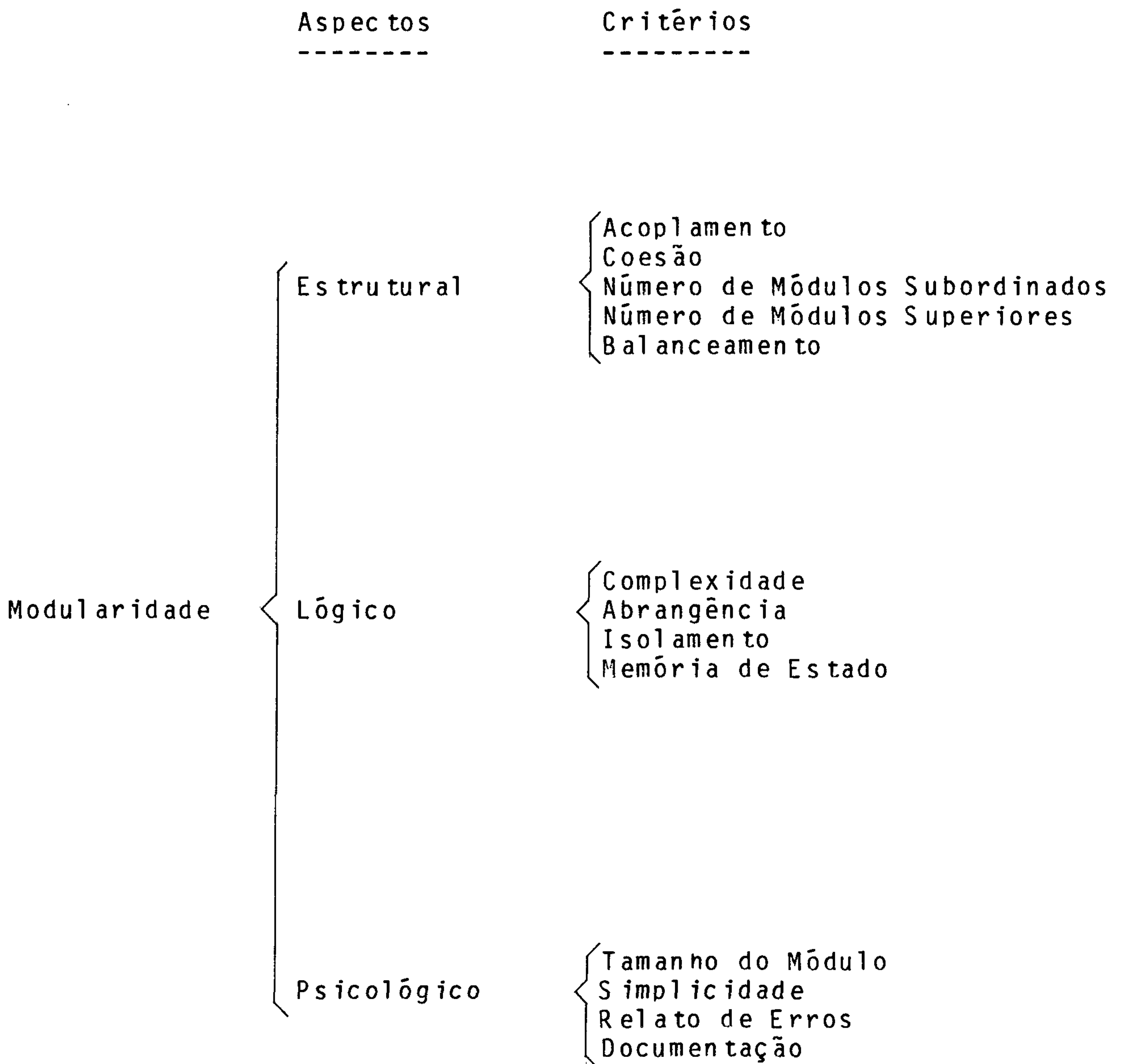



Figura II.5 - Atributos para Avaliação da Modularidade de Programas
Fonte: NASCIMENTO (19)

O aspecto estrutural se preocupa com a qualidade da estrutura modular do programa. Estruturas modulares não adequadas geram programas não manuteníveis.

O aspecto lógico se refere às características intrínsicas de cada módulo.

O aspecto psicológico leva em conta se um programa está inteligível ou não para as pessoas.

A Avaliação da modularidade segundo o aspecto estrutural envolve os seguintes conceitos:

- Acoplamento

  É o grau de interdependência entre os módulos.

  A possibilidade de modificação em um módulo pode provocar alterações em outro módulo. Um sistema bem particionado é aquele cujos módulos apresentam baixo aclopamento.

  O acoplamento se dá de acordo com a seguinte classificação:

  . Acoplamento por dados estruturais

    A comunicação entre os módulos é feita através de estruturas de dados;

  . Acoplamento por Controle

    A comunicação entre os módulos é através da passagem de variáveis de controle;

. Acoplamento por área de dados comum

Os dados se referem a mesma área de dados (área global);

. Acoplamento por conteúdo

Um módulo referencia o conteúdo de outro módulo;

. Acoplamento de dados

A comunicação entre os módulos é feita através de parâmetros que podem ser um dado elementar ou uma tabela homogênea.

- Coesão

É a força que mantém unidos os elementos de um módulo.

A coesão está classificada em:

. Funcional

Os elementos estão relacionados ao desempenho de uma única função;

. Sequencial

A saída de um elemento é a entrada para o próximo;

. Comunicacional

Os elementos referenciam os mesmos dados de entrada e saída;

. Procedural

Os elementos estão envolvidos em atividades diferentes e não relacionadas;

. Temporal

Os elementos são executados ao mesmo tempo;

. Lógica

Existe alguma relação entre os elementos do módulo;

. Coincidencial

Aparentemente não há razão para os elementos de um módulo estarem juntos.

- Número de Módulos Subordinados

Também conhecido por "Fan-out". É o número de módulos que um módulo está chamando. Um projeto é considerado pobre se o "Fan-out" for muito alto ou muito baixo.

- Número de Módulos Superiores

Significa o número de módulos superiores que se utilizam de um módulo subordinado. Também é conhecido por "Fan-in". A presença do "Fan-in" indica que a duplicação de código está sendo evitada.

- Balanceamento

Os dados são tratados por módulos superiores de modo que não sejam dependentes das características físicas. Estas características são tratadas pelos módulos de níveis mais baixos.

A avaliação da modularidade segundo o aspecto lógico deve ser feita, considerando-se a:

- Complexidade

A presença da complexidade diminui a modularidade de

um programa.

- Abrangência

  Módulos devem ser flexíveis. A flexibilidade em módulo acontece quando eles não são muito restritos. Porém, a generalização dos módulos, para permitir a flexibilidade, deve ser cuidadosa porque o cógido pode se tornar extenso, criando pior acoplamento e pior coesão.

- Isolamento

  Através deste critério pode-se avaliar tudo o que se deve saber sobre a implementação do módulo. Com isto, torna-se possível a sua utilização corretamente.

- Memória de estado

  Um módulo quando devolve o controle para seu superior tende a morrer. Ao ser reativado, todo o processamento é feito como se fosse a primeira vez. Módulos que de alguma forma retêm informações passadas possuem a chamada memória de estado. A existência deste comportamento é um risco para a execução, porque os resultados obtidos de uma execução são imprevisíveis. Evitá-los é a melhor solução.

A avaliação da modularidade segundo o aspecto psicológico dá-se de acordo com os seguintes critérios:

- Tamanho do módulo

  Não há um rigor quanto ao tamanho ideal do módulo.

Alguns pesquisadores, dizem, por exemplo, que o módulo deve ter no máximo 50 comandos, YOURDON (18). Outros limitam o tamanho do módulo a uma página e assim por diante. Neste caso, o bom senso do programador deve funcionar, sempre tendo em mente que a legibilidade deve ser assegurada.

- Relato de erros

  Um módulo torna-se legível se ele próprio é capaz de tratar os erros que detecta. Sempre que um erro é detectado, o módulo deve decidir o que deverá ser feito, tal como: emitir a mensagem de erro correspondente, recuperar o erro para que a execução prossiga ou abortar a execução se o erro não for recuperável.

- Documentação

  Uma documentação adequada deve conter as seguintes informações:
  - . projeto físico;
  - . listagem do código fonte;
  - . alterações efetuadas;
  - . planejamento de testes

## II.3 - Técnicas de Avaliação da Qualidade de Software

Atualmente a grande preocupação em torno do desenvolvimento de software é com relação a sua qualidade. Várias técnicas de avaliação são usadas em ambientes de processamento de

dados de modo a garantir a sua presença.
A avaliação da qualidade não se dá somente a nível de código; abrange todas as fases do ciclo de vida do software.
Em DUNN (21) encontramos uma classificação para as técnicas mais usuais para a Avaliação da qualidade de software:

. Revisões de requisitos de software;
. Revisões de projetos;
. Processadores de pseudolinguagem;
. Revisões de código;
. Prova de correção;
. Testes estruturais;
. Testes Funcionais;
. Análise estática.

- Revisões de requisitos de software

São revisões minuciosas realizadas no documento que contém a especificação de requisitos de software. Inicialmente, estas revisões devem focalizar a rastreabilidade das especificações existentes, ou seja, deve-se observar se as especificações correspondem à expectativa desejada para a perfeita execução de suas tarefas, de modo a permitir um desempenho adequado ao produto.
Em segundo lugar, cada função definida na especificação de requisitos deve ser testável. Para tal, cada uma delas estar definida de forma completa e explícita.

- Revisões de projeto

Analogamente às revisões de requisitos, as revisões de

projeto devem focalizar a rastreabilidade. Em projeto de grande porte não basta uma simples revisão. Em geral, estes projetos são particionados e a revisão é aplicada a cada sub-projeto isoladamente.
Além da rastreabilidade devem ser verificados os seguintes atributos:

. A estruturabilidade do projeto
  Verifica se o projeto atende às normas de estruturação.
. A robustez
  verifica se o programa continuará executando mesmo na presença de violações dos resultados esperados.
. A testabilidade
  verifica se cada função é testável.
. Verifica se o programa é factível com as restrições de hardware, etc.

As revisões de projeto são também chamadas de Inspeções ou "Walk_throughs".

- Processadores de pseudolinguagens

As especifições de requisitos de software e os projetos podem ser documentados através de linguagens especiais que se aproximam das linguagens naturais e ,ao mesmo tempo, possuem características de linguagens de programação. Há nestas linguagens uma rigidez sintática e semântica, mas que permite um certo grau de liberdade na escrita, tornando o texto mais elucidativo. Além disso, o uso de

pseudolinguagens tende a gerar produtos de software estruturados.
A automatização de alguns aspectos das revisões de especificação de requisitos e do projeto são possíveis através destas linguagens. Os processadores para pseudolinguagens podem pesquisar inconsistências entre as seções, sequências impróprias dos passos de execução, anomalias de interfaces, entre outras coisas . Os problemas encontrados podem ser destacados através da emissão de relatórios.

- Revisões de código
O código implementa o projeto. A rastreabilidade do projeto é mais fácil de ser obtida se a correspondência entre os segmentos de código e a representação dos detalhes do projeto for unívoca. A codificação, entretanto, é mais uma fonte de erros que acontecem pelo mau uso dos comandos das linguagens de programação, pelos erros de sintaxe, etc. O objetivo das revisões de código é a detecção e correção destes erros.

- Prova de correção
É a técnica usada para demonstrar que um programa está correto. Esta técnica se fundamenta na teoria de lógica matemática, onde o programa é modelado de acordo com as formas matemáticas e, a partir delas, são determinadas as condições de correção.
Na prática, comumente utiliza-se o método de inclusões de assertivas indutivas em pontos chaves do código fonte. As

assertivas retratam as relações que existem entre as variáveis do programa e vão sendo testadas a partir do início de um módulo até a sua saída. Cada assertiva é validada considerando-se os resultados obtidos na avaliação das anteriores.

Geralmente, é um processo com alto consumo de tempo.

- Testes estruturais

  Neste tipo de teste cada detalhe do programa a ser testado é conhecido: o código, a estrutura de dados e a documentação do projeto.

  Teoricamente as informações da especificação de requisitos de software são irrelevantes.

  Os testes estruturais se preocupam com as operações ao longo do código fonte, considerando cada passo de execução e as decisões.

  Os testes estruturais mais usados atualmente implicam na instrumentação do programa. Estas instrumentações também são conhecidas por Analisadores dinâmicos de programa.

- Testes funcionais

  Ao contrário dos testes estruturais, os testes funcionais são programas de testes que se baseiam nas informações contidas nas especificações de requisitos de software.

  Através deles é possível se demonstrar que um programa desempenha bem cada função que lhe é atribuída.

  Na literatura técnica também encontramos a denominação "testes de caixa preta" para os testes funcionais.

- Análise Estática

Análise estática é o nome dado a um grupo de técnicas de software que servem, em geral, para detectar defeitos de estrutura e de semântica existentes no código fonte.

A análise estática em programas proporciona informações do tipo: variáveis não inicializadas, variáveis não referenciadas, trechos de programas não alcançáveis, etc. Um compilador, quando verifica a correção da sintaxe de comandos, realiza uma análise estática sobre o código fonte. Além destas verificações, a análise estática também pode fornecer outras informações quanto à qualidade dos programas, tais como: medida da complexidade, grau de modularidade e grau de simplicidade.

Esta técnica basicamente trabalha com dois conceitos fundamentais de otimização de código: grafo de fluxo e grafo de chamada.

O grafo de fluxo representa o fluxo de controle de um programa, onde cada nó identifica um ou mais comandos e os arcos identificam os passos de controle que ligam os comandos entre si. A figura (II.6) mostra um exemplo de grafo de fluxo.

O grafo de chamada, por sua vez, representa a comunicação entre as unidades dos programas, onde os nós identificam as funções, as subrotinas, etc. e os arcos identificam as chamadas de uma unidade por outra. A figura (II.7) mostra um exemplo de grafo de chamada.

Um programa pode estar bem escrito ou não. A experiência revela que os programas que não se enquadram segundo alguma norma de programação geram mais erros e se tornam de difícil compreensão, prejudicando, em consequência, a legibilidade. Em tal circunstância é necessário se analisar a sua estrutura sistematicamente, procurando identificar as anomalias ocorridas no decorrer do seu texto. O estudo do grafo de controle e da análise do fluxo de dados possibilita a detecção destas anomalias.

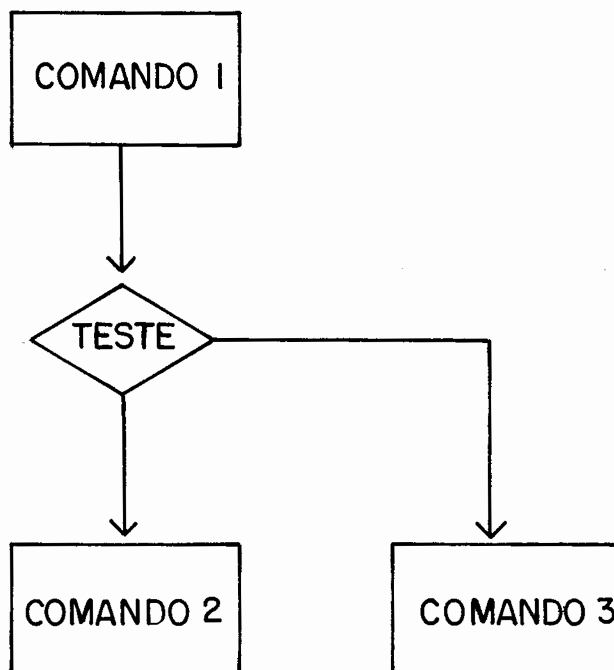



FIGURA II. 6 — GRAFO DE FLUXO

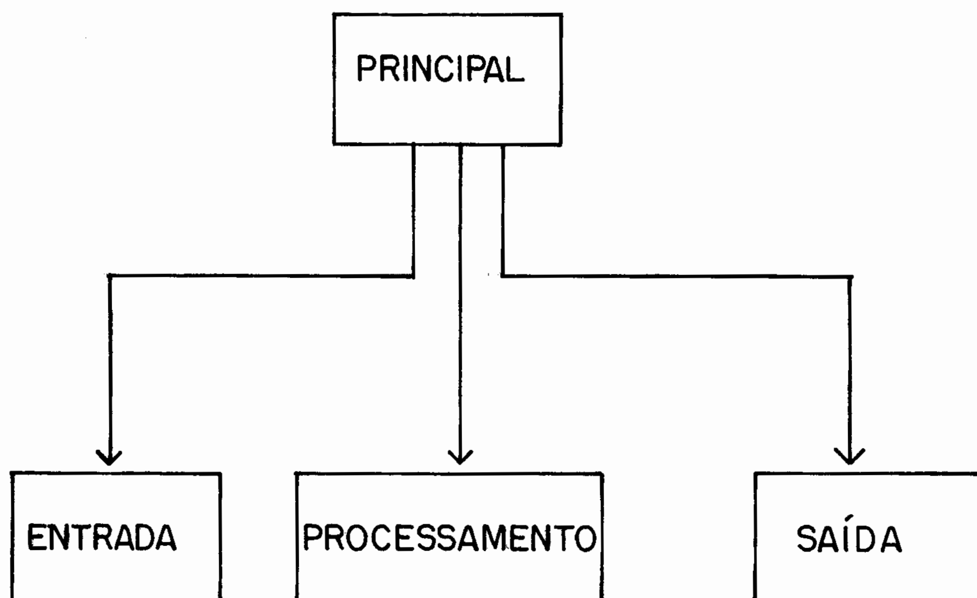



FIGURA II . 7 - GRAFO DE CHAMADA

Vários pesquisadores desenvolveram trabalhos a esse respeito. CARRÉ (22), por exemplo, destaca alguns tipos de anomalias de código:

(a) desvios para rótulos inexistentes;

(b) rótulos inalcançáveis;

(c) comandos inalcançáveis;

(d) comandos que não permitem se alcançar o fim.

HUANG (23), se preocupa com o comportamento das variáveis de programa. Para ele, uma variável de programa está sujeita a três ações diferentes, a saber: definição (d), referência (r) e indefinição (i). Uma variável é dita definida se a ela for atribuído algum valor. Uma variável é referenciada se o seu valor for obtido da memória. E a variável pode tornar-se indefinida se ela só tiver validade dentro de um bloco, ao sair dele, deixa de existir. Assim, durante a execução, uma variável pode ocupar um dos quatro estados conforme mostrado no diagrama da figura (II.8):

1 - indefinido;

2 - definido mas não referenciado;

3 - referenciado;

4 - anômalo.

Qualquer situação que leve ao estado quatro, apresenta uma anomalia.

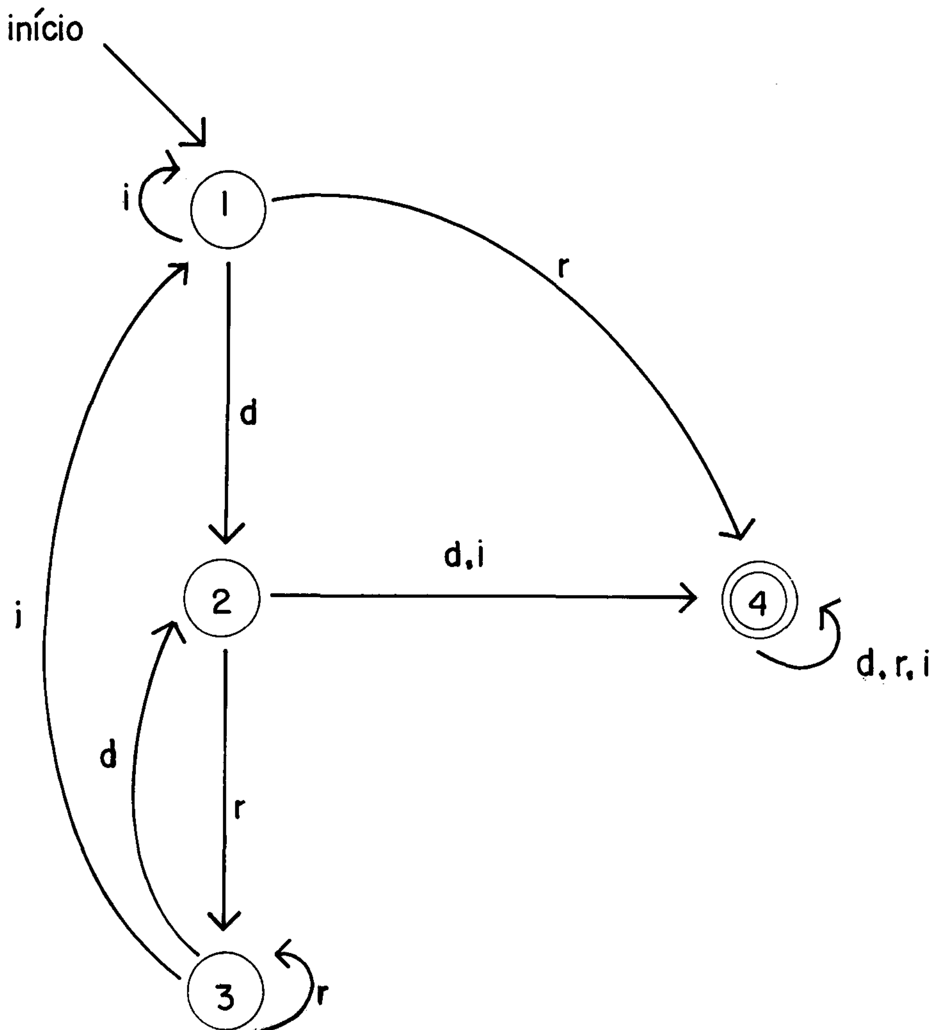



DIAGRAMA DE ESTADOS DAS VARIÁVEIS
Fonte: HUANG (23)

Nem todos concordam com a importância que a análise estática possui no processo de detecção de erros, DUNN (21). Para alguns trata-se de um simples estágio que antecede a análise dinâmica, tendo função de pré-processador. Para outros, as tarefas desenvolvidas por esta técnica podem ser supridas pelos compiladores e pelos ligadores. Tais afirmações seriam verdadeiras se todo e qualquer tipo de erro pudesse ser detectado por algum destes processos. O que ocorre na prática é que a análise estática tem se

mostrado necessária e eficiente no auxílio ao programador, tanto para indicar defeitos de código quanto para avaliar a qualidade dos programas.

## II.4 - Analisadores Estáticos de Programas

Embora a Análise Estática possa ser realizada manualmente, o desejo de reduzir o tédio e aumentar a produtividade do programador resultou na sua automatização, FAIRLEY (24), dando origem aos Analisadores Estáticos de Programas.

Os Analisadores Estáticos são ferramentas automatizadas de apoio aos programadores na repetitiva e cansativa tarefa de garantir a qualidade do programa desenvolvido. Nos últimos anos, um grande número de analisadores estáticos tem sido implementado. DAVE, FOSDICK (25) e FACES, RAMAMOORTHY (1), são exemplos de analisadores estáticos para o código FORTRAN. SELECT, BOYER (26), é um analisador estático para o código LISP.

Os analisadores estáticos não são ferramentas genéricas, isto é, não são aplicáveis a qualquer tipo de código. Os analisadores são específicos para cada linguagem de programação. Assim, um analisador estático para o código FORTRAN não é valido para o código COBOL.

## II.5 - Um Analisador Estático de Programas escritos em C

A linguagem C tem sido de uso bastante difundido, principalmente em ambientes de desenvolvimento de software básico de grande escala, por possuir características de linguagem de baixo nível e de alto nível ao mesmo tempo e por conter modernas estruturas de dados e de fluxos de controle. Infelizmente, são raras as ferramentas de apoio desenvolvidas para a linguagem C, por exemplo o Lint (31). Assim, o objetivo deste trabalho é propor um analisador estático de programas escritos na linguagem C, conforme está especificada em KERNINGHAN et alli (27), para auxiliar na avaliaçao das características de legibilidade de programas apresentados na seção II.2.1.

Este analisador vem compor o Ambiente de Apoio Automatizado para o Desenvolvimento de Software Básico, descrito no capítulo I.

No capítulo seguinte, veremos com mais detalhes a descrição desta ferramenta.

# CAPÍTULO III - AEP : O ANALISADOR ESTÁTICO

## III.1 Introdução

Nos últimos anos tem sido considerável o desenvolvimento de ferramentas automatizadas de apoio ao programador. São os editores de programas, os depuradores, os analisadores, etc. Entretanto, a maioria das ferramentas atende a uma linguagem de programação específica. Este capítulo descreve uma destas ferramentas de apoio, mais precisamente um analisador estático de programas, denominado AEP. Este analisador foi construído para auxiliar os programadores da linguagem C na avaliação da qualidade de seus produtos, notadamente no tocante à legibilidade. Na seção III.2 serão descritas as principais características da ferramenta e na seção III.3 será descrita a especificação de requisitos.

## III.2 - O AEP e suas características

O AEP é um analisador estático destinado a programas escritos na linguagem C. Como os demais analisadores estáticos, o AEP percorre o código de um programa em busca de possíveis anomalias. Entretanto, o que o torna diferente dos demais analisadores é que através dele também podem ser obtidas informações que verifiquem a legibilidade do programa. Estas informações estão baseadas em conceitos descritos no capítulo anterior. O AEP tem as seguintes características:

- É um analisador estático destinado a programas escritos na linguagem C, conforme está descrita em KERNINGHAN et alli (27);

- Através dele são pesquisadas anomalias de código, tais como:
  - . Variáveis e rotinas não referenciadas;
  - . Variáveis não definidas;
  - . Retornos de rotinas;
  - . Batimento de argumentos contra parâmetros, quanto a número e tipo;
  - . Atribuições envolvendo tipos diferentes;

- Para cada módulo (rotina) do programa, AEP ainda fornece:
  - . o seu tamanho;
  - . a sua complexidade;
  - . o número de módulos que o chamam ("Fan-in") e quais são os módulos;
  - . o número de módulos chamados por ele ("Fan-out") e quais são os módulos.

Além destas características, AEP permite a análise de todo o código fonte ou somente de módulos selecionados previamente.

É importante destacar que o AEP não é um compilador para a linguagem C, portanto os programas a serem analisados devem ser compilados anteriormente. AEP não faz críticas quanto à

correção da sintaxe. Parte-se do princípio que o programa não tem erro de compilação.

Na seção seguinte descreveremos a especificação de requisitos do AEP. Nesta fase do ciclo de vida do software deve ser fornecida uma especificação completa das funções desempenhadas pelo produto e as suas características de qualidade e desempenho, sem se prender a detalhes de implementação. Para tanto, este trabalho utilizará o modelo de especificação descrito em ANSI/IEEE (28).

## III.3 A Especificação de Requisitos do AEP

### III.3.1 Introdução

#### III.3.1.1 Escopo

O Analisador Estático de Programas para a linguagem C (AEP) é uma ferramenta que compõe o Ambiente de Apoio Automatizado para o desenvolvimento de Software Básico descrito em Santos (9). O seu objetivo é auxiliar os programadores da linguagem C a melhorarem o código de seus programas, a partir dos resultados produzidos.

Inicialmente, AEP verificará a legibilidade de programas através da dectecção de anomalias de código e de cálculos de medidas que avaliem alguns dos critérios que assegurem a sua presença.

AEP não está concluída totalmente. Futuramente será expandido para verificar outros atributos relacionados à qualidade dos programas.

#### III.3.1.2 Glossário

Anomalias:

Problemas encontrados nos códigos fontes dos programas que podem ou não causar erros durante à execução dos mesmos.

Biblioteca C:

Conjunto de rotinas de auxílio à programação em C. Estas rotinas implementam funções matemáticas, comparação de "strings", cópias de "strings", etc.

Ferramenta:

Programas de apoio ao programador/projetista durante o desenvolvimento de softwares.

Módulo:

Qualquer rotina (função) do programa.

### III.3.1.3 Referências


(1) McCABE, T. J., " A complexity Measure ", IEEE Transactions on Software Engineering, Vol. SE-2,4, 308-320, Dec. 1976.

(2) JUGNET, P., Introdução à Metodologia Orientada a Objetos, COBRA/DSO/SEFER, publicação interna, 1986.

(3) KERNINGHAN et alli, The C Programming Language, The Prentice-Hall, Inc., Englewood Cliffs, New Jersey.


## III.3.2 Descrição Geral

### III.3.2.1 Atributos do Produto

(a) **Legível**

Os módulos que comporão a ferramenta AEP deverão ser escritos de forma clara, para permitir a qualquer programador/projetista entender o seu código sem dificuldade.

(b) **Modificável**

Os códigos dos programas deverão ser simples e as documentações deverão corresponder à versão mais atualizada.

(c) **Expansível**

Inicialmente, AEP será desenvolvido para avaliar a legibilidade de programas escritos na linguagem C. Futuramente, poderá ser expandido para a avaliação de programas com relação a outros fatores que verifiquem a sua qualidade.

(d) **Portável**

AEP poderá ser executado em qualquer configuração que contenha o ambiente de software composto de um compilador C e da biblioteca C.

### III.3.2.2 Características do Usuário

É uma ferramenta destinada a usuários que saibam programar na linguagem C.

### III.3.2.3 Ambiente do Produto

Este produto será executável em qualquer ambiente que suporte programas escritos na linguagem C.

### III.3.2.4 Restrições Gerais

- AEP terá todo o seu código escrito em C.

- Os programas a serem avaliados devem ter sido compilados corretamente. A ferramenta não fará críticas quanto à sintaxe da linguagem;

- Inicialmente, AEP não tratará elementos de pré-processamento, ou seja: MACROS, Inclusão de arquivos e Compilação condicional;

- Os tipos declarativos permitidos para as variáveis são: declaração de inteiros, declaração de caracteres e de Ponto flutuante.

### III.3.2.5 Funções do Produto

AEP deverá executar as seguintes funções:

(a) Medir a Complexidade de Módulos

Através da aplicação da teoria desenvolvida por McCabe (1), para cada módulo AEP fornecerá um valor correspon-

dente à sua complexidad;

(b) Medir o Tamanho de Módulos

AEP fornecerá o tamanho de cada módulo do programa ou somente dos módulos selecionados pelo programador/projetista;

(c) Calcular o "Fan-in"

AEP calculará o número de módulos superiores ao módulo sendo analisado, especificando quais são estes módulos;

(d) Calcular o "Fan-out"

AEP calculará o número de módulos que estão subordinados ao módulo sendo analisado, especificando quais são estes módulos;

(e) Detectar Anomalias

Serão detectadas algus tipos de anomalias de código, tais como:

. Variáveis e rotinas não referenciadas;
. Variáveis não definidas;
. Retornos de rotinas;
. Batimento de argumentos contra parâmetros, quanto a número e tipo;
. Atribuições envolvendo tipos diferentes;

Os resultados serão discriminados de acordo com cada categoria quando da emissão do relatório de saída.

## III.3.3 Requisitos Específicos

### III.3.3.1 Requisitos de Atributos

(a) Legibilidade

Explicitar as particularidades existentes nos módulos através da documentação adequada.

(b) Modificabilidade

Utilizar estruturas de módulos manipuláveis. Evitar o uso de técnicas que dificultem a alteração de módulos.

(c) Expansibilidade

Implementar módulos que permitam a inclusão de novas funções.

(d) Portabilidade

Encapsular em módulos definidos as dependências de máquina geradas durante a implementação.

### III.3.3.2. Requisitos de Interfaces Externas

#### III.3.3.2.1. Interface Com o Usuário

##### III.3.3.2.1.1. Interface Interativa

AEP deverá ser ativado através de um comando de ativação. Durante a sua execução não haverá nenhuma forma de comunicação

interativa com o usuário. Ao final da execução será emitido um relatório de saída que fornecerá ao usuário as informações necessárias à análise de seu programa.

III.3.3.2.1.2. **Interface de Programação**

Não se aplica.

III.3.3.2.2. **Interface Com o Ambiente**

III.3.3.2.2.1. **Interface Com o Software**

Não se aplica.

III.3.3.2.2.2. **Interface Com o Hardware**

Não se aplica.

III.3.3.3. **Restrições de Desenho**

- Os módulos que comporão a ferramenta deverão atender à metodologia orientada a objetos (2).

- Todos os módulos serão escritos na linguagem C (3).

III.3.3.4. Requisitos **Funcionais**

(a) Medição da Complexidade.

Calcular a equação V(g) = e-n+2p, obtida da teoria de McCabe, onde e corresponde ao número ligações entre os nós do grafo, n corresponde ao número de nós do grafo e **p** corresponde ao número de componentes conexos (p.ex., rotinas) no grafo;

(b) Medição do Tamanho

Calcular o número total de linhas do módulo;

(c) Medição do "Fan-in"

Totalizar o número de módulos que chamam o módulo correntemente analisado, destacando-os;

(d) Medição do "Fan-out"

Totalizar o número de módulos que são chamados pelo módulo correntemente analisado, destacando-os;

(e) Dectecção de Anomalias

Utilizar algoritmos que percorram o texto fonte a procura de anomalias de código;

## III.3.4. Definição de Interfaces Externas

### III.3.4.1. Interface de Usuário

#### III.3.4.1.1. Interface Interativa

#### III.3.4.1.1.1. Comando de Ativação

AEP deverá ser ativado através do seguinte comando de ativação (para o COBRA-540):

:EX AEP PA

NO=nome_arq parâmetros-de-ativação

Onde:

**nome_arq**

Especificará o nome do arquivo de entrada que contém o programa fonte a ser analisado, e

**parâmetros-de-ativação**

É qualquer conjunto composto dos parâmetros que especificarão as funções a serem axecutadas pelo AEP, conforme descrição abaixo:

-A

Informa que deverão ser detectadas as anomalias encontradas no código;

-C

Informa que deverá ser calculada a complexidade dos módulos;

-T

Informa que deverá ser calculado o tamanho dos Módulos;

-f

Informa que o "Fan-in" de cada módulo deverá ser calculado;

- F

  Informa qu e o "Fan-out" de cada módulo deverá ser calculado;

-R nomes-rotina

  Informa que os demais parâmetros só serão verificados nos trechos de código contidos nas rotinas especificadas em **nomes-rotina**;

**nomes-rotina**

É o conjunto de rotinas selecionadas para análise. Os nomes das rotinas devem estar encerradas por abre e fecha chaves, separadas entre si por barra.

OBS:

(1) se o nome do arquivo de entrada não existir, o AEP deverá ser desativado automaticamente;

(2) Qualquer outro erro a nível de parâmetro de ativação também deverá causar a desativação do AEP;

(3) Com exceção do parâmetro -R, todos os demais terão a negativa correspondente. A negação será através da prefixação do caractere N em cada parâmetro de ativação desejado;

    ex: -NT (não calcula tamanho)

(4) Todos os parâmetros deverão ser obrigatoriamente prefixados por "-" sem espaço separador;

(5) Não deverá haver ordem exigida para os parâmetros de ativação;

(6) Não havendo parâmetros de ativação, ou na ausência de algum

deles, deverá ser assumido o seguinte:

- calcular o tamanho;
- calcular a complexidade;
- detectar anomalias;
- calcular o "Fan-in";
- calcular o "Fan-out";
- a análise será aplicada a todo o texto fonte.

### III.3.4.1.1.2. Relatório de Saída

O Relatório de Saída gerado pelo AEP deverá ter a seguinte composição:

1 - Cabeçalho;
2 - Lista de Parâmetros;
3 - Resultados da análise.

- Cabeçalho

A cada início de página do relatório, deverão ser listadas as seguintes informações:

. Nome do produto - AEP;
. Versão do produto, formada de uma letra e de dois dígitos decimais;
. Número da página corrente, formada por três dígitos decimais.

- **Lista de parâmetros**

Na primeira página do relatório de saída, após a impressão do cabeçalho, deverão ser impressos o nome do arquivo que contém o programa a ser analisado e as opções fornecidas pelo usuário no momento da ativação da ferramenta, no seguinte formato:

PARAMETROS DE ATIVACAO:

ARQUIVO DE ENTRADA: xxxxxxxx

OPCOES: xxx xxx xxx

- **Resultados da Análise**

O restante do relatório de saída deverá ser apresentado de acordo com a opção do usuário, ou seja:

Se a análise for realizada em todo o programa fonte, o relatório deverá estar esquematizado de acordo com o descrito em RELAT;

Se a análise for realizada somente em rotinas selecionadas do programa, o relatório deverá ter o mesmo conteúdo de RELAT, excetuando as informações das variáveis globais.

- **RELAT**

Sempre que se iniciar uma nova rotina do programa deverão ser impressas as seguintes informações:

```
*** ANOMALIAS DO MODULO: xxxxxxxx

* VARIAVEIS NAO DEFINIDAS

      NOME                LINHA DE DECLARACAO
      xxxxxxxx            xxx
      xxxxxxxx            xxx

* VARIAVEIS NAO REFERENCIADAS

      NOME                LINHA DE DECLARACAO
      xxxxxxxx            xxx
      xxxxxxxx            xxx

* PARAMETROS - ARGUMENTOS:
  ROTINA        NUM PAR       NUM ARG      LIN DECL    LIN OCOR
  xxxxxxxx      xx            xx           xxx         xxx
  xxxxxxxx      xx            xx           xxx         xxx

  POS PAR ARQ   TIPO PAR     TIPO ARG     LIN DECL     LIN OCOR
  xx            xxxxxxxx     xxxxxxxx     xxx          xxx
  xx            xxxxxxxx     xxxxxxxx     xxx          xxx

* ATRIBUICOES INCOMPATIVEIS:
      LINHA             TIPO A ESQUERDA          TIPO A DIREITA
      xxx               xxxxxxxx                 xxxxxxxx
      xxx               xxxxxxxx                 xxxxxxxx

* OUTRAS ANOMALIAS:
```

```
- lin xxx       xxx... mensagem ...xxx
- lin xxx       xxx... mensagem ...xxx


*** TAMANHOS DOS MODULOS:
* MODULO: xxxxxxxx          xxx linhas
* MODULO: xxxxxxxx          xxx linhas


*** COMPLEXIDADES DOS MODULOS:
* MODULO: xxxxxxxx          xxx
* MODULO: xxxxxxxx          xxx


*** MODULOS SUBORDINADOS A:
* MODULO: xxxxxxxx    TOTAL: xxx     MODULOS SUBORDINADOS:
                                     - xxxxxxxx
                                     - xxxxxxxx
                                     - xxxxxxxx


*** MODULOS SUPERIORES A:
* MODULO: xxxxxxxx    TOTAL: xxx     MODULOS SUPERIORES:
                                     - xxxxxxxx
                                     - xxxxxxxx
                                     - xxxxxxxx


*** ANOMALIAS DE VARIAVEIS GLOBAIS:
```

* VARIAVEIS NAO DEFINIDAS:

| NOME | LINHA DE DECLARACAO |
|---|---|
| XXXXXXXX | XXX |
| XXXXXXXX | XXX |

* VARIAVEIS NAO REFERENCIADAS:

| NOME | LINHA DE DECLARACAO |
|---|---|
| XXXXXXXX | XXX |
| XXXXXXXX | XXX |

Quando a análise for aplicada a módulos selecionados não deverão ser impressas as anomalias correspondentes às variáveis globais do programa.

Obs.: Para cada opção precedida de N (negação da opção), deverá ser inibida a impressão da saída correspondente no relatório.

**III.3.4.1.1.3. Comandos de Interação**

Não se Aplica.

**III.3.4.1.2. Interface de Programação**

Não se Aplica.

**III.3.4.2. Interface com o Ambiente**

**III.3.4.2.1. Interface com o Software**

Não se Aplica.

III.3.4.2.2. Interface com o Hardware

Não se Aplica.

# CAPÍTULO IV - O PROJETO AEP

## IV.1 Introdução

Este capítulo descreve a fase do ciclo de vida correspondente ao Desenho de Software. Esta fase deve fornecer mais detalhes quanto às especificações dos módulos existentes, tais como: Comunicação entre os módulos, "interfaces" das estruturas de dados, configuração das estruturas de controle e dados.

O modelo de especificação de Desenho de Software (EDS) utilizado neste trabalho está descrito em ANSI/IEEE (29). Para a descrição do projeto será utilizada a metodologia Orientada a Objetos, JUGNET (30).

## IV.2 Especificação do Desenho de Software do AEP.

### IV.2.1 Introdução

#### IV.2.1.1 Escopo

O conteúdo desta EDS é referente à primeira versão do projeto da ferramenta **Analisador Estático de Programas escritos em C** (AEP) . As funções do AEP foram descritas na Especificação de Requisitos de Software (ERS) apresentada no capítulo III.

Este documento aplica os conceitos relativos à metodologia de

desenvolvimento de projetos denominada "Metodologia Orientada a Objetos" |4|.

A seguir, serão apresentadas a descrição geral da arquitetura e as interfaces de cada módulo.

### IV.2.1.2 Definições e Normas de Batismo dos Arquivos

#### IV.2.1.2.1 Definições

Corpo de Objeto:

Descrição do modo de implementação de suas propriedades lógicas, bem como da estrutura de dados encapsulada.

Encapsulação:

Ação que consiste em "esconder" as propriedades físicas de uma estrutura de dados.

Interfaces de Objeto:

Concatenação das interfaces das funções que implementam suas propriedades lógicas.

Interfaces de Funções:

Especificações da declaração da função e de seus parâmetros

em conjunto com a descrição do que a função faz sem se ater ao modo de implementação.

Módulo:

Unidade de programação correspondente, em geral, a um objeto.

Objeto

Estrutura de dados "encapsulada" de tal forma que só pode ser manipulada através de suas propriedades lógicas.

Propriedade Lógica:

Propriedade abstrata de uma estrutura de dados, implementada por uma função só conhecida de seus usuários através de especificações de interface.

Relação entre Objetos:

São relações de utilização: se o objeto A utiliza as propriedades do objeto B, então A é um usuário de B.

IV.2.1.2.2 Normas de Batismo dos Arquivos

No ambiente SOD, onde o protótipo do AEP será implementado, os nomes dos arquivos são prefixados por um caractere alfabético

que especifica a natureza do arquivo. Assim, a lei de formação para a denominação de arquivos é a seguinte:

Cxxxxxxx: Arquivo de programas escritos em C;

Axxxxxxx: Arquivo de código "assembly";

Oxxxxxxx: Arquivo de código objeto;

Hxxxxxxx: Arquivo com as definições físicas de interface que deverão ser incluídas nos programas escritos em C que as utilizam.

### IV.2.1.3 Referências/Bibliografia

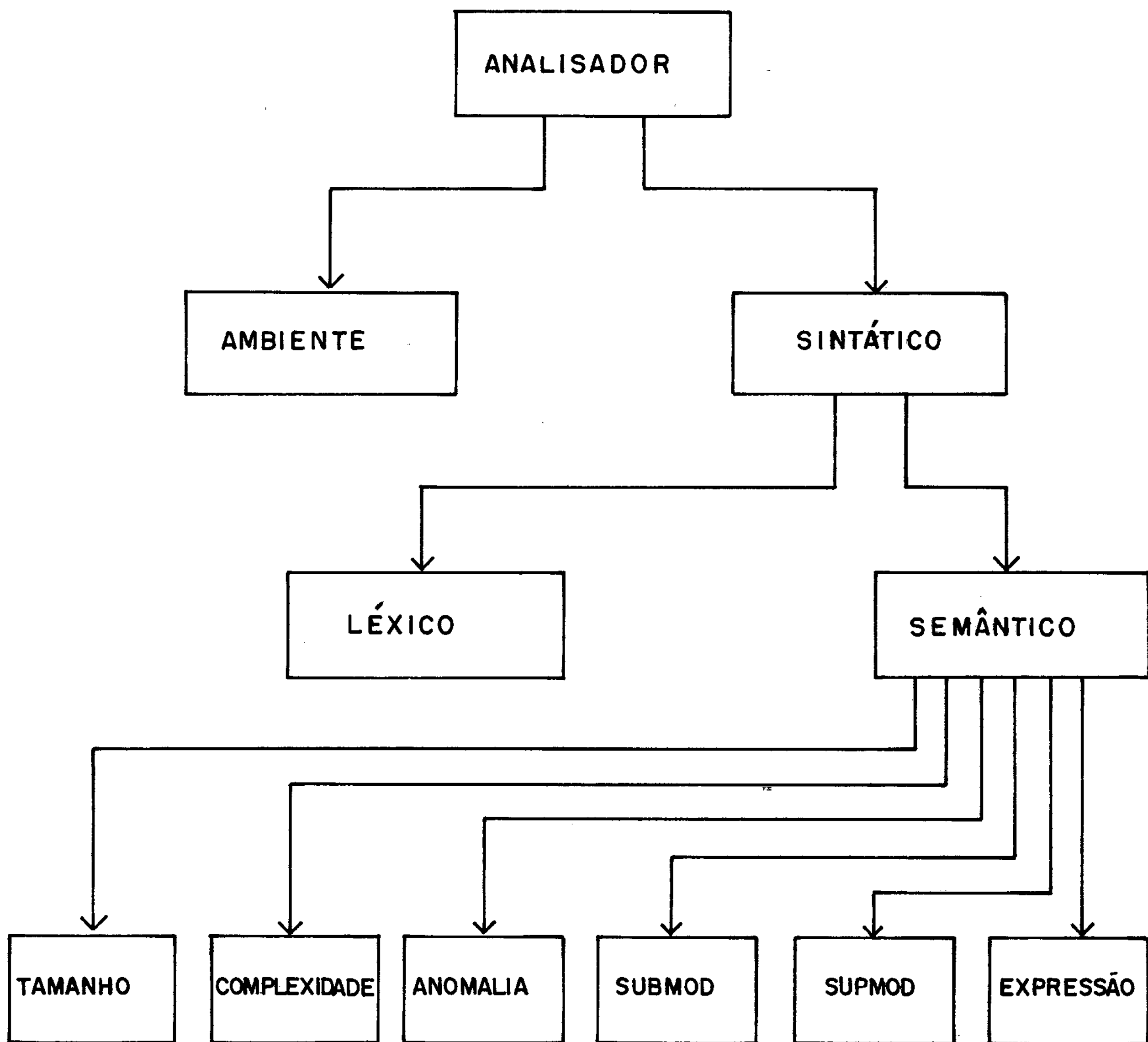
|1| PARNAS,D.L., "On the Criteria for Descomposing System into Modules", Comm. ACM, 15 (12), 1053-1058 (1972).

|2| LISKOV, B.H., "Programming With Abstract Data Types", Proce. ACM SIGPLAN Symp. on Very High Order Languages, SIGPLAN (1974).

|3| ROBINSON, L, "The Hierarquical Development Methology (HDM) Handbook", SRI Project 4828, Naval Ocean System Center, San Diego, California 95152 (1979).

|4| JUGNET, P., "Introdução à Metodologia Orientada a Objetos", COBRA/DSO/SEFER, Publicação Interna (1986).

IV.2.2 DESCRIÇÃO GERAL DA ARQUITETURA




| TABSIMB |
| --- |
| RELATÓRIO |
| ERRO |

ANALISADOR

Objeto responsável pelo controle da execução da ferramenta AEP. Os objetos que se encontram no nível abaixo são ativados por ele.

Na linguagem C, este objeto representa o programa principal - main () . É um objeto diferente dos demais por não possuir propriedade lógica associada.

AMBIENTE

Objeto responsável pela "inicialização" do ambiente para a execução da ferramenta AEP.

- Propriedades:
  - . Iniciar_ambiente

    Lê os parâmetros de ativação, validando-os. Inicializa as variáveis do ambiente de execução.

  - . Analisar_rotina

    Verifica se uma rotina deve ser analisada pela ferramenta.

SINTATICO

Objeto responsável pela análise sintática do programa fonte.

Os objetos LEXICO e SEMANTICO são ativados por este objeto.

- Propriedade:
  - . Analisar_sintaticamente

    Faz a análise sintática do programa fonte.

LEXICO

Objeto responsável pela análise léxica do program fonte.

- Propriedade:
  - . Ler_componente

    Identifica os elementos léxicos, classificando-os.

SEMANTICO

Objeto responsável pelo tratamento das ações semânticas.

- Propriedade:
  - . Chamar_acao_semantica

    Executa a ação semântica correspondente.

TAMANHO

Objeto responsável pelas informações relativas a tamanho de módulos.

- Propriedades:
  - . Calcular_tamanho_rotina

    Calcula o tamanho da rotina.

  - . Verificar_tamanho

    Verifica se o tamanho da rotina deve ser calculado.

COMPLEXIDADE

Objeto responsável pelas informações relativas à complexidade de módulos.

- Propriedades:
  - . Calcular_complexidade_modulo

    Calcula a complexidade de um módulo.

  - . Verificar_complexidade

Verifica se a complexidade deve ser calculada num módulo.

ANOMALIA

Objeto responsável pelas informações relativas às anomalias de código.

- Propriedades:

. Verificar_anomalia

Verifica se as anomalias de uma rotina devem ser detectadas.

. Armazenar_atribuição_incompatível

Armazena as informações necessárias para a impressão da anomalia correspondente à atribuição de tipos incompatíveis (lado esquerdo do sinal de atribuição diferente do lado direito do sinal de atribuição).

. Armazenar_parâmetro_argumento

Armazena as informações necessárias para a impressão da anomalia correspondente à diferença de número de parâmetros e número de argumentos de uma rotina.

. Armazenar_tipos_par_arg

Armazena as informações necessárias para a impressão da anomalia correspondente ao tipo do parâmetro diferente do tipo do argumento de uma rotina.

. Armazenar_outras_anomalias

Armazena as demais anomalias não classificadas acima.

SUBMOD

Objeto responsável pela identificação de módulos subordinados a um módulo.

- Propriedades:

. Calcular_mod_subordin

Calcula o número de rotinas chamadas por outra.

. Verificar_mod_subordin

Verifica se deve ser calculado o número de módulos subordinados.

SUPMOD

Objeto responsável pela identificação de módulos superiores a um módulo.

- Propriedades:

. Calcular_mod_superior

Calcula o número de rotinas que chamam uma mesma rotina.

. Verificar_mod_superior

Verifica se deve ser calculado o número de módulos superiores.

EXPRESSAO

Objeto responsável pelo tratamento das expressões.

- Propriedades:

. Armazenar_token_operador

Armazena o token correspondente a um operador de expressão.

. Armazenar_token_operando

Armazena o token correspondente a um operando de expressão.

. Finalizar_expressão

Conclui o armazenamento dos componentes de expressão.

. Resolver_expressão

A partir dos componentes armazenados pelas propriedades de armazenamento, calcula o resultado da expressão.

. Iniciar_expressão

Prepara o ambiente destinado ao recebimento do corpo da expressão.

. Desfazer_expressão

Ignora as informações recebidas como expressão.

TABSIMB

Objeto responsável pelo tratamento dos símbolos do programa.

- Propriedades:

  . Criar_tabela_símbolos

  Aloca área para tabela de símbolos.

. Liberar_tabela_símbolos

Libera área alocada para a tabela de símbolos.

. Inserir_símbolo

Insere símbolo na tabela de símbolos.

. Ler_símbolo

Lê um símbolo da tabela de símbolos.

. Verificar_símbolos_não_definidos

Verifica a existência de símbolos não definidos.

. Verificar_símbolos_não_referenciados

Verifica a existência de símbolos não referenciados.

. Atualizar_referência

Indica na tabela de símbolos que o símbolo foi referenciado.

. Atualizar_definição

Indica na tabela de símbolos que o símbolo foi definido.

. Atualizar_parâmetro

Atribui à rotina o tipo de seus parâmetros.

RELATORIO

Objeto responsável pela preparação e impressão do relatório

de saída.

- Propriedades:

  . Imprimir_relatório_inicial

  Imprime a parte inicial do relatório de saída.

  . Imprimir_relatório_rotina

  Imprime no relatório de saída o resultado da análise de uma rotina.

  . Imprimir_relatório_final

  Imprime no relatório de saída o resultado da análise do programa.

ERRO

Objeto responsável pelo tratamento de erro.

- Propriedade:

  . Tratar_erro

  Imprime mensagem de erro e aborta a execução da ferramenta.

Obs.:

Os objetos ERRO, RELATORIO E TABSIMB se encontram à parte do diagrama por serem chamados por objetos de diferentes níveis da arquitetura.

## IV.2.3 Especificações de Interfaces

OBJETO: ANALISADOR

IV.2.3.1 - CLASSE: Não há.

IV.2.3.1.1 - **NOME**

Programa_principal

IV.2.3.1.2 - **INTERFACES**

```
main (argc, argv)
int argc ;
char * argv | |;
```

IV.2.3.1.3 - **DESCRIÇÃO**

. Chamar o objeto AMBIENTE para "inicializar" o ambiente de execução do AEP;

. Chamar o objeto SINTATICO para realizar a análise sintática;

. Chamar o objeto RELATORIO para imprimir o relatório final de saída.

IV.2.3.1.4 - REFERÊNCIAS

Inic-ambiente (CAMBIENT), Sintatic (CSINTAT), Re-lat-final (CRELAT).

IV.2.3.1.5 - **VERSÃO** - 01

OBJETO: AMBIENTE

IV.2.3.2 - CLASSE: "Inicialização"

IV.2.3.2.1 - NOME

Iniciar_ambiente

IV.2.3.2.2 - INTERFACE

```
void Inic_ambiente (argc,argv)
int   argc;
char *argv | |;
```

IV.2.3.2.3 - DESCRIÇÃO

Analisar cada parâmetro de ativação, inicializando as variáveis globais correspondentes.

Chamar o objeto TABSIMB para criar ambiente para a tabela de símbolos.

Chamar Objeto RELATORIO para imprimir a página inicial do relatório de saída.

Qualquer erro ocorrido a nível dos parâmetros, chamar o objeto ERRO para abortar a execução do AEP.

IV.2.3.2.4 - REFERÊNCIAS

Aborta_aep (CERRO), Cria_tabsimb (CTABSIMB), Relat_inic (CRELAT), fopen (CBIB), strcpy (CBIB), strlen (CBIB), strncmp (CBIB).

IV.2.3.2.5 - VERSÃO - 01

OBJETO: AMBIENTE

IV.2.3.3 - CLASSE: Análise

IV.2.3.3.1 - **NOME**

Analisar_rotina

IV.2.3.3.2 - **INTERFACES**

```
#include "NO=HCONST"
int Analisa_rot (p1,p2)
char *p1, *p2;
```

p1 : ponteiro para o nome da rotina corrente;

p2 : ponteiro para a tabela com os nomes das rotinas a serem analisadas.

IV.2.3.3.3 - **DESCRIÇÃO**

Percorre a Tabela que contém os nomes das rotinas a serem analisadas, apontada pelo ponteiro *p2, comparando-as com o nome da rotina em análise, apontado por *p1. Se a rotina for encontrada na tabela, devolver VERO; caso contrário, FALSO.

IV.2.3.3.4 - **REFERÊNCIAS**

strncmp (CBIB), strlen (CBIB)

IV.2.3.3.5 - **VERSÃO**- 01

OBJETO: SINTATICO

IV.2.3.4 - CLASSE: Análise

IV.2.3.4.1 - **NOME**

Analisar_sintaticamente

IV.2.3.4.2 - **INTERFACE**

void Sintatic ( )

IV.2.3.4.3 - **DESCRIÇÃO**

Enquanto não for o fim do arquivo de entrada, chamar o objeto LEXICO para ler a próxima entrada.
Se o tipo da entrada da tabela sintática for Terminal, verificar se a entrada corrente corresponde à entrada da tabela sintática. Se verdade, chamar o objeto SEMANTICO para realizar a ação semântica correspondente. Ler a próxima entrada. Seguir para o estado seguinte (sucessor). Se falso, tentar outro estado.

Se o tipo da entrada da tabela sintática for Não Terminal, seguir para o estado cujo elemento léxico pertença ao conjunto que inicia este Não Terminal.

Se o tipo da entrada da tabela sintática for Fim de Produção, chamar o objeto SEMANTICO para realizar a ação semântica associada. Voltar para o estado que deu origem a esta produção.

Qualquer erro ocorrido na análise sintática, chamar o objeto

ERRO para reportar a mensagem de erro e abortar a execução.

IV.2.3.4.4 - REFERÊNCIAS

Chama_asem (CSEMANT), lexico (CLEXICO)

IV.2.3.4.5 - VERSÃO - 01

OBJETO: LEXICO

IV.2.3.5 - CLASSE: Leitura

IV.2.3.5.1 - NOME

Ler_componente

IV.2.3.5.2 - INTERFACES

```
#include "NO=H108TOK"
void lexico (p)
token_t *p;
```

p: ponteiro para a estrutura que contém o token a ser classificado.

IV.2.3.5.3 - DESCRIÇÃO

Lê caractere a caractere, separando os elementos léxicos. Classificar cada elemento obtido.

Qualquer erro léxico, chamar o objeto ERRO para emitir a mensagem e abortar a execução da ferramenta.

IV.2.3.5.4 - REFERÊNCIAS

Aborta_aep (CERRO).

IV.2.3.5.5 - VERSÃO - 01

OBJETO: SEMANTICO

IV.2.3.6 - CLASSE: Análise semântica

IV.2.3.6.1 - NOME

Chamar_acao_semantica

IV.2.3.6.2 - INTERFACES

```
#include "NO=H108TOK"
#include "NO=HINDSEM"
void Chama_asem (acao, p)
int acao;
token_t *p;
```

acao: índice para tabela de rotinas semânticas.

p : ponteiro para a estrutura que contém o elemento léxico classificado.

IV.2.3.6.3 - DESCRIÇÃO

Executa a ação semântica que se encontra na entrada da tabela de ações semânticas especificada em acao.

acao é um valor definido no arquivo HINDSEM.

IV.2.3.6.4 - REFERÊNCIAS

---

IV.2.3.6.5 - VERSÃO - 01

OBJETO: TAMANHO

IV.2.3.7 - CLASSE: Cálculo

IV.2.3.7.1 - NOME

Calcular_tamanho_rotina

IV.2.3.7.2 - INTERFACE

```
void Ctam_rot( )
```

IV.2.3.7.3 - DESCRIÇÃO

. Ctam_rot

Totaliza o número de linhas da rotina correntemente analisada.

IV.2.3.7.4 - REFERÊNCIAS

---

IV.2.3.7.5 - VERSÃO - 01

OBJETO: TAMANHO

IV.2.3.8 - CLASSE: Verificação

IV.2.3.8.1 - NOME

Verificar_tamanho

IV.2.3.8.2 - INTERFACE

```
#include "NO=HCONST"
int Ver_tam (p)
char *p;
```

p: ponteiro para o nome da rotina.

IV.2.3.8.3 - DESCRIÇÃO

. Ver_tam

Verifica se o tamanho da rotina especificada por p deve ser calculado. Esta rotina retorna VERO se verdade e FALSO caso contrário.

IV.2.3.8.4 - REFERÊNCIAS

---

IV.2.3.8.5 - VERSÃO - 01

OBJETO: COMPLEXIDADE

IV.2.3.9 - CLASSE: Cálculo

IV.2.3.9.1 - NOME

Calcular_complexidade_modulo

IV.2.3.9.2 - INTERFACE

```
void Ccomp_rot ( )
```

IV.2.3.9.3 - DESCRIÇÃO

. Ccomp_rot

Calcula a complexidade da rotina sendo analisada, através do cálculo da fórmula $V(G) = e - n + 2$, onde e significa o número de ligações entre os nós, n é o número de comandos.

IV.2.3.9.4 - REFERÊNCIAS

---

IV.2.3.9.5 - VERSÃO - 01

OBJETO: COMPLEXIDADE

IV.2.3.10 - CLASSE: Verificação

IV.2.3.10.1 - NOME

Verificar_complexidade

IV.2.3.10.2 - INTERFACE

```
#include "NO=HCONST"
int Ver_complex (p)
```

p: ponteiro para o nome da rotina.

IV.2.3.10.3 - DESCRIÇÃO

. Ver_complex

Verifica se a rotina especificada por p deve ter a complexidade calculada. Esta rotina retorna VERO, se verdade e FALSO, caso contrário.

IV.2.3.10.4 - REFERÊNCIAS

---

IV.2.3.10.5 - VERSÃO - 01

OBJETO: ANOMALIA

IV.2.3.11 - CLASSE: Verificação

IV.2.3.11.1 - NOME

Verificar_anomalia

IV.2.3.11.2 - INTERFACE

```
#include "NO=HCONST"
int Ver_anom (p)
char *p;
```

p: ponteiro para o nome da rotina.

IV.2.3.11.3 - DESCRIÇÃO

Verifica se a rotina especificada por p deve ter as anomalias detectadas. Esta rotina retorna VERO, se verdade e FALSO, caso contrário.

IV.2.3.11.4 - REFERÊNCIAS

Analisa_rot (CAMBIENT)

IV.2.3.11.5 - VERSÃO - 01

OBJETO: ANOMALIA

IV.2.3.12 - CLASSE: Armazenamento

IV.2.3.12.1 - NOMES

armazenar_atribuição_incompatível

armazenar_parâmetro_argumento

armazenar_tipos_par_arg

armazenar_outras_anomalias

IV.2.3.12.2 - INTERFACES

```
(01) #include "NO=HCONST"
     void Arm_atrinc (te, td)
     int te, td;

(02) #include "NO=HTABSIMB"
     void Arm_argpar (psimb, num_arg)
     TABSIMB *psimb;
     int      num_arg;

(03) #include "NO=HCONST"
     void Arm_tippa (posição, tipo_par, tipo_arg, lin_decl)
```

```
    char posição, tipo_param, tipo_argum;
    int lin_decl;

(04) void Arm_outanom (ind_msg)
    int ind_msg;
```

te: valor correspondente ao tipo da esquerda de uma atribuição. Deve ser uma das seguintes macros: CHAR, INT, LONG, FLOAT ou DOUBLE.

td: valor correspondente ao tipo da direita de uma atribuição. Deve ser uma das seguintes macros: CHAR, INT, LONG, FLOAT ou DOUBLE.

psimb : ponteiro para a entrada na tabela de símbolo.

num_arg: valor correspondente ao número de argumentos.

lin_decl: valor correspondente ao número da linha de declaração da rotina.

posicao: valor correspondente à posição do param/argum da rotina.

tipo_par: valor correspondente ao tipo do parâmetro. Deve ser uma das seguintes macros: CHAR, INT, LONG, FLOAT ou DOUBLE.

tipo_arg: valor correspondente ao tipo do argumento. Deve ser uma das seguintes macros: CHAR, INT, LONG, FLOAT ou DOUBLE.

ind_msg: valor correspondente ao índice da mensagem a ser impressa no relatório de saída.

IV.2.3.12.3 - DESCRIÇÃO

. Arm_atrinc

Atribui aos campos da estrutura global **Atrib_inc** os valores passados nos parâmetros. Totaliza o número de atribuições incompatíveis.

. Arm_argpar

Atribui aos campos da estrutura global **Num_par** os valores passados nos parâmetros. Totaliza o número de anomalias deste tipo.

. Arm_tippa

Atribui aos campos da estrutura global **Tipo_pa** os valores passados nos parâmetros. Totaliza o número de anomalias deste tipo.

. Arm_outanom

Atribui aos campos da estrutura global **Out_anom** os valores passados nos parâmetros. Totaliza o número de anomalias deste tipo.

IV.2.3.12.4 - REFERÊNCIAS

\---

IV.2.3.12.5 - VERSÃO - 01

OBJETO: SUBMOD

IV.2.3.13 - CLASSE: Cálculo

IV.2.3.13.1 - NOME

Calcular_mod_subordin

IV.2.3.13.2 - INTERFACE

Void Cmod_sub ( )

IV.2.3.13.3 - DESCRIÇÃO

Calcula o número de rotinas que estão sendo chamadas pela rotina correntemente analisada, armazenando o nome de cada uma delas na estrutura de dados global Msb para serem relacionadas no relatório de saída.

IV.2.3.13.4 - REFERÊNCIAS

---

IV.2.3.13.5 - VERSÃO - 01

OBJETO: SUBMOD

IV.2.3.14 - CLASSE: Verificação

IV.2.3.14.1 - NOME

Verificar_mod_subordin

IV.2.3.14.2 - INTERFACE

int Ver_fanout (p)

char *p;

p: ponteiro para o nome da rotina.

IV.2.3.14.3 - DESCRIÇÃO

Verifica se o número de rotinas chamadas pela rotina especificada por p deve ser calculado.

IV.2.3.14.4 - REFERÊNCIAS

---

IV.2.3.14.5 - VERSÃO - 01

OBJETO: SUPMOD

IV.2.3.15 - CLASSE: Cálculo

IV.2.3.15.1 - NOME

Calcular_mod_superior

IV.2.3.15.2 - INTERFACE

Void Cmod_sup (p)

char *p;

p: ponteiro para o nome da rotina.

IV.2.3.15.3 - DESCRIÇÃO

Calcula o número de rotinas que chamam a rotina especificada

pelo parâmetro p. Armazena o nome de cada rotina que a está chamando na estrutura global Msp.

IV.2.3.15.4 - REFERÊNCIAS

---

IV.2.3.15.5 - VERSÃO - 01

OBJETO: SUBMOD

IV.2.3.16 - CLASSE: **Verificação**

IV.2.3.16.1 - **NOME**

Verificar_mod_subordin

IV.2.3.16.2 - **INTERFACE**

```
int Ver_fanin (p)
char *p;
```

p: ponteiro para o nome da rotina.

IV.2.3.16.3 - **DESCRIÇÃO**

Verifica se o número de rotinas que chamam a rotina especificada pelo parâmetro p deve ser calculado.

IV.2.3.16.4 - REFERÊNCIAS

---

## IV.2.3.16.5 - VERSÃO - 01

OBJETO: EXPRESSAO

## IV.2.3.17 - CLASSE: Armazenamento

### IV.2.3.17.1 - NOMES

Armazenar_token_operador

Armazenar_token_operando

### IV.2.3.17.2 - INTERFACES

```
(01) #include "NO=H108TOK"
     void Arm_tk_expr (p)
     tokent_t *p;

(02) #include "NO=HCONST"
     void Arm_tk_opndo (tipo)
     int tipo;
```

p: ponteiro para o token da expressão;

tipo: tipo do operando de uma expressão (CHAR, INT, LONG, FLOAT, DOUBLE).

### IV.2.3.17.3 - DESCRIÇÃO

. Arm_tk_expr

Armazena o token do operador especificado por **p** na forma pós-fixa.

. Arm_tk_opndo

Armazena o tipo do operando especificado no parâmetro **tipo** na forma pós-fixa.

IV.2.3.17.4 - REFERÊNCIAS

---

IV.2.3.17.5 - VERSÃO - 01

OBJETO: EXPRESSÃO

IV.2.3.18 - CLASSE: Finalização

IV.2.3.18.1 - NOME

Finalizar_expressão

IV.2.3.18.2 - INTERFACES

void Fim_expr ( )

IV.2.3.18.3 - DESCRIÇÃO

Conclui o armazenamento da expressão na forma pós-fixa.

IV.2.3.18.4 - REFERÊNCIAS

---

IV.2.3.18.5 - VERSÃO - 01

OBJETO: EXPRESSAO

IV.2.3.19 - CLASSE: Cálculo

IV.2.3.19.1 - NOME

Resolver_expressão

IV.2.3.19.2 - INTERFACE

```
#include "NO=HCONST"
int Res_expr ( )
```

IV. 2.3.19.3 - DESCRIÇÃO

Calcula o tipo resultante da expressão. Esta rotina devolve o tipo do resultado da expressão que assume um dos seguintes valores: CHAR, INT, LONG, FLOAT ou DOUBLE.

IV.2.3.19.4 - REFERÊNCIAS

---

IV.2.3.19.5 - VERSÃO - 01

OBJETO: EXPRESSÃO

IV.2.3.20 - CLASSE: Inicialização

IV.2.3.20.1 - NOME

Iniciar_expressão

IV.2.3.20.2 - INTERFACE

void Inic_expr ( )

IV.2.3.20.3 - DESCRIÇÃO

"Inicializa" o ambiente para receber o corpo de uma expressão.

IV.2.3.20.4 - REFERÊNCIAS

---

IV.2.3.20.5 - VERSÃO - 01

OBJETO: EXPRESSÃO

IV.2.3.21 - CLASSE: Negação

IV.2.3.21.1 - NOME

Desfazer_expressão

IV.2.3.21.2 - INTERFACE

void Desfaz_expr ( )

IV.2.3.21.3 - DESCRIÇÃO

Retorna o ambiente de expressão ao estado em que estava antes de considerar os tokens recém_armazenados como de expressão.

IV.2.3.21.4 - REFERÊNCIAS

---

IV.2.3.21.5 - VERSÃO - 01

OBJETO: TABSIMB

IV.2.3.22 - CLASSE: Inserção

IV.2.3.22.1 - NOME

Inserir_símbolo

IV.2.3.22.2 - INTERFACES

```
#include "NO=HCONST"
#include "NO=HTABSIMB"
int Ins_simb (p)
TABSIMB *p;
```

p : ponteiro para a estrutura que contém o símbolo e atributos.

IV.2.3.22.3 - DESCRIÇÃO

Insere símbolo com atributos na tabela de símbolos.

Esta rotina retorna OK se a inserção for realizada com sucesso e retorna FALH, em caso contrário.

IV.2.3.22.4 - REFERÊNCIAS

hpesq, memcpy (CBIB)

IV.2.3.22.5 - VERSÃO - 01

OBJETO: TABSIMB

IV.2.3.23 - CLASSE: Criação

IV.2.3.23.1 - NOME

Criar_tabela_símbolos

IV.2.3.23.2 - INTERFACE

void Cria_tabsimb ( )

IV.2.3.23.3 - DESCRIÇÃO

Para cada bloco de comandos aberto, aloca uma área de memória para armazenar os símbolos declarados neste bloco.

IV.2.3.23.4 - REFERÊNCIAS

exit (CBIB), hcria

IV. 2.3.23.5 - VERSÃO - 01

OBJETO: TABSIMB

IV.2.3.24 - CLASSE: Liberação

IV.2.3.24.1 - NOME

Liberar_tabela_símbolos

IV.2.3.24.2 - INTERFACE

```
void Lib_tabsimb ( )
```

IV.2.3.24.3 - DESCRIÇÃO

Para cada bloco de comandos fechado, libera a área alocada pela Cria_tabsimb.

IV.2.3.24.4 - REFERÊNCIAS

hdestroi, free (CBIB)

IV.2.3.24.5 - VERSÃO - 01

OBJETO: TABSIMB

IV.2.3.25 - CLASSE: Leitura

IV.2.3.25.1 - Nome

Ler_símbolo

IV.2.3.25.2 - INTERFACE

```
#include "NO=HTABSIMB"
TABSIMB *Le_htabsimb (p)
char * p;
```

p: ponteiro para o nome do símbolo a ser pesquisado na tabela de símbolos.

IV.2.3.25.3 - DESCRIÇÃO

Acessa a tabela de símbolos desde o nível de blocos corrente

até o nível zero, pesquisando o símbolo apontado por p. Esta pesquisa é feita por "hashing".

Esta rotina retorna um ponteiro para a estrutura do tipo TABSIMB que contém o símbolo e atributos ou zero caso o símbolo não seja encontrado.

IV.2.3.25.4 - REFERÊNCIAS

hpesq

IV.2.3.25.5 - VERSÃO - 01

OBJETO: TABSIMB

IV.2.3.26 - CLASSE: Verificação

IV.2.3.26.1 - NOMES

Verificar_símbolos_não_definidos

Verificar_símbolos_não_referenciados

IV.2.3.26.2 - INTEFACES

(01) void Ver_defsimb ( )

(02) void Ver_refsimb ( )

IV.2.3.26.3 - DESCRIÇÃO

. Ver_defsimb

Percorre a tabela de símbolos a procura de símbolos não definidos.

Para cada ocorrência de símbolo não definido, armazenar o

nome do símbolo e o número da linha de declaração na estrutura global **Var_ndef**. Totalizar o número de ocorrências.

. Ver_refsimb

Percorre a tabela de símbolos a procura de símbolos não referenciados.

Para cada ocorrência de símbolo não referenciado, armazenar o nome do símbolo e o número da linha de declaração na estrutura global **Var_nutil**. Totalizar o número de ocorrências.

IV.2.3.26.4 - REFERÊNCIAS

---

IV.2.3.26.5 - **VERSÃO** - 01

OBJETO: TABSIMB

IV.2.3.27 - CLASSE: **ATUALIZAÇÃO**

IV.2.3.27.1 - **NOMES**

Atualizar_referência

Atualizar_definição

Atualizar_parâmetro

IV.2.3.27.2 - **INTERFACES**

(01) #include "NO=HTABSIMB"

```
     TABSIMB *Atual_refsimb (p)
     char *p;

(02) #include "NO=HTABSIMB"
     TABSIMB *Atual_defsimb (p)
     char *p;

(03) #include "NO=HCONST"
     void Atual_par (p_rot, tipo, num_par)
     char *p_rot;
     int tipo;
     int num_par;
```

p: ponteiro para o nome do símbolo.;

p_rot: ponteiro para o nome da rotina;

tipo: tipo do parâmetro.

Deve ser uma das seguintes macros: CHAR, INT, LONG, FLOAT ou DOUBLE.

num_par: índice correspondente à posição do parâmetro na declaração da rotina.

IV.2.3.27.3 - DESCRIÇÃO

. Atual_refsimb

Liga o campo da tabela de símbolos correspondente, indicando que o símbolo foi referenciado. Esta rotina retorna um ponteiro para a estrutura do tipo TABSIMB ou zero caso o símbolo não tenha sido encontrado.

. Atual_defsimb

Liga o campo da tabela de símbolos correspondente, indicando que o símbolo foi definido. Esta rotina retorna um ponteiro para a estrutura do tipo TABSIMB ou zero caso o símbolo não tenha sido encontrado.

. Atual_par
Coloca o tipo do parâmetro na posição indicada por num_par na entrada da tabela de símbolo da rotina especificada por p_rot.

IV.2.3.27.4 - REFERÊNCIAS

---

IV.2.3.27.5 - VERSÃO - 01

OBJETO: RELATORIO

IV.2.3.28 - CLASSE: IMPRESSÃO

IV.2.3.28.1 - NOMES:

Imprimir_relatório_inicial
Imprimir_relatório_rotina
Imprimir_relatório_final

IV.2.3.28.2 - INTERFACES

(01) void Relat_inic (num_opções)

```
int num_opções;

(02) void Relat_rot ( )

(03) void Relat_final ( )
```

num_opções: número de opções fornecidas nos parâmetros de ativação.

IV.2.3.28.3 DESCRIÇÃO

. Relat_inic

Imprime a página inicial do relatório de saída de acordo com o formato especificado na ERS (relatório de saída).

. Relat_rot

Imprime os resultados obtidos da análise da rotina de acordo com o formato especificado na ERS (relatório de saída). As saídas serão fornecidas respeitando-se as opções fornecidas pelo usuário.

. Relat_final

Imprime os resultados obtidos da análise geral do programa.

IV.2.3.28.4 REFERÊNCIAS

---

IV.2.3.28.5 VERSÃO - 01.

OBJETO: ERRO

IV.2.3.29 - CLASSE: Tratamento

IV.2.3.29.1 - NOME

Tratar_erro

IV.2.3.29.2 - INTERFACE

void Aborta_aep (num_erro)

int num_erro;

num_erro: número do erro

IV.2.3.29.3 - DESCRIÇÃO

Emite a mensagem de erro especificada pelo parâmetro num_erro e aborta a execução.

IV.2.3.29.4 - REFERÊNCIAS

exit (CBIB)

IV.2.3.29.5 - VERSÃO - 01

## IV.2.4 Descrição Detalhada das Estruturas de Dados

A seguir será apresentada a descrição detalhada das estruturas de dados usadas na programação do AEP.

### IV.2.4.1 Arquivo de interface HCONST

```
/* Constantes de uso geral                    */

#define FALH          -1
#define FALSO          0
#define OK             0
#define TERRA         -1
#define VERO           1
#define UM             1
#define ZERO           0

#define NUM_BLOCO     20
#define NUM_OPCAO      8
#define NUM_ROT       20
#define NUM_SIMB      20
#define TS             9

/* Constantes de classe                       */

#define EXTERNO        1
#define ESTATICO       2

/* Constantes de tipo                         */
```

```
#define VOID        1
#define CHAR        2
#define INT         3
#define LONG        4
#define FLOAT       5
#define DOUBLE      6
#define CADEIA      7

/* Constantes de especificacao de identificadores */

#define PARM        1
#define ROTINA      2
```

## IV.2.4.2 - ARQUIVO DE INTERFACE H108TOK

```
/*
 *     H108TOK  - Interface do modulo tokens:
 *                - tipos ctoken_t e token_t
 *                - macros para acesso a estes tipos
 *
 */

#ifndef H_TOK
#define H_TOK

/*   definicao dos tipos    */

typedef int      ctoken_t;

typedef struct  ctipo_t      tipo_cte;
                union          long      cte_int;
                               ldouble   cte_fl;
                               cte_val;
                cte_t;

typedef struct  ctoken_t     cl_tk;
                union          char      *p_tk;
                               opdor_t   op_tk;
                               cte_t     cte_tk;
                               un_tk;
                token_t;
```

```
/*   definicao das macros dos tokens   */

#define TK_ID          1
#define TK_IDTP        2
#define TK_IDROT       3
#define TK_CADEIA      4
#define TK_CTE         5
#define TK_ACHAVE      6
#define TK_FCHAVE      7
#define TK_APAREN      8
#define TK_FPAREN      9
#define TK_ACOLCH     10
#define TK_FCOLCH     11
#define TK_VIRGULA    12
#define TK_PTOVIRG    13
#define TK_RETIC      14
#define TK_OPINC      15
#define TK_OPUN       16
#define TK_OPEST      17
#define TK_OPSEMIG    18
#define TK_OPBIN      19
#define TK_IGUAL      20
#define TK_INTERR     21
#define TK_2PONTOS    22
#define TK_OPUB       23
#define TK_MULT       24
#define TK_AUTO       25
#define TK_BREAK      26
#define TK_CASE       27
```

```
#define TK_CHAR       28
#define TK_CONST      29
#define TK_CONTINUE   30
#define TK_DEFAULT    31
#define TK_DO         32
#define TK_DOUBLE     33
#define TK_ELSE       34
#define TK_ENUM       35
#define TK_EXTERN     36
#define TK_FLOAT      37
#define TK_FOR        38
#define TK_GOTO       39
#define TK_IF         40
#define TK_INT        41
#define TK_LONG       42
#define TK_REGISTER   43
#define TK_RETURN     44
#define TK_SHORT      45
#define TK_SIGNED     46
#define TK_SIZEOF     47
#define TK_STATIC     48
#define TK_STRUCT     49
#define TK_SWITCH     50
#define TK_TYPEDEF    51
#define TK_UNION      52
#define TK_UNSIGNED   53
#define TK_VOID       54
#define TK_VOLATILE   55
#define TK_WHILE      56
```

```
#define TK_EOF        57

#define NTOKENS       84
#define TK_NENHUM     95

/*   definicao das macros/rotinas de acesso ao token    */

#define  clas_tk(t)       ((t) -  cl_tk)
#define  opdor_tk(t)      ((t) -  un_tk.op_tk)
#define  lex_tk(t)        ((t) -  un_tk.p_tk)
#define  ctipo_tk(t)      ((t) -  un_tk.cte_tk.tipo_cte)
#define  ctei_tk(t)       ((t) -  un_tk.cte_tk.cte_val.cte_int)
#define  ctef_tk(t)       ((t) -  un_tk.cte_tk.cte_val.cte_fl)

export token_t *muda_rot ();  /* retorna token com classe
                                TK_IDROT                 */

export token_t *muda_tp ();   /*  retorna token com classe
                                  TK_IDTP                */

export   void    copia_token (); /*  Copia token          */
#endif
```

## IV.2.4.3 - ARQUIVO DE INTERFACE HTABSIMB

```
/*
 * Definicao da entrada da tabela de simbolos interna do
 * utilitario.
 */

typedef union
struct princ
  char  ts_nome |9| ; /* Nome do simbolo                  */
  char  ts_clas     ; /* Classe do simbolo                */
  char  ts_tip      ; /* Tipo                             */
  char  ts_esp      ; /* Especificacao do simbolo         */
  char  ts_ref      ; /* Indicativo de simbolo
                               referenciado               */
  char  ts_def      ; /* Indicativo de simbolo definido   */
  char  ts_nlin     ; /* Numero da linha no programa      */
  char  ts_naux     ; /* Numero de entradas auxiliares    */
 principal ;          /* Entrada principal                */
 struct aux
  char  ts_npar     ; /* Numero de parametros             */
  char  ts_par |10| ; /* Tipo dos parametros              */
 auxiliar ;           /* Entrada auxiliar                 */
TABSIMB ;


/*
 * Macros para acesso aos campos da tabela de simbolos
 */
```

```
#define   TS_NOME(x)      x -  principal.ts_nome
#define   TS_CLAS(x)      x -  principal.ts_clas
#define   TS_TIP(x)       x -  principal.ts_tip
#define   TS_ESP(x)       x -  principal.ts_esp
#define   TS_REF(x)       x -  principal.ts_ref
#define   TS_DEF(x)       x -  principal.ts_def
#define   TS_NLIN(x)      x -  principal.ts_nlin
#define   TS_NAUX(x)      x -  principal.ts_naux
#define   TS_NPAR(x)      x -  auxiliar.ts_npar
#define   TS_PAR(x,y)     x -  auxiliar.ts_par |y|

/*
 * Rotinas exportadas do modulo
 */

extern void       Atual_par     ( ) ;
extern void       Cria_tabsimb  ( ) ;
extern void       Lib_tabsimb   ( ) ;
extern void       Lis_tabsimb   ( ) ;
extern void       Ver_defsimb   ( ) ;
extern void       Ver_refsimb   ( ) ;
extern int        Ins_simbolo   ( ) ;
extern TABSIMB * Atual_defsimb ( ) ;
extern TABSIMB * Atual_refsimb ( ) ;
extern TABSIMB * Le_htabsimb   ( ) ;
```

## IV.2.4.4 - ARQUIVO DE DECLARACOES DE ESTRUTURAS DE DADOS GLOBAIS

```
/* Declaracao das estruturas                                */


struct ai              /* Estrut. de atributos incompativ */
beg
  int ai_linha;        /* linha da ocorrencia             */
  char *ai_tipesq;     /* tipo esquerdo do sinal de atrib */
  char *ai_tipdir;     /* tipo direito do sinal de atrib  */
end Atrib_inc  NUM_ROT ;
int Tot_ai = TERRA;


struct tpa             /* Estrutura para tipos de par/arg */
beg
  char tpa_pos;        /* posicao do parametro/argumento  */
  char *tpa_par;       /* tipo do parametro               */
  char *tpa_arg;       /* tipo do argumento               */
```

```
  int  tpa_ldecl;      /* linha de declaracao              */

  int  tpa_locor;      /* linha de ocorrencia              */

end Tipo_pa  NUM_ROT ;

int Tot_tpa = TERRA;


struct npa             /* Estrutura para num. par/arg     */
beg

  char npa_rot  TS ;  /* nome da rotina                   */

  int  npa_par;        /* numero de parametros             */

  int  npa_arg;        /* numero de argumentos             */

  int  npa_ldecl;      /* linha de declaracao              */

  int  npa_locor;      /* linha de ocorrencia              */

end Num_par  NUM_ROT ;

int Tot_npa = TERRA;


struct vnd             /* Estrut. de simb nao definidos  */
beg
```

```
  char vnd_nome  TS ; /* nome do simbolo                */

  int  vnd_lin;       /* linha de declaracao            */

end Var_ndef  NUM_SIMB ;

int Tot_vnd = TERRA;


struct vnu            /* Estrut. de simb nao utilizados */
beg

  char vnu_nome  TS ; /* nome do simbolo                */

  int  vnu_lin;       /* linha de declaracao            */

end Var_nutil  NUM_SIMB ;

int Tot_vnu = TERRA;


struct oa             /* Estrut. de outras anomalias    */
beg

  int  oa_lin;        /* numero da linha                */

  char *oa_msg;       /* mensagem da anomalia           */
```

```
end Out_anom  10 ;

int Tot_oa = TERRA;


struct beg              /* Estrut. de complexidade de rot */

   char cnome  TS ;     /* nome da rotina                 */

   int  cval;           /* valor da complexidade          */

end Comp_rot  NUM_ROT ;


struct beg              /* Estrut. de tamanho de rotinas  */

   char tnome  TS ;     /* nome da rotina                 */

   int  tlin;           /* numero da linha de declaracao  */

end Tam_rot  NUM_ROT ;


struct beg              /* Estrutura para o Fan-out       */

   char sbnome  TS ;    /* nome da rotina                 */

   int  sbn;            /* numero de rotinas chamadas     */
```

```
   char * sbmod  15 ; /* nomes das rotinas chamadas     */

end Msb  NUM_ROT ;


struct beg              /* Estrutura para Fan-in          */

   char spnome  TS ;  /* nome da rotina                 */

   int  spn;          /* numero de rotinas que a chamam */

   char * spmod  15 ; /* nomes das rotinas              */

end Msp  NUM_ROT ;
```

# CAPÍTULO V - CONCLUSÕES

A necessidade da criação de ambientes automatizados de desenvolvimento de software tornou-se uma realidade, especialmente em ambientes de desenvolvimento de software básico de grande escala. Estes ambientes automatizados se caracterizam pelo agrupamento de várias ferramentas de apoio aos desenvolvedores de software durante o processo de desenvolvimento. Entre elas encontram-se as ferramentas de apoio à fase de programação, objetivando a garantia da qualidade de programas. AEP tem esta finalidade.

O AEP é uma proposta de analisador estático de programas escritos na linguagem C. A escolha desta linguagem se deveu ao tipo de ambiente a que a ferramenta se destina. Os programas desenvolvidos para este ambiente estão escritos em C.

No âmbito dos analisadores estáticos de programas, poucas linguagens de programação receberam tanta atenção quanto a linguagem FORTRAN. A razão disto ainda não se sabe. O fato é que, no caso particular da linguagem C, as raras ferramentas disponíveis de auxílio ao programador não são suficientes, considerando a expansão do uso da linguagem em vários tipos de ambientes de processamento de dados. A construção desta ferramenta vem de encontro a esta necessidade. Além disso, AEP foi originalmente concebido para que, em conjunto com outras ferramentas automatizadas, compusesse um ambiente de apoio automatizado à construção de softwares

com características bastante específicas. Este ambiente foi descrito em SANTOS (9).

Em última análise, o que se pretendeu com a construção desta ferramenta foi gerar um analisador estático não limitado à detecção de anomalias de código, característica básica dos analisadores estáticos existentes, mas que também fosse capaz de fornecer outros tipos de resultados que auxiliassem o programador na avaliação da qualidade de seus programas.

No estágio atual, o AEP procura avaliar programas considerando o fator legibilidade de código. Entretanto, muitos dos atributos relacionados ainda não podem ser verificados por ele. Futuramente, a ferramenta deverá ser expandida para que tais verificações sejam possíveis. Neste processo de expansão, o AEP também deverá ser capacitado a avaliar outros fatores que influenciam a qualidade dos programas.

O AEP também foi totalmente programado em C. Atualmente, a ferramenta está funcionando de forma experimental. Inicialmente, ela pode ser executada em equipamentos COBRA-500 sob o sistema operacional SOD.

# REFERÊNCIAS BIBLIOGRÁFICAS


(01) RAMAMOORTHY, C.V. and HO, S.F., "Testing Large Software with Automated Software Evaluation System", IEEE Transactions on Software Engineering, Vol. SE-1 (1): 20-27, Mar. 1975.

(02) BOEHM, B.W., BROWN, J.R., KASPAR, H., LIPOW, M., MACLEOED, G.J. and MERRIT, M.J. Characteristics of Software Quality, North Holland, Amsterdan, 1978.

(03) DUNN, R. and ULMAN, R., Quality Assurance for Computer Software, McGraw Hill.

(04) CHOW, T.S., "Software Tools", Tutorial Software Quality Assurance: A practical approach, IEEE, Catalog Number EH0223-8, p.369-373, 1985.

(05) CONTE, S.D., DUNSMORE, H.E. and SHEM, V.Y., Software Engineering Metrics Models, The Benjamin/Cummings Publishing Company, Inc., 1986.

(06) HANS_LUDWIG HAUSEN, SOFTWARE VALIDATION, North-Holland, 1984.

(07) McCALL, J.A., RICHARDS, P.K. and WALTERS, G.F., Factors in Software Quality, General Electric, Command and Information Systems, Technical Report 77CIS02, Sunnyvale, California, 1977.

(08) PERLIS, A., SAYWARD, F. and SHAW, M. , Software Metrics: An Analysis and Evaluation, The MIT Press Series in Computer Science, 1981.

(09) SANTOS, C.J. Um Ambiente de Apoio Automatizado para o Desenvolvimento de Software Básico , Tese de Mestrado, COPPE - URFJ, março de 1987.

(10) ANSI/IEEE std 829-1983, IEEE Standard for Software Test Documentation, Instituite of Electrical and Eletronics Engineers, 1983.

(11) ROCHA, A.R.C. da, Análise e Projeto Estrutado de Sistemas, Ed. Campos, 1987.

(12) FREITAS, A.C.C.T., BARGUT, M.F. e ROCHA, A.R.C. "Características de Qualidade de Programas", ES-83/85 COPPE - UFRJ.

(13) FERREIRA, A.B.H., ANJOS, M., FERREIRA, E.T., MARQUES, J.C. e MOUTINHO, S.R.O., Novo Dicionário da Língua Portuguesa, Editora Fronteira S.A.

(14) LOWELL, JAY ARTHUR, Measuring Programmer Productivity and Software Quality, John Wiley & Sons, Inc., 1985.

(15) McCABE, T.J., "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, 4: 308-320,

December 1976.

(16) ELSHOFF, J., "An Analysis of Some Commercial PL/I Programs", IEEE Transactions on Software Engineering, Vol. SE-2: 113-120, June 1976.

(17) HALSTEAD, M.H., Elements of Software Science, New York: Elsevier North Holland, 1977.

(18) YOURDON, E. and CONSTANTINE, L. L., Strutured Design, prentice Hall, 1979.

(19) NASCIMENTO, E. M e ROCHA, A. R. C., "Manual para Avaliação da Qualidade de Programas: Sub-fator Moduralidade", ES-112/86 COPPE/RJ.

(20) FAIRLEY, E.R., Software Engineering Concepts, Mc Graw-Hill Book Company,1985.

(21) DUNN, ROBERT H., Software Defect Removal, Mc Graw-Hill, 1984.

(22) CARRÉ, B.A., "Software Validation", Microprocessors and Microsystems, Vol. 4: 395-406.

(23) HUANG, J.C.,"Program Instrumentation and Software Testing", IEEE Computer, Vol. 11 (4): 25-32, Apr. 1978.

(24) FAIRLEY, R.E., Tutorial: Static Analysis and Dynamic

Testing of Computer Software, IEEE Computer Magazine, 11 (4): 14-23, April 1978.

(25) FOSDICK, L.D. & OSTERWEIL, L.J., "Data Flow Analysis in Software Reliability", ACM Computing Surveys, Vol. 8 (3): 305-330, Sept. 1976.

(26) BOYER, R. et al., "SELECT - A Formal System for Testing and Debugging Programs by Simbolic Execution", Proc.Internatl Conf. Reliable Software, IEEE, No.75, 234 - 245, 1975.

(27) KERNIGHAN, B.W. and RITCHIE, D.M, The C Programming Language, The Prentice-Hall, Inc., Englewood Cliffs, New Jessey.

(28) ANSI/IEEE std 830-1984, Software Requirements Specifications Standard, 1984.

(29) ANSI/IEEE std 1016/D6-1986, Recommended Practice Software Design Descriptions, 1986.

(30) JUGNET, P., Introdução ã Metodologia Orientada a Objetos, COBRA/DSO/SEFER, publicação interna, 1986.

(31) JOHNSON, S. C., Lint, a C program checker, Bell Lab., Murray Hill, NJ, Computer Science Tecn. Rep., July 1978.

## ANEXO 1: Manual do Usuário

### I. Introdução

O objetivo deste manual é colocar você, usuário, em contato com o AEP - O Analisador Estático de Programas escritos na linguagem C. Com este manual, você estará capacitado a executar a ferramenta corretamente e sem dificuldades.

Para sua maior compreensão, este manual foi organizado da seguinte forma:

- **O que é o AEP**
  Explica o que significa a ferramenta AEP;
- **Ativação do AEP**
  Contém como ativar a execução do AEP;
- **Saídas geradas pelo AEP**
  Contém os tipos dos resultados obtidos pela sua execução e a interpretação de cada um deles;
- **Formato do Relatório de Saída**
  Contém o formato do relatório de saída emitido pelo AEP;
- **Mensagens**
  Contém as mensagens geradas pelo AEP.

**Notação :**
Os colchetes que aparecem no comando de ativação significam que o item contido neles é opcional.

### II. O que é o AEP

O AEP é um protótipo de Analisador Estático destinado a programas escritos na linguagem C. A finalidade deste analisador é auxiliar o programador na difícil tarefa de construir programas de qualidade. Entretanto, nem todos os fatores que assegurem a qualidade de programas são verificados por esta ferramenta. O AEP procura medir o quanto um programa é ou não legível. Baseado neste princípo, o AEP percorre o código do programa fonte sendo analisado, detectando alguns tipos de anomalias, tais como:

- Variáveis não definidas;
- Variáveis não referenciadas;
- Retornos de rotinas;
- Batimento de argumentos contra parâmetros, quanto ao número e ao tipo;
- Atribuições envolvendo tipos diferentes;

e fornecendo medidas que permitam a avaliação de cada módulo do programa, tais como:

- Tamanho do módulo;
- Complexidade do módulo;
- Número e identificação dos módulos que o chamam;
- Número e identificação dos módulos chamados por ele;

## III. Ativação do AEP

Para ativar a execução do AEP, deve ser fornecido o seguinte comando:

```
:EX AEP PA
 NO=Arquivo_entrada [Parametros_de_ativacao]
```

Onde:

. Arquivo_Entrada

É o Nome do arquivo de entrada que contém o programa a ser analisado;

. Parâmetro_de_Ativação

Lista de parâmetros que especificam as funções a serem executadas pelo AEP. Abaixo estão relacionados os parâmetros válidos:

-A

Pesquisar anomalias de código;

-C

Calcular complexidade de módulo;

-T

Calcular tamanho de módulo;

-f

Calcular o número de módulos superiores a cada módulo do programa;

-F

Calcular o número de módulos subordinados a cada módulo do programa.

Além dos parâmetros acima, você pode selecionar as funções que o AEP deve executar. Para tanto, os parâmetros correspondentes às funções não desejadas devem ser prefixados pelo caractere alfabético N.

Você também pode selecionar os módulos do programa que deseja analisar. Isto é possível através da especificação do parâmetro de ativação -R, conforme formato abaixo:

-R modulos

Onde **modulos** especifica o conjunto de módulos selecionados para a análise. Os nomes dos módulos devem corresponder a nomes de rotinas definidas no programa fonte. Se houver mais de uma rotina selecionada, elas devem estar separadas entre si por barra (/).

Obs.:

(1) Todos os parâmetros devem estar obrigatoriamente prefixados com hífen (-) sem espaço separador;

(2) A ausência de qualquer parâmetro de ativação indica que a função correspondente deve ser executada;

(3) Qualquer erro a nível dos parâmetros causa a desativação do AEP.

## IV. Saídas geradas pelo AEP

Ao terminar a análise de um programa fonte, o AEP gera um relatório de saída, fornecendo os seguintes tipos de resultados:

- **Variável não definida**

  Acontece quando a uma variável não é feita nenhuma atribuição de valor. Isto pode gerar um resultado imprevisível. A definição de uma variável é importante para evitar o uso de valo-

res indesejados durante a execução de programa.

- **Variável não referenciada**

Uma variável é não referenciada quando ela está declarada no corpo do programa mas não é utilizada em nenhum outro lugar. Neste caso, você deve se certificar da sua necessidade. Caso não tenha utilização, esta variável deve ser eliminada do código.

- **Anomalias em uso de rotinas**

São dois os tipos de anomalias detectadas:

(1) Sempre que uma rotina for chamada, são verificados o número de argumentos passados e o tipo de cada argumento. Estes dados são comparados com a definição da rotina e caso não sejam compatíveis, são geradas informações no relatório de saída, explicitando cada tipo de ocorrência;

(2) Sempre que houver um comando de retorno explícito de rotina, o tipo do resultado retornado é comparado com o tipo que a rotina deve retornar de fato. Caso os tipos sejam incompatíveis, é gerada uma mensagem, destacando a ocorrência.

- **Atribuição**

Sempre que houver um comando de atribuição, o tipo do lado esquerdo do sinal de atribuição é comparado com o tipo do lado direito. Caso sejam incompatíveis, a ocorrência é informada no relatório de saída.

- **Tamanho de módulo**

  O tamanho do módulo (rotina) representa o número de linhas que ele contém. Para maior clareza do módulo, este tamanho não deve ultrapassar a duas páginas de listagem ou aproximadamente 50 linhas.

- **Complexidade de módulo**

  A complexidade de um módulo (rotina) representa o grau de dificuldade do seu código. No relatório de saída é fornecido um valor que corresponde à esta complexidade. Este valor quando maior do que 10 indica que o módulo deve ter sua lógica refeita.

- **Módulos Subordinados ("Fan_out")**

  Um módulo do programa fonte pode chamar vários outros módulos. Para cada módulo, AEP calcula o número de módulos chamados por ele, identificando cada um. Este número não deve ultrapassar a nove.

- **Módulos Superiores ("Fan_in")**

  Um módulo do programa pode ser chamado por vários outros módulos do programa. Para cada módulo chamado, AEP calcula o número de módulos que o chamou, identicando-os. Este número não deve ultrapassar a três.

## V. Formato do Relatório de Saída

A cada execução do AEP, é gerado um relatório de saída que pos-

sui o seguinte formato:

(1) Página inicial do relatório:

Na primeira página do relatório de saída gerado pelo AEP são impressos os parâmetros de ativação fornecidos no momento da ativação da execução da ferramenta. Esta página tem seguinte formato:

```
PARAMETROS DE ATIVACAO:
      ARQUIVO DE ENTRADA: xxxxxxxx
      OPCOES: xxx xxx xxx
```

(2) As demais páginas do relatório contêm o resultado da análise efetuada sobre o programa fonte. As informações contidas nestas páginas vão depender das funções que você pediu que fossem executadas. Estes resultados estão dispostos na seguinte ordem:

```
*** ANOMALIAS DO MODULO: xxxxxxxx

* VARIAVEIS NAO DEFINIDAS

       NOME                 LINHA DE DECLARACAO
       xxxxxxxx             xxx
       xxxxxxxx             xxx

* VARIAVEIS NAO REFERENCIADAS

       NOME                 LINHA DE DECLARACAO
       xxxxxxxx             xxx
```

```
     xxxxxxxx          xxx

* PARAMETROS - ARGUMENTOS:
  ROTINA        NUM PAR       NUM ARG     LIN DECL   LIN OCOR
  xxxxxxxx      xx            xx          xxx        xxx
  xxxxxxxx      xx            xx          xxx        xxx

  POS PAR/ARQ   TIPO PAR     TIPO ARG     LIN DECL    LIN OCOR
  xx            xxxxxxxx     xxxxxxxx     xxx         xxx
  xx            xxxxxxxx     xxxxxxxx     xxx         xxx

* ATRIBUICOES INCOMPATIVEIS:
     LINHA          TIPO A ESQUERDA        TIPO A DIREITA
     xxx            xxxxxxxx               xxxxxxxx
     xxx            xxxxxxxx               xxxxxxxx

* OUTRAS ANOMALIAS:
    - lin xxx      xxx... mensagem ...xxx
    - lin xxx      xxx... mensagem ...xxx


*** TAMANHOS DOS MODULOS:
* MODULO: xxxxxxxx         xxx linhas
* MODULO: xxxxxxxx         xxx linhas


*** COMPLEXIDADES DOS MODULOS:
* MODULO: xxxxxxxx         xxx
* MODULO: xxxxxxxx         xxx
```

```
*** MODULOS SUBORDINADOS A:
* MODULO: xxxxxxxx    TOTAL: xxx     MODULOS SUBORDINADOS:
                                     - XXXXXXXX
                                     - XXXXXXXX
                                     - XXXXXXXX

*** MODULOS SUPERIORES A:
* MODULO: xxxxxxxx    TOTAL: xxx     MODULOS SUPERIORES:
                                     - XXXXXXXX
                                     - XXXXXXXX
                                     - XXXXXXXX

*** ANOMALIAS DE VARIAVEIS GLOBAIS:

* VARIAVEIS NAO DEFINIDAS:
       NOME                LINHA DE DECLARACAO
       XXXXXXXX            XXX
       XXXXXXXX            XXX

* VARIAVEIS NAO REFERENCIADAS:
       NOME                LINHA DE DECLARACAO
       XXXXXXXX            XXX
       XXXXXXXX            XXX
```

# VI. Mensagens

As mensagens do AEP estão classificadas em dois tipos: Mensagens de Anomalias de código e Mensagens de Erros Fatais.

## VI.1. Mensagens de Anomalias de Código

São as mensagens impressas no relatório de saída devido à ocorrência de anomalias de código. Estas mensagens estão relacionadas abaixo:

. **Tipo retornado diferente do tipo da rotina**

O Tipo do resultado retornado pela rotina não é compatível com o tipo que ela deve retornar, de acordo com a sua definição;

. **Rotina do tipo VOID**

A rotina está retornando valor quando não deveria. Rotinas definidas como VOID não retornam valores.

## VI.2. Mensagens de Erros Fatais

São as mensagens correspondentes aos erros que causam a desativação do AEP. Estes erros ocorrem por problemas nos parâmetros de ativação, na especificação do arquivo de entrada ou por erros sintáticos ou léxicos. As mensagens são as seguintes:

. **Simbolo nao declarado**

O símbolo não está declarado no programa fonte.

. **Simbolo ja existe**

Tentativa de declarar o mesmo símbolo mais de uma vez.

. **Operacao invalida**

Tentativa de fazer operação em expressão não permitida pela linguagem C.

. **Declaracao de parametro inexistente**

O parâmetro fornecido a nível de parâmetros de ativação não existe.

. **Simbolo lexicamente incorreto**

Elemento não reconhecido pelo AEP.

. **Erro na abertura de Arquivo**

O arquivo de entrada fornecido como parâmetro não existe.

. **Arquivo de entrada duplicado**

Mais de um arquivo de entrada especificado.

. **Arquivo de entrada mal especificado**

A especificação do arquivo de entrada não está de acordo com o formato reconhecido.

# ANEXO 2: Exemplo de execução do AEP

## I. Introdução

Neste anexo apresentamos um exemplo de execução da ferramenta AEP aplicado a um determinado programa.

O programa exemplo se destina a calcular a média obtida por cada aluno de uma turma hipotética, onde as médias resultantes abaixo de cinco, inclusive, indicam os alunos em prova final e as médias acima de cinco indicam os alunos aprovados sem prova final. Como resultado final, o programa deve fornecer as taxas de aprovação e de alunos em prova final, respectivamente.

O programa contém anomalias de código. A análise do fonte indicará tais anomalias, além de fornecer os tamanhos de cada módulo que o compõe, as medidas de complexidade, o "fan-out" e o "fan-in".

```
 1
 2   /*
 3    * Este programa calcula a media obtida por cada aluno
 4    * de uma determinada turma.  Ao final, deve ser for -
 5    * necido o rendimento da turma, considerando-se a ta-
 6    * xa de aprovacao sem a necessidade de prova final  e
 7    * a taxa de alunos em prova final.
 8    *
 9    */
10
11   int media (n_tst)
12   int n_tst;            /* numero de testes realizados */
13
14   {
15
16     extern int getc ();
17
18     char contador ;
19
20     int
21          acum = 0
22        , med
23        ;
24
25     while (contador < n_tst)
26
27     {
28
29       acum = acum + getc ();
30
31       ++ contador;
32
33     }
34
35     med = acum / n_tst;         /* calculo da media    */
36
37     return (med);
38
39   }
40
```

```
-----  NO=CMEDIA   [DP05]  CO=VALPRO  -----  12/05/88  14:12:37  ----  PAGIN

 42   main ()
 43
 44   {
 45
 46     extern int  getc   ();
 47     extern void printf ();
 48
 49     char   med_al, n_testes, n_alunos, ctr_tot;
 50
 51     int    al_apr, al_pf, porc_apr, porc_pf;
 52
 53     while (ctr_tot < n_alunos)
 54     {
 55
 56       med_al = media (n_testes);
 57
 58       if (med_al <= 5)
 59       {
 60
 61         ++ al_pf;
 62
 63         ++ ctr_tot;
 64
 65         printf ("\n Aluno %02d => Media: %d - FINAL",
 66                     ctr_tot , med_al);
 67
 68       }
 69
 70       else {
 71
 72         ++ al_apr;
 73
 74         ++ ctr_tot;
 75
 76         printf ("\n Aluno %02d => Media: %d - APROVADO",
 77                     ctr_tot , med_al);
 78
 79       }
 80
 81     }
 82
 83     porc_apr = (al_apr * 100) / n_alunos;
 84
 85     printf ("\n *** Taxa de alunos aprovados: %d",
 86               porc_apr);
 87   }
```

```
PARAMETROS DE ATIVACAO:

        ARQUIVO DE ENTRADA: NO=CMEDIA

        OPCOES:  -A  -T  -C  -F  -f
```

*** ANOMALIAS DETECTADAS: media

* VARIAVEIS NAO DEFINIDAS:

| NOME | LINHA DE DECLARACAO |
| --- | --- |
| contador | Ø18 |

*** ANOMALIAS DETECTADAS: main

* VARIAVEIS NAO DEFINIDAS:

| NOME | LINHA DE DECLARACAO |
|---|---|
| n_testes | 049 |
| n_alunos | 049 |
| ctr_tot | 049 |
| al_apr | 051 |
| al_pf | 051 |
| porc_pf | 051 |

* VARIAVEIS NAO REFERENCIADAS:

| NOME | LINHA DE DECLARACAO |
|---|---|
| getc | 046 |
| porc_pf | 051 |

* PARAMETROS - ARGUMENTOS:

| POS PAR/ARG | TIPO PAR | TIPO ARG | LIN DECL | LIN_OCOR |
|---|---|---|---|---|
| 01 | INT | CHAR | 011 | 056 |

* ATRIBUICOES INCOMPATIVEIS:

| LINHA | TIPO A ESQUERDA | TIPO A DIREITA |
|---|---|---|
| 056 | CHAR | INT |

*** TAMANHOS DOS MODULOS:

| | |
|---|---|
| * MODULO: media | 025 linhas |
| * MODULO: main | 043 linhas |

*** COMPLEXIDADES DOS MODULOS:

* MODULO: media 002

* MODULO: main 003

*** MODULOS SUBORDINADOS A:

| | | |
|---|---|---|
| * MODULO: media | TOTAL: 001 | MODULOS SUBORDINADOS:<br><br>- getc |
| * MODULO: main | TOTAL: 002 | MODULOS SUBORDINADOS:<br><br>- media<br>- printf |

*** MODULOS SUPERIORES A:

* MODULO: media TOTAL: 001 MODULOS SUPERIORES:

- main

*** ANOMALIAS DE VARIAVEIS GLOBAIS:

* VARIAVEIS NAO REFERENCIADAS:

| NOME | LINHA DE DECLARACAO |
|---|---|
| main | 042 |