ESPECIFICAÇÕES FORMAIS EM OBJ

por

Nina Edelweiss
Adagenor Lobato Ribeiro

RP nº 113 Julho 1989

Trabalho desenvolvido na disciplina
COMP06-Especificações Formais

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Av. Osvaldo Aranha, 99
90.210 - Porto Alegre - RS - BRASIL
Telefone: (0512) 21-84-99
Telex: (051) 2680 - CCUF BR

Correspondência: UFRGS-CPGCC
Caixa Postal 1501
90001 - Porto Alegre - RS - BRASIL

Engenharia de Software - SBU
Engenharia: Software
Especificação formal
OBJ

ÍNDICE

RESUMO

Este trabalho apresenta a linguagem de especificação formal OBJ. Trata-se de uma linguagem executável, que serve para implementar e testar especificações algébricas. São apresentados a sintaxe e a semântica do OBJ e um pequeno exemplo de sua utilização.

ABSTRACT

This work presents OBJ, a formal specification language. OBJ is an executable language, used to implement and test algebraic specifications. There are presented OBJ's syntax and semantics, and a short example of its use.

1. INTRODUÇÃO

A complexidade crescente dos programas de aplicação tem tornado os custos do software cada vez mais elevados. Na procura de meios para diminuir estes custos, surgiram dois enfoques: a utilização de especificações formais e a reutilização de software.

No ciclo de vida de um software, a primeira operação a ser realizada é a especificação informal do sistema pelo usuário, através de linguagem natural. Da análise dos requisitos por ele colocados deve resultar um modelo do sistema, expresso em algum formalismo. É, portanto, realizado um processo de abstração, que, por transformações sucessivas, leva à especificação formal do sistema.

As especificações formais tem o objetivo básico de definir o que um sistema deve fazer, sem descrever como isto deve ser feito. Uma especificação formal define um sistema de uma maneira independente da implementação. Isto pode ser feito, por exemplo, descrevendo seus estados internos em termos de **tipos abstratos de dados**, caracterizados somente pelas operações permitidas sobre eles.

Segundo [CAS86], uma especificação formal é a descrição de um pedaço da realidade com o uso de um formalismo. Por formalismo se entende o sistema formado por uma linguagem simbólica e por um conjunto de axiomas definidos sobre esta linguagem (sintaxe e semântica da linguagem). Segundo [MEN88], as especificações se classificam em *formais* e *semi-formais*. As *semi-formais* se caracterizam pela utilização de linguagens que ppdem ter uma sintaxe bem definida, mas não tem a semântica precisa, permitindo que uma série de erros não sejam detectados. Uma especificação é *formal* quando utiliza uma linguagem com sintaxe e semântica rigorosamente definidas.

Quanto maior o grau de formalismo, menor se torna a legibilidade da especificação. Entretanto, segundo [BOE80], quanto mais estáveis os requisitos do problema e quanto maior a necessidade de inexistência de erros, mais formal deve ser a especificação.

Dentre as linguagens de especificação formal, grande ênfase tem sido dada àquelas que se baseiam em propriedades matemáticas. Entre estas se situam as **especificações algébricas**, que especificam as operações de tipos abstratos de dados implicitamente, relacionando-os entre si através de equações algébricas.

Atualmente já se dispõe de um bom número de técnicas diferentes para especificação formal. Algumas se preocupam com a melhoria da comunicação entre o projetista e o usuário; outras, em fornecer ferramentas para provar alguma característica específica do programa; outras ainda, dão prioridade à facilidade de implementação da especificação. Dentro deste último grupo, uma especificação executável representa o ideal.

A outra característica citada inicialmente para a diminuição dos custos de desenvolvimento, é a reutilização de software. Uma das maneiras de se alcançar a sua maximização é através da utilização de

programação parametrizada.

Neste trabalho será apresentado o método de especificação formal **OBJ**, projetado por Joseph Goghen em 1976, cujo objetivo é escrever e testar especificações algébricas. O **OBJ**, além das vantagens intrínsecas dos métodos de especificação formal, apresenta dois importantes aspectos do que foi antes apresentado: é parametrizado, permitindo portanto a reutilização de software e é executável, o que proporciona a implementação direta da especificação.

Apresentamos inicialmente, na seção 2, alguns aspectos gerais de especificações algébricas. As características principais da linguagem **OBJ** são vistas na seção 3. Na seção 4 é detalhada a sintaxe da linguagem **OBJ**, ficando a semântica para a seção 5. A seção 6 traz um exemplo simples, inicialmente especificado através de uma especificação algébrica e depois na linguagem **OBJ**, com o objetivo de mostrar a facilidade de utilização da linguagem. A seção 7 apresenta conclusões.

No Anexo é apresentado um exemplo encontrado em [GOG82], que mostra a saída da execução da especificação em **OBJ** de uma linguagem de programação simples, chamada **FUN**. Este exemplo utiliza a versão de **OBJ** T, que não é a mesma apresentada neste trabalho. Foi acrescentado com o objetivo de mostrar como se apresenta uma especificação em execução, além de permitir a comparação da sintaxe da última versão da linguagem, **OBJ 3**, com uma versão anterior.

2. ESPECIFICAÇÕES ALGÉBRICAS

2.1 ABSTRAÇÃO E PROGRAMAÇÃO

Uma tendência muito forte, hoje, na ciência da programação é o desenvolvimento e a utilização de métodos formais, utilizando-se extensivamente dos conceitos de abstração.

Um problema que se apresenta central no desenvolvimentto de software é a redução de sua complexidade. Isto pode ser feito tratando o problema aos poucos, inicialmente de uma maneira global, incluindo depois os detalhes em algum momento da concepção do produto. A abordagem mais comum visando tratar desse aspecto é a decomposição de uma dada tarefa em subtarefas (algumas vezes uma subtarefa é tão complexa que deve ser novamente subdividida) o que leva ao desenvolvimento por refinamentos sucessivos, em vários níveis de abstração.

Dijktra já afirmou que o maior recurso de que dispomos para tratar a complexidade é a abstração. A abstração separa os atributos que são relevantes num dado contexto, daqueles que não o são. Reduzindo o total de detalhes, o problema a resolver torna-se muito mais simples. Ignorar os detalhes que são irrelevantes para uma determinada fase de desenvolvimento de um software, permite que nos concentremos no que é altamente relevante para aquela fase. Segundo essa forma de visão e em concordância com [GUT78a], um sistema de software deve conter uma variedade de tipos (listas, pilhas, árvores, matrizes, etc.) e uma variedade de operações. Um usual procedimento de projeto é tratar essas operações que atuam primariamente sobre um simples tipo de dado, como formando uma unidade (esse é o princípio que rege o conceito de classe da linguagem SIMULA 67) para considerar a semântica dessas operações como a definição do tipo. Na contextualização de programação, as linguagens de programação trouxeram uma substancial ajuda, representada pela introdução do subprograma. O subprograma executa função ou procedimento abstrato específico, através de um algoritmo nele embutido. Quando chamamos um subprograma, separamos claramente "o que" ele faz do "como" ele faz. Esse recurso é deveras poderoso, considerando que via subprogramação podemos desenvolver uma hierarquia de abstrações, ou refinamentos, gerando uma árvore que reflete todos os passos da solução do problema.

A subprogramação, através do conceito de procedimento, foi uma forma primitiva de abstração. A abstração procedural é, hoje, a base fundamental da qualidade de software, tornando este modular e reutilizável.

Abstração de dados é uma extensão deste conceito, tratando o objeto de dado como uma função, onde o mais importante está representado em sua sintaxe e semântica. O comportamento esperado do objeto nesse contexto, como detalharemos *a posteriori* numa particular implementação, passa a ser somente uma instanciação do objeto considerado.

2.2 TIPOS ABSTRATOS DE DADOS

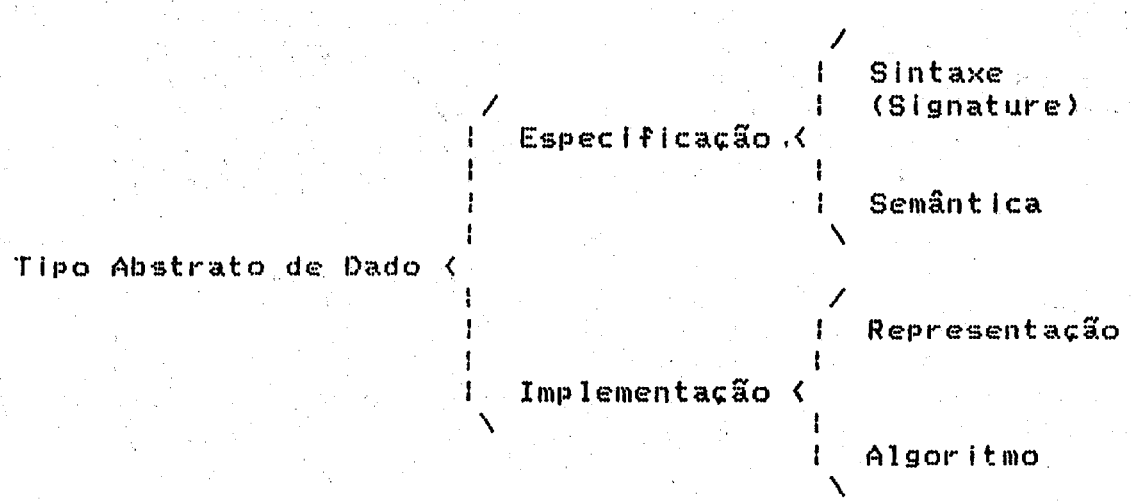
Um dos mais importantes conceitos de porgramação introduzidos nos últimos anos foi o de programação com tipos abstratos de dados. Os princípios de abstração no desenvolvimento de *software* promovem um caminhamento natural para a programação com tipos abstratos de dados. O conceito de "Tipo Abstrato de Dado" é utilizado para se referir uma **classe de objetos** pertencentes a um determinado domínio, definidos por uma especificação independente de sua representação. Abstração de dados é uma técnica de projeto de *software* que promove a modularidade e o desenvolvimento independente da implementação de abstração de dados e de programas de aplicação.

Existem muitas razões para a inclusão de tipos abstratos de dados em programação, principalmente o fato de que o texto do programa resultante reflete com precisão e naturalidade todo o processo de pensamento que comandou sua construção, o que resulta em um programa com alto grau de legibilidade e manutenível.

A filosofia de tipos abstratos de dados, como já foi visto, provê uma nova forma de organizar e projetar programas que são mais confiáveis, autodocumentados e em sua concepção global, um produto lógico e elegante.

A idéia fundamental acerca de tipos abstratos de dados é precisamente a separação da utilização do tipo de dado de sua eventual implementação.

A idéia de separar esses estágios não é recente, porém sua aplicação em tipos abstratos de dados, formalização e inclusão em linguagens de programação é nova, tendo aparecido inicialmente nas linguagens de programação ALGOL, ADA, ALPHARD, SIMULA, CLU e muitas outras. Um tipo abstrato de dado, de acordo com [CLE86], pode ser visualizado como dividido em quatro grandes partes, como mostrado abaixo:



As duas primeiras partes, sintaxe e semântica, definem como o programa de aplicação usa o tipo abstrato de dado. As últimas duas partes, representação e algorítmo, definem a possível implementação de um tipo abstrato de dados.

O que é fundamental, que voltamos a frisar neste ponto, é o fato de que para se utilizar um tipo abstrato de dados, não é necessário saber como ele foi implementado. É muito importante nesse paradigma de programação, que o conhecimento sobre a implementação do tipo não seja utilizado, sendo este o ponto central da abstração de dados. Segundo [GUT78b], no projeto de tipos de dados, na criação de um sistema de *software*, nosso maior propósito é explorar um meio de especificar um tipo de dado que seja independente de sua eventual implementação. Assim a abstração de tipos de dados permite ao usuário a definição de novos tipos e o seu consistente uso como parte do repertório do elenco de tipos em seu particular programa ou sistema de *software*.

A **sintaxe** de um tipo abstrato de dados especifica todos os nomes de função, o número e tipo de todos os operandos e o tipo de valor retornado. A sintaxe não especifica que valor deve ser retornado para uma dada entrada. A sintaxe é dada numa declaração de procedimento. A **semântica** é representada por um conjunto de axiomas que definem o significado das operações, através do estabelecimento dos relacionamentos funcionais de uma operação para outra.

A **implementação** de um tipo abstrato de dado consiste de representação das estruturas de dados e algoritmos de manipulação. A separação entre o uso de um tipo de dado (especificado pela sintaxe e pela semântica) é importante por várias razões, uma das quais diz respeito a aspectos de correção. Algumas vezes isto é criticamente importante para implementar determinada parte de um programa corretamente, outras vezes o aspecto de correção concerne a implementação e manipulação de algum dado ou tipo de dado. Tipos abstratos de dados simplificam estes aspectos de correção, colocando o código que fisicamente toca o dado em um lugar e tornando-o inacessível para o resto do programa de aplicação. Ao restante do programa de aplicação é permitido o uso do dado, utilizando para tal somente um conjunto de operações bem definidas sobre o tipo de dado. Esta restrição é algumas vezes chamada na literatura corrente de **proteção** ou **encapsulamento**. O dado que é oculto por um abstração recebe também as denominações de **dado privado** ou **dado protegido**. Esses aspectos constituem os maiores esforços presentes em muitas linguagens de programação que trabalham com tipos abstratos de dados.

Em síntese, a programação com tipos abstratos de dados, para ser utilizada na resolução de um problema, preocupa-se em primeiro lugar em determinar que conjunto de operações devem ser feitas sobre o tipo abstrato de dado, ou melhor, como o dado será usado. Na abordagem tradicional determina-se primeiro a sua representação. Programas com tipos abstratos de dados são naturalmente bem estruturados e mais modulares. A interface bem definida, especificada pelos operadores que agem sobre o tipo, comanda o desenvolvimento independente das duas partes do programa.

2.3 ESPECIFICAÇÃO DE TIPOS ABSTRATOS DE DADOS

Uma linguagem para especificação de tipos abstratos de dados deve ser formal. Uma especificação é dita formal se for escrita em uma linguagem com sintaxe e semântica explícitas e precisamente definidas. Uma linguagem informal, como a natural, é para esse propósito muito ineficiente, por causa de sua inerente ambiguidade, o que acarreta imprecisão e, concequentemente, uma especificação incompleta. É importante que se frise bem esse aspecto neste momento, pois a especificação constitui-se em um documento para o desenvolvimento de todo o sistema de software.

O uso de uma linguagem formal não é necessariamente uma garantia de que a especificação está completa e consistente. Esse é um aspecto que diz muito mais respeito ao especificador, sua habilidade, capacidade e inteligência, análise e conhecimento acerca do problema. Nenhuma linguagem formal faz a mágica de produzir uma especificação correta, ela é apenas uma ferramenta que deve ser utilizada com habilidade, capacidade e inteligência pelo especificador do software.

Uma boa técnica de especificação formal facilita a produção de boas especificações formais. Uma boa especificação deve ser bastante restritiva para assegurar que alguma coisa que não seja acessível ao especificador possa reunir requerimentos impostos pela especificação.

Uma boa especificação deve ser adaptada à aplicação intencionada. Deve-se procurar especificar formalmente um software para um determinado problema com uma linguagem formal que reuna propriedades e características esperadas para o comportamento deste problema. Por exemplo, nos problemas que apresentam concorrência deverá ser utilizada uma linguagem que possa tratar desta característica, tal como *Rede de Petri* ou *CCS*, o que não é o caso de *OBJ*, *VDM* e outras.

Uma linguagem de especificação é também um veículo de comunicação entre o usuário e o projetista do software. Mesmo que para alguns usuários seja difícil entender um formalismo, isso não elimina o papel da especificação como documento de uma comunicação precisa. A lingugem natural é fácil de ser lida mas não é precisa, enquanto que as linguagens formais são precisas mas não tão fáceis de serem compreendidas. Esses dois objetivos conflitantes não são únicos da especificação de software com tipos abstratos de dados. Eles apresentam um grande desfio para a comunidade científica da engenharia de software, principalmente ante a necessidade crescente de programação em larga escala, gerando programas corretos, confiáveis, de boa qualidade e econômicos.

2.4 ESPECIFICAÇÃO ALGÉBRICA

O conceito de tipo abstrato de dado é de extrema importância na especificação algébrica, a qual trabalha exaustivamente com a manipulação de objetos. Acerca de objetos em programação, ainda não existe um consenso preciso na comunidade de software. Para nossos propósitos vamos considerar uma definição de [GUT78a] a qual diz que um dado objeto, abstraído em um tipo abstrato de dado T, é uma classe de valores e uma coleção de operações sobre esses valores.

Em [CAS86] encontramos que, através de um formalismo algébrico, é possível especificar que um objeto abstrato corresponda à parte relevante da realidade que desejamos formalizar. Os objetos abstratos assim formalizados constituem Álgebras. O desenvolvimento da especificação algébrica está ligado aos trabalhos de Guttag, Goghen e outros que pesquisam particularmente em especificações algébricas de tipos abstratos de dados.

A especificação algébrica determina a sequência de operações válidas em um determinado sistema. Devemos observar que, em outras formas de especificação, normalmente cada operação é definida de forma independente. Podemos então dizer que sob certo prisma, uma especificação algébrica é muito similar a uma especificação procedural.

Mas afinal, em que consiste uma especificação algébrica? Basicamente é um conjunto de equações, ou melhor, de axiomas que deverão ser válidos para quaisquer valores de assinalamento para as variáveis. O grupo de trabalho acima referenciado afirma em um de seus artigos que a abordagem algébrica é mais poderosa que a abordagem lógica e que equações e inicializações podem fazer o que fórmulas não podem.

A especificação algébrica de um tipo abstrato de dados, de acordo com [GUT78b], consiste de tres partes:

i) **ESPECIFICAÇÃO SINTÁTICA** : é onde é dada a sintaxe do tipom, através de informações como nomes, domínios (*sortes*) e contradomínios, juntamente com as operações associadas ao tipo;

ii) **ESPECIFICAÇÃO SEMÂNTICA**: é representada por um conjunto de axiomas que definem o significado das operações, através do estabelecimento de relacionamentos de uma operação para outra;

iii) **ESPECIFICAÇÃO DE RESTRIÇÕES**: é a parte da especificação onde são feitos os tratamentos de excessões e condições de erro esperado em vista das pré-condições.

Os trabalhos de [GOG75] e [GUT78b] enfatizam fortemente a abordagem algébrica, desenvolvendo uma teoria de tipos abstratos de dados, como uma aplicação de álgebras polisortidas. Implementações são tratadas sobre esta abordagem como outras álgebras e o problema de mostrar que uma implementação está correta é tratado de forma completa mostrando-se a existência de um mapeamento homomórfico de uma álgebra para outra.

O coração de uma técnica de especificação formal repousa na linguagem utilizada. Para nossos propósitos, em concordância com [GUT78b], assumiremos uma linguagem básica com cinco primitivas:

i) composição funcional e
ii) uma relação de igualdade (=) e
iii) duas constantes distintas (true, false) e
iv) um suprimento limitado de variáveis livres.

A partir dessas primitivas podemos construir uma linguagem de especificação algébrica muito rica. Uma vez tendo definido um tipo de abstração, ele é incorporado à linguagem de especificação. Podemos, por exemplo, desejar incluir uma operação *IF-THEN-ELSE* e um tipo booleano, definida pelos axiomas a seguir:

if-then-else(true,q,r) = q
if-then-else(false,q,r) = r.

Em qualquer lugar, a partir desta axiomatização, podemos assumir que a expressão **if-then-else(bool,q,r)** é parte integrante da linguagem de especificação. Podemos também assumir a disponibilidade de operadores booleanos infixos como necessário. A axiomatização desses operadores em termos de função **if-then-else** é trivial.

Em [GUT78a] descreve-se um conjunto de caractrísticas recomendáveis para uma linguagem de especificação algébrica. Podmos observar que esse conjunto proposto, de certa forma, constitui a idéia primitiva do que posteriormente eles publicam em [GUT78b], já visto neste trabalho, no item anterior. Mas é oportuno listar aqui este conjunto, porque fica bem claro o aspecto de utilização de RECURSÃO ao se escrever uma especificação algébrica. São as seguintes as características:

i) variáveis livres;
ii) expressões if-then-else;
iii) expressões booleanas;
iv) recursividade.

Muitos aspectos normalmente supostos presentes em uma linguagem de programação convencional, como atribuição de valores a uma variável e declaração de iterações, não são permitidas neste formalismo. Qualquer iteração será feita via recursividade. Além disso, a mesma recomendação restringe o uso de procedimentos para aqueles que são simples valores e não tem efeitos colaterais.

Em [MEN88] encontramos a afirmação de que, apesar de não haver teoricamente restrições ao uso de especificação algébrica, ela tem sido mais utilizada para especificar a parte **estática** dos sistemas. Entende-se por parte estática de um sistema, seus diferentes estados, e por parte dinâmica, as diferentes transformações pelas quais ele passa.

Uma especificação algébrica pode ser construída a partir de outras especificações, assim como pequenos blocos de especificação, constituindo uma grande especificação. A resultante do uso de uma ou mais especificações deve ser rigorosamente a menor extensão consisten-

te de cada uma das especificações usadas, se existir essa extensão. A extensão deve incluir equações adicionais que devem ser fechadas quanto a um determindado **operador de clausura**, ou seja, não pode ser definida uma equação que de como resultado um tipo não definido. É importante que se diga que quando construimos especificações grandes a partir de especificações menores ou **blocos**, esse fato relaciona-se à possibilidade de **reutilizar** uma especificação já construida, o que é de altíssima importância no contexto atual da Engenharia de Software, por que representa um substrato de economia, praticabilidade, adequabilidade e de outras características úteis. Mas é importante que seja observado que, para tal propósito ser atingido, é fundamental a **compatibilidade de linguagens** e a **consistência das especificações.**

A **parametrização** de especificações algébricas é o recurso que normalmente se utiliza quando se trata de reutilizar o que já está especificado em termos algébricos. É uma poderosa técnica em projeto, manutenção e outros aspectos da produção de software. A importância de criar um novo tipo a partir de tipos existentes é a essência da parametrização. Isso significa usar o tipo definido como **parâmetro** real numa especificação definida com parâmetro formal. Devemos garantir que um parâmetro formal, quando substituido por um parâmetro real, preserve a consistência. Em geral a consistência deve ser provada caso a caso, para cada especificação parametrizada construida.

2.5 CONSIDERAÇÕES FINAIS

A especificação algébrica é um formalismo usado para especificar um objeto abstrato, que corresponde aos elementos de relevância da realidade que se deseja formalizar. No âmbito dessa forma de especificação, a formalização dos objetos abstratos constitue álgebras.

A verificação de programas que usam tipos abstratos especificados algebricamente conta com o recurso de que os tipos provêem regras de inferência que podem ser usadas para demonstrar a consistência entre um programa e a sua especificação.

A presença de definições axiomáticas de tipos abstratos de dados provê um mecanismo para provar que um programa é consistente com sua especificação, desde que a implementação da operação astrata que ele usa esteja consistente com a sua especificação. Assim, uma técnica para fatoração de provas é provida para especificações axiomáticas, e serve como a especificação do intento para um reduzido nível de abstração. Para as provas de correção de representação de tipos abstratos de dados, a especificação axiomática provê um conjunto mínimo de asserções que devem ser verificadas.

O objetivo deste capítulo foi o de prover os elementos necessários para o entendimento mais preciso da linguagem de especificação OBJ. Entendemos que, sendo OBJ uma linguagem que executa especificações algébricas, nada mais natural que, antes de utilizá-la, conhecer com mais detalhes os fundamentos algébricos que a originaram. Buscou-se mostrar os fundamentos algébricos, de uma maneira mais próxima do usuário final da linguagem. Sabemos que por trás de tudo o que foi

discutido neste capítulo existe uma sólida e consistente ferramenta matemática, que de maneira formal prova tudo o que afirmamos, mas para o contexto deste trabalho, julgamos que a forma de abordagem feita aqui é a mais conveniente.

3. A LINGUAGEM OBJ COMO UM MÉTODO DE ESPECIFICAÇÃO ALGÉBRICA

OBJ é uma linguagem formal, usada para escrever e testar especificações algébricas de programas. Além de ser uma linguagem formal, **OBJ** é também considerada como sendo uma linguagem de programação.

Através da utilização desta linguagem, o projeto e a implementação de sistemas é muito facilitado, sendo inclusive proporcionada a oportunidade da verificação automática das especificações feitas. Sendo uma linguagem razoavelmente fácil de ser utilizada, é possível sua utilização como ferramenta de prototipação rápida. Neste enfoque, permite a experimentação com variações da especificação, na procura de um melhor desempenho do sistema implementado.

Trata-se de uma linguagem de programação funcional de lógica de primeira ordem, que apresenta uma semântica algébrica formal baseada em lógica equacional, além de uma semântica operacional baseada na interpretação das equações como regras de reescrita. Isto torna possível a validação das especificações feitas através de testes e a utilização das especificações como protótipos para explorar ou confirmar a corretеza de decisões de projeto.

A linguagem **OBJ** utiliza equações algébricas para definir os tipos abstratos de dados, como foi apresentado no capítulo anterior. Estes podem ser definidos independentemente de qualquer representação e especificação. Também são encontradas no **OBJ** importantes características de modularidade e de parametrização, o que torna as especificações facilmente utilizáveis.

OBJ possibilita parametrização, modularidade e de abstração de dados. Algumas das características principais do **OBJ**, segundo [MEN88], são:

i) a possibilidade de dividir grandes espécificações em pequenas partes;
ii) testar estas pequenas partes separadamente;
iii) testar suas interconexões;
iv) usar uma notação flexível e definível pelo usuário;
v) uso sistemático de condições de erro;
vi) testes de consistência, tanto sintáticos como semânticos.

A grande utilidade do **OBJ** reside na sua ajuda para especificar estruturas de dados algebricamente de maneira correta. É bastante difícil escrever estas especificações sem cometer erros. A utilização de ferramentas automatizadas, como é o caso do OBJ, torna-se portanto fundamental para que tais tipos de especificações sejam efetivamente utilizadas.

Segundo [GOG82], um grande número de especificações algébricas publicadas contém erros, pelos mais diversos motivos, entre eles:

- erros tipográficos;
- notação incorreta ou imprecisa;
- falha em lidar com casos especiais;

- manipulação incorreta de certos casos;
- escrever especificações que não terminam, quando deveriam terminar;
- falhas em tratamento de erros, ou mesmo ausência de tal tratamento.

A utilização do OBJ tem por finalidade ajudar a evitar tais tipos de erros durante uma especificação e, quando ocorrerem, fornecer uma ferramenta que auxilie a detectá-los e corrigi-los imediatamente.

Ainda segundo [GOG82], as características que uma linguagem de programação deve apresentar para que possa ser utilizada para escrever especificações algébricas corretas são as seguintes:

- a sintaxe de cada especificação deverá ser verificada automaticamente;
- a sintaxe deve ser bastante flexível;
- o modo como as especificações se apresentam deve representar o seu significado semântico;
- deve apresentar uma *tipagem* muito forte;
- deve possibilitar a definição e a manipulação de condições de erros;
- a linguagem deve produzir mensagens para os erros mais complexos;
- cada especificação deverá ser testada através de execução mecânica de testes;
- cada especificação deverá ser dividida empartes pequenas, significativas;
- deverá permitir especificações parametrizadas;
- deverá apresentar um embasamento matemático preciso, de modo a permitir a prova matemática da correteza das especificações e de suas implementações.

O OBJ foi projetado procurando obedecer a estes requisitos. Além das características apresentadas antes, é uma linguagem com uma *tipagem* muito forte e ao mesmo tempo flexível. Isto permite a detecção de expressões incorretas antes de sua execução; a separação de conceitos lógica e intuitivamente distintos; e a documentação destas diferenças, aumentando a legibilidade da especificação.

Segundo [GOG87a], OBJ também pode ser visto como uma linguagem de programação baseada em lógica pois apresenta características de programação funcional, que é uma das características deste tipo de programação.

3.1 HISTÓRICO

Conforme [GOG87a], o OBJ surgiu como uma sintaxe para ***álgebras com erros***, na tentativa de obter um modo simples e uniforme de estender as características de tipos abstratos de dados, expressos algebricamente, incluindo manipulação de erros e de funções parciais. Foi apresentado pela primeira vez por Joseph Goghen em suas aulas na UCLA no outono de 1874. Em 1976 Goghen apresentou o seu primeiro projeto,

usando idéias propostas no projeto *Clear* [BUR81] para módulos parametrizados. A primeira implementação foi feita por Joseph Tardo em um IBM 360/91 da UCLA, em Lisp 1.5, no inverno de 1976, sendo chamada de **OBJ 0**.

Posteriormente surgiram várias outras versões, devido a mudanças de equipamento e ao acréscimo de novas idéias. Joseph Tardo implementou também o **OBJ T**; David Paisted, o **OBJ 1** na SRI, em 1982-83, acrescentando melhorias em alguns aspectos do **OBJ T**; Kokichi Futatsugi e Jean-Pierre Jouannaud implementaram a **OBJ 2** a partir de um projeto do qual também participaram José Meseguer e Joseph Goghen, sendo esta versão baseada não mais na álgebra de erros mas em álgebras classificadas por ordem (*order-sorted algebras*).

A versão que está sendo atualmente desenvolvida na SRI por Timothy Winkler, José Meseguer, Claude e Helene Kirchner e Aristide Megrelis é denominada de **OBJ 3**, apresentando uma sintaxe muito parecida com o **OBJ 2**. Diferenciam-se principalmente por uma implementação diferente, baseada em uma semântica operacional mais eficiente para as álgebras classificadas por ordem, apresentando expressões modulares mais sofisticadas e vistas *default*. **OBJ 3** está escrito em *Common Lisp* e pode ser executado nas estações de trabalho *Sun* da SRI.

Outras implementações do **OBJ** são, ainda segundo [GOG87a], **UMIST-OBJ** no Instituto de Ciência e de Tecnologia da Universidade de Manchester, e **Abstract Pascal** na Universidade de Manchester, ambos escritos em *Pascal*; **MC-OBJ**, na Universidade de Milão, escrito em *C*.

Estão atualmente também em desenvolvimento na SRI outras versões de **OBJ**, chamadas **Eqlog** e **FOOPS**, com extensões em direção a programação relacional e orientada a objetos, respectivamente.

3.2 ESTRUTURA DO OBJ

O **OBJ** apresenta duas unidades sintáticas básicas: uma para declaração e a outra para avaliação. Operacionalmente, as declarações colocam definições em uma base de dados, enquanto que as avaliações reescrevem expressões, usando informações desta mesma base de dados.

Nos próximos capítulos serão apresentadas a sintaxe e a semântica da linguagem **OBJ**, conforme apresentada em [GOG87a], que se refere à versão **OBJ 3**. Quando parecer interessante, será feita referência à diferença desta versão para as anteriores, para as quais foram encontrados muitos exemplos nas referências bibliográficas assinaladas. A sintaxe será apresentada, sempre que possível, em forma de regras de produção da gramática, de acordo com a notação BNF, conforme foi feito em [BAS87].

4. SINTAXE DO OBJ

Uma especificação escrita na metalinguagem **OBJ** tem uma estrutura modular, cujos componentes mais básicos são chamados de **objetos.** Outros módulos que podem constar de uma especificação **OBJ** são as **teorias** e as **visões.**

A **convenção** utilizada para representação de palavras em **OBJ** é a seguinte:

- LETRAS MAIÚSCULAS para representar nomes de objetos, de teorias, de visões, de variáveis, em sua declaração;
- PRIMEIRA LETRA MAIÚSCULA e o resto MINÚSCULAS, nomes de *sortes;*
- LETRAS MINÚSCULAS para palavras reservadas da linguagem.

4.1 DECLARAÇÃO DE OBJETO

Um **objeto** é um módulo que define um tipo abstrato de dado, introduzindo novos *sortes* de dados e operações sobre estes dados, conforme visto no capítulo 2. Em lugar da palavra *tipo* para caracterizar os diferentes tipos de dados, usa-se a palavra *sorte*, utilizada nas especificações encontradas na bibliografia. Um **objeto** encapsula código executável.

4.1.1 ESTRUTURA DA DECLARAÇÃO DE UM OBJETO

A sintaxe dos **objetos** foi inspirada na notação das álgebras iniciais poli-sortidas, pretendendo-se com isto que um **objeto** represente uma álgebra particular. Um **objeto** inicia com a palavra-chave **obj** e termina com **endo** ou **job.** Logo após a palavra-chave **obj** vem o **cabeçalho do objeto,** a palavra **is** e o **corpo do objeto.** O nome do objeto (que aparece no cabeçalho) poderá ser, opcionalmente, repetido após a palavra jbo, com a finalidade de aumentar a legibilidade da especificação.

A declaração de um objeto apresenta-se portanto da seguinte forma:

```
< objeto > ::=  obj <cabeçalho_do_objeto> is

                <corpo_do_objeto>

                Job [ <nome_do_objeto> ]
```

O **cabeçalho de um objeto** é formado pelo nome do objeto seguido de uma lista opcional de parâmetros, como mostra a regra abaixo:

```
< cabeçalho_do_objeto > ::= < nome_do_objeto > [ < parâmetros > ]
< nome_do_objeto > ::= < identificador >
< parâmetros > ::= [ < lista_de_parâmetros > ]
```

```
⟨ lista_de_parâmetros ⟩ ::=
                ⟨ parâmetro ⟩
              | ⟨ parâmetro ⟩ , ⟨ lista_de_parâmetros ⟩
⟨ parâmetro ⟩ ::=
                ⟨ identificador ⟩ :: ⟨ nome_de_teoria ⟩
              | ⟨ lista_de_identificadores⟩ :: ⟨ nome_de_teoria ⟩
```

Os **nomes de objetos** são representados por letras maiúsculas, podendo ser utilizados também caracteres especiais. O nome de um determinado objeto deverá ser único.

A **lista de parâmetros**, quando presente no cabeçalho de um objeto, deverá ser colocada entre colchetes, sendo cada par de parâmetros separado por vírgula. Deverá ser fornecido, para cada parâmetro, seu nome formal seguido do símbolo :: e do nome de uma teoria (as teorias expressam propriedades de módulos e seus interfaces, como será visto mais adiante, no item 4.2). Se um objeto tem, portanto, dois parâmetros, esta lista terá a forma:

[X :: TH1 , Y :: TH2]

Se estas duas teorias forem a mesma, escreve-se simplesmente:

[X Y :: TH]

No item 4.2 serão apresentados exemplos de objetos parametrizados, explicando melhor este aspecto.

No **corpo do objeto** podem ocorrer as seguintes seções:

```
⟨ corpo_do_objeto ⟩ ::= ⟨ declaração_de_sortes ⟩
                        ⟨ declaração_de_sub-sortes ⟩
                        ⟨ declaração_de_importações ⟩
                        ⟨ declaração_de_operações ⟩
                        ⟨ declaração_de_variáveis ⟩
                        ⟨ declaração_de_equações ⟩
```

As declarações de sortes, sub-sortes, importações, de operações e de variáveis representam a sintaxe do objeto; as equações representam a sua semântica.

Cada uma destas seções inicia por sua palavra-chave e devem aparecer na ordem acima citada (com exceção da seção de importações, que poderá vir antes da seção de sub-sortes). A seção ⟨declaração de sortes⟩ não pode ser vazia. Uma **seção** termina quando outra começa, ou quando termina o objeto.

4.1.2 DECLARAÇÃO DE SORTES

Esta parte da declaração define os **sortes** criados neste objeto, que são os tipos dos dados definidos, conforme visto nas especificações algébricas. Apresenta a seguinte sintaxe:

```
< declaração_de_sortes > ::= sorts < lista_de_sortes > .
< lista_de_sortes > ::= < nome_de_sorte > | < lista_de_sortes >
```

Por exemplo:

```
obj BITS1 is
    sorts Bit Bits .
    ...
endo
```

No exemplo acima é definido um objeto chamado **BITS1**; neste objeto são definidos dois *sortes* chamados, respectivamente, **Bit** e **Bits**.

Os nomes de *sortes* podem conter quaisquer caracteres, com exceção dos caracteres vírgula, branco e parêntesis.

Existem alguns ***sortes pré-definidos***, que não poderão ser redefinidos. São êles:

- **INT** - números inteiros
- **NAT** - números naturais
- **RAT** - números racionais
- **FLOAT** - números em ponto flutuante
- **BOOL** - valores lógicos
- **ID** - identificadores
- **QID** - identificadores que iniciam com apóstrofe ('abc)
- **TUPLE** - tuplas

O conceito de *sorte* é proveniente das álgebras heterogêneas ou poli-sortidas, conforme antes mencionado. Para o tratamento de *sortes* que pertencem a mais de um objeto, facilitar o tratamento de erros e utilizar os conceitos de coerção possibilitando a sobreposição de tipos, foram introduzidos os conceitos das álgebras sortidas ordenadas (*order-sorted* algebra). Esta abordagem envolve o uso de relações de ordem parcial em conjuntos e suporta o conceito de múltipla herança.

4.1.3 DECLARAÇÃO DE SUB-SORTES

Este item faz com que elementos pertencentes a um *sorte* também pertençam a outro *sorte*. Isto é, declara um *sorte* como sendo o subconjunto de outro *sorte*. Sua sintaxe é a seguinte:

```
< declaração_de_subsortes > ::= subsorts < lista_de_subsortes > .
< lista_de_subsortes > ::=
        < nome_de_sorte > < < lista_de_sortes >
      | < lista_de_sortes > < <lista_de_sortes>
```

Os elementos destas listas devem ser separados por brancos. O significado desta definição é que o conjunto de objetos do primeiro sorte é um subconjunto (não necessáriamente próprio) do conjunto de objetos do segundo sorte. Na segunda forma de apresentação da <lista de subsortes>, a cada elemento da primeira lista de sortes corresponde um elemento da segunda lista, com o mesmo significado colocado antes.

Por exemplo:

```
obj BITS1 is
    sorts Bit Bits .
    subsorts Bit < Bits .
    ...
endo
```

Neste exemplo é especificado que o *sorte* Bit é um subconjunto do *sorte* **Bits**.

Caso ocorra algum ciclo na definição dos *sortes*, o OBJ é capaz de detectá-lo e dar um aviso ao especificador.

4.1.4 DECLARAÇÃO DE IMPORTAÇÕES

Esta seção indica os **módulos** que serão **importados** para este objeto, podendo portanto ser utilizados por ele. A importação poderá ser feita de tres maneiras diferentes, conforme a utilização que poderá ser efetuada. A sintaxe desta seção é a seguinte:

```
< declaração_de_importações > ::=
        < declaração_de_importação >
      | < declaração_de_importação > < declaração_de_importações >
< declaração_de_importação > ::=
        < modo_de_importação > < lista_de_módulos > .
< modo_de_importação > ::= using | extending | protecting
```

A operação de importação é uma relação transitiva, isto é, se um módulo M importa um módulo M', e se este módulo M' importa um módulo M'', então o módulo M'' também está sendo importado pelo M.

Um módulo só pode ser importado uma vez por um outro módulo.

O significado dos tres modos de importação está relacionado com a semântica da álgebra inicial de objetos. Se um módulo M' estiver sendo importado pelo módulo M, teremos, conforme o modo de importação especificado:

- **protecting** : M' não acrescenta novos itens de dados aos *sortes* de M e também não identifica nenhum item de dado antigo de M;
- **extending** : M' não identifica itens de dados antigos de M;
- **using** : não existe nenhuma garantia.

No modo de importação **using** é feita uma cópia do texto do módulo importado, sem copiar os módulos que este importa. Isto poderá ser feito acrescentando o nome destes módulos à lista dos módulos a serem importados.

Por exemplo:

```
obj LIST-OF-INT is
    sorts List .
    protecting INT .
    subsorts Int < List .
    ...
endo
```

Neste exemplo, o objeto **INT**, no qual é definido o *sorte* **Int**, é importado, com modo protegido.

4.1.5 DECLARAÇÃO DE OPERAÇÕES

As operações permitidas sobre os *sortes* definidos serão especificadas nesta seção. O conjunto de *sorte* e de operações definidas sôbre êle constitui uma **assinatura**, conforme foi visto no capítulo 2.

A sintaxe da declaração de uma operacão é a seguinte:

```
< declaração_de_operações > ::=
        < declaração_de_operação >
      | < declaração_de_operação > < declaração_de_operações >
< declaração_de_operação > ::=
        op < identificação_do_operador >
        : <lista_de_sortes> -> <sorte> [<atributos>].
```

A declaração de um operador é feita, portanto, definindo para ele um **nome** (identificador do operador), uma **lista de *sortes*** que serão os seus argumentos e o *sorte* do seu resultado (***sorte*-alvo** ou *co-aridade*). O identificador da operação aparece na < identificação_do_operador >, juntamente com informações que definem a sua sintaxe. Poderão ser definidos **atributos** para um operador.

Por exemplo, a definição do operador f através da declaração:

```
op f : S1 S2 -> S3 .
```

indica que o resultado da função f (X , Y) é do *sorte* S3, sendo X do *sorte* S1 e Y do S2.

A **lista de sortes** dos argumentos define a aridade da operação, podendo ser vazia.

O **identificador do operador** poderá conter espaços brancos, mas não poderá apresentar o símbolo '_'. Este símbolo será utilizado para definir a sintaxe deste operador, ou seja, a sua forma de apresentação.

4.1.5.1 FORMAS DE APRESENTAÇÃO DE UM OPERADOR

São tres as formas que um operador pode apresentar.

A primeira, mostrada no exemplo anterior, apresenta a ⟨ identificação ⟩ dada somente através do identificador do operador. A sintaxe desta forma é a pré-fixada simples, com os argumentos entre parêntesis e separados por vírgulas.

A segunda forma permite a declaração de notação fixada distribuída. Nesta, a ⟨ identificação ⟩ será formada por uma sequência de caracteres que contém um número de '_' igual à aridade do operador, não podendo esta sequência apresentar espaços brancos, vírgulas ou parêntesis. Os caracteres '_' serão separados por zero ou mais caracteres diferentes de '_'. As sequências de caracteres que separam os '_' deverão ser apresentadas exatamente da forma como aparecem nesta definição quando da utilização do operador, sendo os '_' substituidos por termos do *sorte* correspondente.

Por exemplo, os operadores definidos como:

```
op log_base_ : Int Int -> Int .
op _! : Int -> Int .
```

poderão ser utilizados da seguinte maneira:

```
log N ! base 2
```

Outros exemplos de notação fixada distribuída:

- operador préfixado:

  declaração: `op top_ : Stack -> Int .`

  utilização: `top push(A,B)`

- operador infixado:

  declaração: `op _ + _ : Int Int -> Int .`

  utilização: `2 + 3`

- operador fixado nas extremidades (*outfixed*):

  declaração: `op { _ } : Int -> Set .`

  utilização: `{ 4 }`

- operador distribuído:

  declaração: `op if _ then _ else _ fi : Bool S S -> S .`

  utilização: `if A then B else C fi`

A terceira forma de apresentação de um operador representa transformações entre *sortes* (*coerções*). Nesta, a ⟨identificação⟩ apresenta somente um '_', sem qualquer nome para o operador, e a ⟨ lista de sortes ⟩ contém somente um *sorte*.

Por exemplo, os operadores abaixo:

```
op _ : Real -> Cpx .
op _ : Int -> List-of-Int .
```

indicam que os *reais* poderão ser transformados em *complexos* e os *inteiros* em *lista-de-inteiros*. Eventuais duplicações ou ciclos destas definições serão verificados pelo **OBJ**.

4.1.5.2 DECLARAÇÃO DE ATRIBUTOS PARA OPERADORES

Ao declarar um operador, pode-se definir certas propriedades para a operação através de **atributos** incluídos na declaração. Estas propriedades incluem axiomas tais como de associatividade, comutatividade, *idempotência* e identidade, com concequências sintáticas e semânticas; e outras que afetam ordem de avaliação, reconhecimento, etc. A maneira de apresentar estes atributos é a seguinte:

⟨ atributos ⟩ ::= [⟨ lista de atributos ⟩]

Observação: neste caso os colchetes representam unidades sintáticas da linguagem e não a notação utilizada anteriormente para classe sintática opcional.

Os atributos que poderão ser utilizados são os seguintes:

```
⟨ lista_de_atributos ⟩ ::=
                    ⟨ atributo ⟩
                  | ⟨ atributo ⟩ ⟨ lista_de_atributos ⟩
⟨ atributo ⟩ ::=  assoc
                | comm
                | idem
                | id : ⟨ expressão ⟩
                | equality
                | prec ⟨ inteiro ⟩
                | memo
                | strat ( ⟨ lista_de_inteiros ⟩ )
⟨ lista_de_inteiros ⟩ ::=
                    ⟨ inteiro ⟩
                  | ⟨inteiro ⟩ , ⟨ lista_de_inteiros ⟩
```

O primeiro atributo, **assoc**, define uma operação que apresenta a propriedade associativa. Por exemplo, o operador lógico **or** é um operador binário infixado e possui a propriedade associativa, o que pode ser definido da seguinte maneira:

```
op _ or _ : Bool Bool -> Bool [assoc] .
```

A declaração deste atributo vai simplificar o trabalho do reconhecedor, uma vez que ele poderá utilizar, neste caso, um número menor de parêntesis durante o reconhecimento.

Do mesmo modo poderá ser utilizado o atributo **comm**, que representa a propriedade comutativa.

O atributo idem declara operações *idempotentes*.

Um atributo de identidade (id:) pode ser declarado para uma operação binária que tenha cada *sorte* em sua lista de argumentos menor que o *sorte-alvo*. Por exemplo, em:

```
op _ or _ : Bool Bool -> Bool [assoc id:false] .
```

o atributo **id:false** fornece o efeito das equações de identidade (B **or false** = B) e (**false or** B = B). Um atributo de identidade pode ser um *termo* ou uma *constante*.

O atributo **equality** pode ser fornecido a qualquer operação lógica binária. Todo *sorte* definido tem uma operação pré-definida de igualdade, definida por:

```
op _ == _ : S S -> Bool .
```

Uma operação à qual se fornece o atributo **equality** apresentará esta mesma igualdade. Por exemplo:

```
op _ iff _ : Bool Bool -> Bool [equality] .
```

O atributo **prec** fornece uma ordem de precedência para o reconhecimento da operação. Ele deve ser seguido de um número inteiro, cujo valor determinará esta precedência; quanto menor este valor, maior será a precedência. Por exemplo, o objeto pré-definido **INT** poderia ter a seguinte precedência de operadores:

```
op _ + _ : Int Int -> Int [assoc prec 8] .
op _ * _ : Int Int -> Int [assoc prec 5] .
```

de modo que a expressão **A + B * C** seria avaliada do modo habitual, ou seja, **A + (B * C)**.

O atributo **memo** faz com que os resultados da avaliação de qualquer termo desta operação sejam precedidos do salvamento da operação propriamente dita. Deste modo, a operação de redução não será repetida no caso deste termo aparecer novamente. Qualquer operação poderá receber o atributo **memo**.

Finalmente, o atributo **strat** define uma estratégia para a ordem de avaliação da operação. Esta estratégia é definida por uma sequência de números naturais, determinando onde e em que ordem as regras deverão ser aplicadas. Por exemplo, a operação: **if_then_else_fi** tem estratégia **(1 0)**, que significa que o primeiro argumento deverá ser reduzido e só depois é que as regras deverão ser aplicadas no topo da árvore sintática(indicado por 0). Já a operação **_ + _** poderia ser de-

finida do seguinte modo:

```
op _ + _ : Int Int -> Int [strat (1 2 0 )] .
```

o que significa que os dois argumentos deverão ser avaliados antes de ser efetuada a adição sôbre êles. Se não for definida nenhuma estratégia de avaliação, o OBJ usa uma *default*, que consiste em avaliar primeiro os argumentos que não possuem termos variáveis.

Nas versões anteriores do OBJ, havia distinção entre tres tipos diferentes de operadores, identificados por palavras-chave diferentes. Estes tres tipos eram os seguintes:

i) operadores para tratar situações comuns - OK-OPS;
ii) operadores para tratar situações de erro - ERR-OPS;
iii) operadores para tratar situações de recuperação - FIX-OPS.

A declaração de todos os operadores de cada um dos tipos era feita em uma só declaração, terminando o conjunto com o início da próxima declaração.

4.1.6 DECLARAÇÃO DE VARIÁVEIS

Nesta seção devem ser declaradas as variáveis que serão usadas nas equações da próxima seção. A sintaxe da declaração de variáveis é a seguinte:

```
< declaração_de_variáveis > ::=
                    var < identificador > : < sorte> .
                  | vars < lista_de_identificadores > : < sorte > .
< lista_de_identificadores > ::=
                    < identificador >
                  | < identificador > < lista_de_identificadores >
```

Por exemplo:

```
vars I J K : Nat .
```

Não é permitido declarar variáveis com nomes iguais ao nome de algum operador nem a palavras reservadas.

4.1.7 DECLARAÇÃO DE EQUAÇÕES

Esta parte define a semântica dos objetos, através de equações escritas em modo declarativo e interpretadas operacionalmente como regras de reescrita, que trocam instâncias obtidas por substituição no lado esquerdo pelas correspondentes instanciações do lado direito das regras.

A sintaxe desta seção é a seguinte:

```
< declaração_de_equações > ::=
        < declaração_de_equação >
      | < declaração_de_equação > < declaração_de_equações >
< declaração_de_equação > ::=
        eq < exp1 > = < exp2 > .
      | ceq < exp1 > = < exp2 > if < exp_lógica > .
      | eq-as < sort > < exp1 > = < exp2 > .
      | ceq-as < sort > < exp1 > = < exp2 > if < exp_lógica > .
```

As expressões < exp1 > e < exp2 > podem ser quaisquer expressões OBJ, utilizando os operadores definidos anteriormente. A < exp_lógica > deve ser do *sorte* **Bool**. Se nestas expressões aparecerem variáveis que não foram declaradas na seção anterior, o reconhecedor determinará o seu *sorte*, adotando o mais alto *sorte* que faça sentido na expressão.

As declarações que iniciam com **ceq** denotam regras de reescrita condicionais, que serão executadas somente no caso da expressão lógica ser verdadeira. Em alguns casos é interessante declarar um *sorte* para a equação, para garantir que seu reconhecimento seja conforme o desejado. Nestes casos usa-se as regras que iniciam com **eq-as** ou **ceq-as**.

Por exemplo:

```
obj EX is
    ...
    vars I J K : Int .
    eq I < J = true .
    ceq I < K = true if I < J .
endo
```

Também na declaração de equações existe uma diferença substancial desta versão do OBJ para as anteriores. Naquelas existiam dois tipos diferentes de equações, definidos por palavras-chave diferentes. São elas:

i) equações que se aplicam a expressões do tipo **OK** (que envolvem operadores do tipo **OK-OPS**);
ii) equações que se aplicam em expressões de tipo **ERR** (que envolvem operadores do tipo **ERR-OPS**).

4.1.8 EXEMPLO DE DECLARAÇÃO DE OBJETO

O objeto abaixo define uma lista de inteiros:

```
obj LIST_OF_INT is
    sorts List .
    protecting INT .
    subsorts Int < List .
    op _ _ : Int List -> List .
    op length_ : List -> Int .
    var I : Int .
    var L : List .
    eq length I = 1 .
    eq length I L = 1 + length L .
endo
```

Neste objeto é definido um *sorte* chamado List (lista) e importado o *sorte* pré-definido Int (inteiro), de modo protegido. A terceira linha declara que o *sorte* Int é um sub-sorte de List. São definidos dois operadores. O primeiro transforma um inteiro seguido de uma lista em uma lista; o segundo calcula o comprimento de uma lista. As equações determinam a semântica do objeto. A primeira determina que o comprimento de um inteiro é 1 e a segunda, que o comprimento de uma lista precedida de um inteiro (separando o primeiro elemento da lista) é dado pelo comprimento dalista mais uma unidade.

4.2 DECLARAÇÃO DE TEORIA

As teorias expressam propriedades e interfaces de módulos. Tem a mesma estrutura dos objetos - tem *sortes*, sub-*sortes*, operações, variáveis e equações, podem importar outras teorias ou objetos, podem ser parametrizados e podem ter vistas. Entretanto, as teorias não possuem código executável, definindo apenas propriedades. Semanticamente, uma teoria tem diversos modelos, que são todas as álgebras que a satisfazem, enquanto que um objeto tem somente um modelo, sua álgebra inicial.

A sintaxe da declaração de uma teoria é, portanto:

```
< teoria > ::= th < cabeçalho_da_teoria > is
               <corpo_da_teoria >
               endth
< cabeçalho_da_teoria > ::=
               <nome_de_teoria > [< lista_de_parâmetros >]
< nome_de_teoria > ::= < identificador >
< corpo_da_teoria > ::= <corpo_do_objeto >
```

Por exemplo, a declaração de uma teoria trivial que requer somente um *sorte* é a seguinte:

```
th TRIV is
   sorts Elt
endth
```

Na teoria declaramos a estrutura e as propriedades que um determinado módulo deve satisfazer. Quando forem declarados módulos parametrizados, cada parâmetro definido deverá ser associado a uma teoria, que vai então definir a estrutura e as propriedades que êste parâmetro deverá apresentar. Desta maneira, será possível expressar propriedades semânticas, tais como associatividade de uma operação, como parte do interface de um módulo que servirá de parâmetro para um objeto. Este é o caso da teoria definida no exemplo abaixo:

```
th MONOID is
   sorts M .
   op _*_ : M M -> M [assoc id: e] .
endth
```

Outro exemplo de declaração de teoria:

```
th EQV is
   sorts Elt .
   op_eq_ : Elt Elt -> Bool .
   vars E1 E2 E3 : Elt .
   eq (E1 eq E1) = true .
   eq (E1 eq E2) = (E2 eq E1).
   ceq (E1 eq E3) = true if (E1 eq E2) and (E2 eq E3) .
endth
```

Esta teoria, que define uma relação equivalente, apresenta uma operação lógica infixada binária, denotada _eq_, que é reflexiva, simétrica e transitiva. Todos os modelos que possuirem estas três propriedades, satisfazem esta teoria.

Como exemplo de declaração de um objeto parametrizado, apresentamos a definição de uma lista, que utiliza a teoria **TRIV** mostrada antes:

```
obj LIST[X :: TRIV] is
    sorts List NeList .
    subsorts Elt < NeList < List .
    op _ _ : List List -> List [assoc id:nil] .
    op _ _ : NeList List -> NeList [assoc] .
    op head_ : NeList -> Elt .
    op tail_ : NeList -> List .
    op empty?_ : List -> Bool .
    var X : Elt .
    var L : List .
    eq head X L = X .
    eq tail X L = L .
    eq empty? L = L == nil .
endo
```

Uma teoria pode também ser parametrizada, conforme mostra a sua sintaxe acima. As teorias que servirão de base para os parâmetros de-

verão ter sido definidas anteriormente. Por exemplo, a teoria definida abaixo:

```
th ABC [X :: Y] is
   ...
endth
```

possue um parâmetro X que deve satisfazer à teoria Y, que deve ter sido definida antes.

4.3 DECLARAÇÃO DE VISÕES

Como foi visto na seção anterior, uma teoria pode ter vários modelos - todas as álgebras que a satisfazem. Isto significa que um módulo poderá satisfazer uma determinada teoria de várias maneiras diferentes. Entretanto, um objeto somente poderá ter um modelo - sua álgebra inicial. Existe uma maneira de definir qual o modelo que deverá ser utilizado por um objeto (ou por um módulo em geral) para satisfazer uma determinada teoria. Isto é feito através da definição de **visões** de uma teoria para um módulo.

Uma **visão** consiste do mapeamento dos *sortes* de uma teoria para os *sortes* do módulo (preservando a relação de *sub-sortes*) e do mapeamento das operações desta teoria para as operações do módulo (preservando aridade, valores de *sortes* e atributos das operações, de tal modo que toda equação da teoria seja satisfeita por todo modelo do módulo.

Uma **visão** v de uma teoria T para um módulo M é indicada pela notação:

v: T = > M

A sintaxe da declaração de uma visão é a seguinte:

```
< declaração_de_visão > ::=
  view <identificador> from <nome_de_teoria>> to <nome_de_modulo> is
      < mapeamento_de_sortes >
      < mapeamento_de_operações >
  endv
< nome_de_modulo > ::= < nome_do_objeto > | < nome_de_teoria >
< mapeamento_de_sortes > ::=
                < map_sort >
              | < map_sorte > < mapeamento_de_sortes >
< map_sorte > ::= sort < nome_de_sorte > to < nome_de_sorte > .
< mapeamento_de_operações > ::=
                < map_operação >
              | < map_operação > < mapeamento_de_operação >
< map_operação > ::=
  op < identificação_do_operador > to < identificação_de_operador > .
```

De acordo com a sintaxe apresentada acima, os mapeamentos serão da forma:

```
sort S1 to S1' .
sort S2 to S2' .

op o1 to o1' .
op o2 to o2' .
```

No exemplo abaixo é definida uma visão NATD, da teoria PREORD para o módulo NAT, usando a relação de divisibilidade para satisfazer a teoria:

```
view NATD from PREORD to NAT is
     sort Elt to Nat .
     op _<=_ to divises .
endv
```

As visões podem ser definidas para módulos parametrizados, como mostra o exemplo:

```
view LISTM from MONOID to LISTC X::TRIV ] is
     op _*_ to _ _ .
     op e to nil .
endv
```

Também é possível definir visões parametrizadas, do mesmo modo mostradq na definição dos objetos.

O próximo exemplo apresenta uma visão que envolve operações derivadas:

```
view NATG from PREORD to NAT is
     vars L1 L2 : Nat .
     op L1 <= L2 to L2 <= L1 .
endv
```

Esta visão faz o mapeamento do operador <= de PREORD para o operador >= de NAT, sendo utilizadas as variáveis L1 e L2 para marcar os lugares dos argumentos nos operadores.

A declaração de visões pode ser feita externamente aos objetos, no corpo do programa OBJ ou então, internamente a um objeto. A forma interna é sintaticamente idêntica à externa, mapeando a teoria para este objeto. Por exemplo:

```
obj ABC [ X :: TEORIA ] is
    sort Abc.
    ...
    view V is sort A to Abc . op + to mais . endv
endo
```

4.4 COMENTÁRIOS

Podem ser colocados comentários entre as principais seções de um módulo, ou entre suas partes. Os comentários só não podem ocorrer no meio de uma equação, de uma declaração de um operador ou de uma lista de *sortes*. A forma de representar os comentários (no **OBJ 0**) é usando como delimitadores o conjunto de símbolos ***.

Por exemplo:

*** ISTO E UM COMENTARIO ***

Em [GOG87a], de onde foi retirada a sintaxe do **OBJ 3** aqui apresentada, não foi encontrado nemhuma referência à comentários. Supomos, portanto, que a sintaxe destes seja ainda a anterior.

5. UTILIZAÇÃO DA LINGUAGEM OBJ

O OBJ apresenta-se em forma interpretativa. O processo interpretativo representa uma vantagem bastante grande, pois possibilita a construção incremental de uma especificação, sendo cada passo da definição imediatamente testado.

5.1 SEMÂNTICA OPERACIONAL

Como citado no capítulo anterior, o OBJ apresenta uma semântica operacional baseada na interpretação das equações como regras de reescrita. O comando para avaliar uma expressão apresenta a seguinte sintaxe:

```
< comando de redução > ::=
        reduce < expressão > endr
      | reduce-in < módulo > < expressão >
      | reduce-as < sorte > < expressão > endr
      | reduce-in-as < módulo > < sorte > < expressão > endr

< expressão > ::=
```

Este comando solicita a redução de uma expressão, que será avaliada no contexto do último módulo que foi fornecido ao sistema, a menos que seja fornecido explicitamente o nome do módulo a ser utilizado (segunda e quarta regras apresentadas acima). Também poderá ser fornecido o nome do *sorte* a ser utilizado durante o reconhecimento da expressão (terceira e quarta regras apresentadas acima).

A operação de redução efetua uma comparação entre a expressão fornecida e o lado esquerdo das equações do módulo, procurando uma equação cujo lado esquerdo seja idêntica a alguma parte da expressão. Ao encontrá-la, esta parte é substituída pelo lado direito da equação, seguindo a avaliação com novas substituições.

Por exemplo, consideremos que o último módulo fornecido seja aquele apresentado no item 4.1.8. que define uma lista de inteiros. O resultado do comando :

```
reduce length  17  -4  329 endr
```

é apresentado pelo sistema da seguinte maneira:

```
result Int : 3
```

A avaliação deste comando corresponde à seguinte sequência de aplicações de regras de reescrita:

```
length  17  -4  329  =>
1  +  length  -4  329  =>
1  +  ( 1  +  length  329 )  =>
1  +  ( 1  +  1 )  =>
```

No primeiro passo é utilizada a regra:

```
eq length L I = 1 + length L .
```

correspondendo I a 17 e L a -4 329. O segundo passo utiliza novamente esta regra, agora valendo I -4 e L 329. Neste passo, o valor 329 é encarado como uma lista, uma vez que Int é um subsorte de List. O terceiro passo utiliza a regra:

```
eq length I = 1 .
```

O quarto passo avalia a expressão usando as funções aritméticas pré-definidas de INT.

Nas versões do OBJ anteriores a esta aqui apresentada, o comando de avaliação de expressões tinha as seguintes formas:

```
RUN ‹ expressão › NUR
```

e

```
( ‹ expressão › )
```

Alguns casos particulares podem complicar esta maneira bastante simples de efetuar as reduções. Entre eles:

- possibilidade da não unicidade da expressão resultante da redução;
- condicionais;
- coerções;
- utilização de condições de erro durante a avaliação;
- possibilidade de, sob alguma condição, originar um processo que não pare.

Apesar de não existir ainda, em 1978, uma teoria matemática completa justificando os algoritmos utilizados pelo interpretador OBJ para tratar destas condições, os principais pontos são discutidos em profundidade em [GOG78a].

5.2 SEMÂNTICA DENOTACIONAL

A semântica denotacional de uma linguagem fornece significado preciso aos programas, em uma maneira conceitual simples e clara. A semântica denotacional do OBJ é algébrica, conforme o conceito algébrico de tipos abstratos de dados. Deste modo, o significado de um objeto é uma álgebra, ou seja, uma coleção de conjuntos com funções entre eles.

OBJ pode ser encarado também como uma linguagem de programação em lógica. Estas devem apresentar, segundo [GOG87a], as seguintes características:

- programação funcional, onde a lógica é representada por algum tipo de lógica equacional;
- programação relacional, onde a lógica é a de cláusulas de Horn, de primeira ordem;
- programação de multiparadigmas, pela unificação de sistemas lógicos.

OBJ possui as características de programação funcional, apresentadas de uma maneira simples, elegante e logicamente precisa, baseadas em álgebras classificadas por ordem e em módulos parametrizados.

Nas linguagens de programação em lógica, como o OBJ, a teoria de provas do sistema lógico implícito se aplica diretamente aos programas, não sendo necessários formalismos complexos tais como a semântica denotacional de Scott-Strachey e a semântica axiomática de Hoare. O conceito de álgebra inicial utiliza a álgebra inicial única como o modelo mais representativo das equações, podendo existir outros modelos.

5.3 ESTRUTURA HIERÁRQUICA

Um programa OBJ tem uma estrutura acentuadamente modular, o que facilita o seu desenvolvimento, a sua depuração e a reusabilidade de módulos. Estes módulos possuem uma estrutura hierárquica de dependência entre eles, determinando com segurança quais os módulos que podem ser utilizados por outros módulos. Segundo [GOG87a], o **contexto** de um módulo é determinado pela conjunto de módulos que ele pode utilizar, juntamente com as relações de dependência entre eles. Para que esta dependência fique perfeitamente explicitada, sempre que um módulo utilizar dados ou operações de um outro módulo, este deverá ser importado explicitamente, ou então ter sido declarado antes dele. O programa desenvolvido desta maneira tem uma estrutura representada por um **grafo acíclico** de módulos. Visto assim, o **contexto** é um grafo de módulos, onde um arco dirigido indica que o módulo de destino importa o de origem, e que o contexto de um determinado módulo é o subgrafo de outros módulos do qual ele é a raiz. Módulos parametrizados também podem ser rpresentados nesta estrutura.

6. EXEMPLO DE APLICAÇÃO

O exemplo que será apresentado tem o objetivo de mostrar a potencialidade da especificação algébrica e a sua implementação na linguagem **OBJ**. A especificação algébrica deste exemplo se encontra em [CLE86]. Trata-se da especificação de um editor de texto que possui apenas alguns comandos simples. Os comandos do editor **editam** um arquivo, que é formado por uma sequência linear de registros. Para nossos propósitos, **registros** e **inteiros** são considerados objetos de dados primitivos, não sendo definidos nesta especificação.

O editor executa modificações no arquivo através de operações de **inserção, substituição e eliminação** de registros. Existem, ainda, operações de navegação no texto . As operações atuam no **registro corrente**, que é um registro particular deste arquivo. A descrição de cada uma das operações é a seguinte:

- **newfile** : inicialização; criação de um novo arquivo, sem a presença de registro corrente.

- **insert** : insere um novo registro após o registro corrente; o novo registro passa a ser o registro corrente.

- **replace** : substitui o registro corrente por um novo registro.

- **delete** : remove o registro corrente, passando o próximo registro a ser o registro corrente.

- **advance** : o próximo registro passa a ser o registro corrente.

- **back** : o registro anterior passa a ser o registro corrente.

Além destas operações existe uma operação oculta chamada **M**, que pode ser considerada como uma maneira de visualizar o arquivo em duas partes. A primeira parte consiste de todos os registros precedentes ao registro corrente, mais o registro corrente. A segunda parte consiste de todos os registros seguintes ao registro corrente, em ordem reversa.

A seguir apresentaremos as especificações algébrica e em **OBJ** deste editor de textos.

A especificação algébrica deste editor é a seguinte:

```
type EDITOR

     function

          newfile :              ∅              ---> FILES
          insert  : RECORDS x FILES             ---> FILES
          replace : RECORDS x FILES             ---> FILES
          delete  : FILES                       ---> FILES
          advance : FILES                       ---> FILES
          back    : FILES                       ---> FILES
          M       : FILES x FILES               ---> FILES

     axioms for x,y ∈ FILES, r,s ∈ RECORDS let

          newfile = M(newfile,newfile)
          insert(r,M(x,y)) = M(insert(r,x),y)
          replace(r,M(insert(s,x),y)) = M(insert(r,x),y)
          replace(r,M(newfile,y)) = M(newfile,y)
          delete(M(insert(r,x),y)) = M(x,y)
          delete(M(newfile,y)) = M(newfile,y)
          advance(M(x,insert(r,y))) = M(insert(r,x),y)
          advance(M(x,newfile)) = M(x,newfile)
          back(M(insert(r,x),y)) = M(x,insert(r,y))
          back(M(newfile,y)) = M(newfile,y)

endtype EDITOR
```

A especificação em linguagem OBJ deste mesmo editor é a seguinte:

```
obj EDITOR is
     sorts Files .
     protecting INT RECORDS .
     op newfile   :                  -> Files .
     op insert    : Records Files    -> Files .
     op replace   : Records Files    -> Files .
     op delete    : Files            -> Files .
     op advance _ : Files            -> Files .
     op back _    : Files            -> Files .
     op M         : Files Files      -> Files .
     vars S,R : Records .
     var  X,Y : Files .
     eq newfile = M(newfile,newfile) .
     eq insert(R,M(X,Y)) = M(insert(R,X),Y) .
     eq replace(R,M(insert(S,X),Y)) = M(insert(R,X),Y) .
     eq replace(R,M(newfile,Y)) = M(newfile,Y) .
     eq delete(M(insert(R,X),Y)) = M(X,Y) .
     eq delete(M(newfile,Y)) = M(newfile,Y) .
     eq advance M(X,insert(R,Y)) = M(insert(R,X),Y) .
     eq advance M(X,newfile) = M(X,newfile) .
     eq back M(insert(R,X),Y) = M(X,insert(R,Y)) .
     eq back M(newfile,Y) = M(newfile,Y) .
endo
```

7. CONCLUSÃO

A importância da utilização de especificações formais no desenvolvimento de *software* não é mais questionada. Entretanto, qual o grau de formalismo que uma especificação deve apresentar, isto ainda sucita dúvidas. Como foi citado na introdução, quanto maior o grau de formalismo de uma especificação, mais difícil se torna a sua compreensão, a sua legibilidade. Especificações utilizando métodos deste tipo podem facilmente apresentar erros. Entretanto, quanto menor o formalismo, menos exata a especificação, no sentido de representação do problema.

O método apresentado, OBJ, embora apresente alto grau de formalismo, uma vez que se trata de uma versão de especificação algébrica, é executável. Deste modo, a deficiência de clareza é suprida pela imediata detecção dos erros cometidos durante a especificação, chegando-se assim a um resultado ideal.

A facilidade de construção de um protótipo do sistema que se está especificando, o que é feito pelo OBJ, permite não só a obtenção de especificações corretas, mas também a opção de alterar a especificação em vistas do protótipo alcançado, com a finalidade de obter outra especificação que melhor se adapte ao problema.

ANEXO

LISTAGEM DA EXECUÇÃO DA ESPECIFICAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO FUN EM OBJ T

```
>IN FUN NI

*** THIS SPECIFICATION OF THE FUN LANGUAGE USES PARAMETERIZED OBJECTS
LIST AND ARRAY FROM THE OBJ LIBRARY ***

     IM (LIST => IDENT)/ BOOL INT
     SORTS (LIST => IDENT)
           (ELM => ID)
     MI

     IM (ARRAY => ENV)/ IDENT INT BOOL
     SORTS (INDEX => IDENT)
           (ELM => INT)
           (ARRAY => ENV)
     MI

     IM (LIST => IDENTL)/ IDENT INT BOOL
     SORTS (ELM => IDENT)
           (LIST => IDENTL)
     OPS   (__ : LIST LIST -> LIST => _:_)
     MI

     IM (LIST => INTL)/ INT BOOL
     SORTS (ELM => INT)
           (LIST => INTL)
     OPS   (__ : LIST LIST -> LIST => _:_)
     MI

     OBJ EXP / ENV INT BOOL IDENT
     SORTS INTEXP BOOLEXP
     OK-OPS
         _ : INT -> INTEXP
         _ : IDENT -> INTEXP
         _ : BOOL -> BOOLEXP
         _AND_ : BOOLEXP BOOLEXP -> BOOLEXP
         _OR_ : BOOLEXP BOOLEXP -> BOOLEXP
         NOT_ : BOOLEXP -> BOOLEXP
         _<_ : INTEXP INTEXP -> BOOLEXP
         _=_ : INTEXP INTEXP -> BOOLEXP
         IF_THEN_ELSE_FI : BOOLEXP INTEXP INTEXP -> INTEXP
         _+_ : INTEXP INTEXP -> INTEXP
         _-_ : INTEXP INTEXP -> INTEXP
         _*_ : INTEXP INTEXP -> INTEXP
         _[_].[_] : INTEXP ENV -> INT
         _[_].[_] : BOOLEXP ENV -> BOOL
     ERR-OPS
         NIL-ID : -> INTEXP
     VARS
         INT : INT
         BOOL : BOOL
         ID : IDENT
         EX EX' : INTEXP
         BX BX' : BOOLEXP
         ENV : ENV
```

```
OK-EQNS
    ((| INT |.[ ENV ])= INT)
    (| ID |.[ ENV ] = ENV [ ID ])
    (|(EX + EX')|.[ ENV ] =(| EX |.[ ENV ])+(| EX' |.[ ENV ]))
    (|(EX - EX')|.[ ENV ] =(| EX |.[ ENV ])-(| EX' |.[ ENV ]))
    (|(EX * EX')|.[ ENV ] =(| EX |.[ ENV ])*(| EX' |.[ ENV ]))
    (| BOOL |.[ ENV ] = BOOL)
    (| EX < EX' |.[ ENV ] = | EX |.[ ENV ] < | EX' |.[ ENV ])
    (| EX = EX' |.[ ENV ] = | EX |.[ ENV ] == | EX' |.[ ENV ])
    ((| BX AND BX' |.[ ENV ])=(| BX |.[ ENV ])AND(| BX' |.[ ENV ]))
    ((|(BX OR BX')|.[ ENV ])=(| BX |.[ ENV ])OR(| BX' |.[ ENV ]))
    ((|(NOT BX)|.[ ENV ])= NOT(|BX |.[ ENV ]))
    (| IF BX THEN EX ELSE EX' FI |.[ ENV ] = IF(| BX |.[ ENV
        ])THEN(| EX |.[ ENV ])ELSE(| EX' |.[ ENV ]) FI)
ERR-EQNS
    (NIL = NIL-ID)
JBO

OBJ STMT / EXP
SORTS STMT
OK-OPS
    _ _ : STMT STMT -> STMT (ASSOC)
    _:=_ : IDENT INTEXP -> STMT
    WHILE_DO_OD : BOOLEXP STMT -> STMT
    |_|.[_] : STMT ENV -> ENV
VARS
    S S' : STMT
    ID : IDENT
    EX : INTEXP
    BX : BOOLEXP
    ENV : ENV
OK-EQNS
    (| ID := EX |.[ ENV ] = PUT(ID,| EX |.[ ENV ],ENV))
    (|(S ; S')|.[ ENV ] = | S' |.[(| S |.[ ENV ])])
    (|(WHILE BX DO S OD)|.[ ENV ] = IF(| BX |.[ ENV ])THEN(|
        WHILE BX DO S OD |.[(| S |.[ ENV ])])ELSE ENV FI)
JBO

OBJ FUN / STMT IDENTL INTL
SORTS INIT1 INIT FUN
OK-OPS
    NILINIT : -> INIT
    _INITIALLY_ : IDENT INTEXP -> INIT1
    _ : INIT1 -> INIT
    _;_ : INIT INIT -> INIT (ASSOC)
    FUN_<_>VARS_BODY_NUF : IDENT IDENTL INIT STMT -> FUN
    |_|.[_|_].[_] : IDENTL INTL ENV -> ENV
    |_|.[_] : INIT ENV -> ENV
    |_|.[_].<_> : FUN ENV INTL -> ENV
    |_|.<_> : FUN INTL -> INT
ERR-OPS
    WRONG#ARGS : -> ENV
VARS
    ID ID' : IDENT
    IDL : IDENTL
    INT : INT
    INTL : INTL
    EX : INTEXP
    INIT1 : INIT1
    INIT : INIT
    STMT : STMT
    ENV : ENV
```

BIBLIOGRAFIA

[BAS87] BASTOS,F.E.A. OBJ = Uma Linguagem Formal para Especificação Algébrica. Porto Alegre, CPGCC/UFRGS, 1987.

[BOE80] BOEHM,B. Software Engineering As It Is. In: Software Engineering, FREEMAN,H. & LEWIS,B.M, Academic Press, 1980.

[BUR81] BURSTALL,R. & GOGHEN,J.A. , An Informal Introduction to Specifications using CLEAR In: GEHANI,N. & McGETTRICK,A. eds. Software Specification Techniques. Workingham, Addison Wesley, p.363-390, 1986.

[CAS86] CASTILHO,J.M.V. Especificações Formais e Sistemas de Bancos de Dados. Buenos Aires,Kapelusz, 1987.

[CLE86] CLEAVELAND,J.C. An Introduction to Data Types. Reading, Addison-Wesley, 1986.

[COH86] COHEN,B. & HARWOOD,W.I. The Specification of Complex Systems. New York, Addison-Wesley, 1986.

[DUC85] DUCE,D.A. & FIELDING,E.V.C. Formal Specification = A Comparision of Two Techniques. Chilton, Rutherford Appleton Laboratory, 1985.

[GOG75] GOGEN,J.A.; THATCHER,J.W.; WAGNER,E.G.; WRIGHT,J.B. Abstract Data Types as Initial Algebras and the Correctness of Data Representations In: CONFERENCE on COMPUTER GRAPHICS PATTERN RECOGNITION and DATA STRUCTURE, Los Angeles, May 14-16, 1975. Proceedings. New York, IEEE, c1975. p.89-93.

[GOG77] GOGHEN,J.A.; THATCHER,J.W.; WAGNER,E.G.; WRIGHT,J.B. Initial Algebra Semantics and Continuous Algebras. Journal of the Association for Computing Machinery, 24(1):68-95, Jan. 1977.

[GOG78a] GOGHEN,J.A. Some Design Principles and Theory for OBJ-0, a Language for Expressing and Executing Algebraic Specifications of Programs. In: INTERNATIONAL CONFERENCE on MATHEMATICAL STUDIES of INFORMATION PROCESSING, KIOTO, Aug.23-26, c1978. Proceedings. Berlin, Springer-Verlag, 1978. p.429-75. Lecture Notes in Computer Science 75.

```
      OK-EQNS
          ((| ID INITIALLY EX |.[ ENV ])=(| ID := EX |.[ ENV ]))
          (| INIT1 ; INIT |.[ ENV ] = | INIT |.[ | INIT1 |.[ ENV ] ])
          ((| ID |.| INT |.[ ENV ])=(| ID := INT |.[ ENV ]))
          ((|(ID ; IDL)|.|(INT ; INTL)|.[ ENV ])=(| IDL |.| INTL |.[(|
                ID |.| INT |.[ ENV ])]))
          (| FUN ID < IDL >VARS NILINIT BODY STMT NUF |.[ ENV ].< INTL
                > = | STMT |.[ ENV ])'
          (| FUN ID < IDL >VARS INIT BODY STMT NUF |.[ ENV ].< INTL > =
                | STMT |.[ | INIT |.[ | IDL |.| INTL |.[ ENV ] ] ])
          (| FUN ID < IDL >VARS INIT BODY STMT NUF |.< INTL > = | FUN
                ID < IDL >VARS INIT BODY STMT NUF |.[ NILARR ].< INTL >
                [ ID ])
      ERR-EQNS
          (| IDL |.| INTL |.[ ENV ] = WRONG#ARGS IF NOT(LENGTH IDL ==
                LENGTH INTL)OR INTL == NIL)
      JBO

=End of file=

>Interrupt (Help=?): ^X
@SAVE (on file) FUN.EXE
FUN.EXE.1 Saved
@VDI FUN.EXE
   PS:<OBJT>
FUN.EXE.1;P775200      179 91648(36)  26-Jun-82 19:44:20 GOGUEN
@
@;THIS SHOWS THAT THE FILE FUN.EXE HAS JUST BEEN CREATED
@;WE NOW RUN IT AS IF IT WERE A UTILITY PROGRAM
@;AND READ IN A FILE FUNP.OBJ OF TEST CASES FOR FUN
@
@FUN

>IN FUNP NI

*** TEST PROGRAMS FOR THE FUN DEFINITION ***

*** POW(N ; M) GIVES THE NTH POWER OF M FOR N POSITIVE OR 0 ***
RUN | FUN 'POW < 'N ; 'M >VARS 'POW INITIALLY 1 BODY WHILE 0 < 'N DO
                'POW := 'POW * 'M ; 'N := 'N - 1 OD NUF |.< 4 ; 2 > NUR
AS INT: 16

*** FACTORIAL OF N ***
RUN | FUN 'FAC < 'N >VARS 'FAC INITIALLY 1 ; 'I INITIALLY 0 BODY WHILE
                'I < 'N DO 'I := 'I + 1 ; 'FAC := 'I * 'FAC OD NUF |.< 5 > NUR
AS INT: 120

*** MAX GIVES THE MAXIMUM OF A LIST OF THREE NUMBERS ***
RUN | FUN 'MAX < 'A ; 'B ; 'C >VARS 'N INITIALLY 2 BODY 'MAX := 'A ;(WHILE 0
                < 'N DO('N := 'N - 1 ; 'X := 'B ; 'B := 'C ;('MAX := IF 'X <
                'MAX THEN 'MAX EL SE 'X FI))OD)NUF |.< 3 ; 123 ; 32 > NUR
AS INT: 123
```

[GOG78b] GOGHEN,J.A.; THATCHER,J.W.; WAGNER,E.G. An Initial Algebra Approach to the Specification Correctness and Implementation of Abstract Data Types. In: Current Trends in Programming Methodology. Englewood Cliffs, Prentice-Hall, c1977 v.4 Data Structuring. p.80-149.

[GOG78c] GOGHEN,J.A. Abstract Errors for Abstract Data Types. In: IFIP TC-2 WORKING CONFERENCE on FORMAL DESCRIPTION of PROGRAMMING CONCEPTS, Saint Andrews, Aug.1-5,1977. Formal Description of Programming Concepts. Amsterdam, North-Holland, c1978.

[GOG79] GOGHEN,J.A. & TARDO,J.J. ,An Introduction to OBJ: A Language for Writing and Testing Algebraic Program Specifications. In: SPECIFICATION of REALIBLE SOFTWARE CONFERENCE, Cambridge, 1979. Proceedings. New York, IEEE, c1979. p.170-89.

[GOG82] GOGHEN,J.A. & MESEGUER,J. Rapid Prototyping in the OBJ Executable Language. Software Engineering Notes, New York, Z(5):75-84, Dec. 1982.

[GOG87] GOGHEN,J.A. Priciples of Parameterized Programming. Stanford University/Center of Study of Language and Information, 1987.

[GUT78a] GUTTAG,J.V.; HOROWITZ,E.; MUSSER,D.R. The Design of Data Types Specifications. In: Current Trends in Programming Methodology Vol.4, ed. YEH,R.T., Prentice Hall, 1978.

[GUT78b] GUTTAG,J.V. & HOROWITZ,E. & MUSSER,D.R. Abstract Data Types and Software Validation. Communication of the ACM, New York, 21(12):1048-1064, Dec. 1978.

[KLA83] KLAREN,A.A. Algebraische Spezifikation : Eine Einfuehrung. Berlin, Springer-Verlag, 1983.

[KRE84] KREOWSKI,H.J. Algebraische Spezifikation. Bremen, Universidade de Bremen, 1984. Notas de Aula.

[MEN88] MENDES, S.B.T. & AGUIAR,T.C. Métodos para Especificação de Sistemas. Curitiba, III EBAI, 1988

Relatórios de Pesquisa

RP-114: "Introdução do Tempo no Ambiente para Desenvolvimento de Bancos de Dados Dedutivos", julho 1989.
N. EDELWEISS

RP-113: "Especificações Formais em OBJ", julho 1989.
N. EDELWEISS & A.L. RIBEIRO

RP-112: "Simulador LAÇO para o Sistema AMPLO", maio 1989.
C. LE FAOU & J. F. SILVA FILHO

RP-111: "Teste de Circuitos Digitais: Uma Experiência em Programação em Lógica", maio 1989.
M. S. LUBASZEWSKI

RP-110: "Linguagem de Especificação Formal Z",maio 1989.
A. AGUSTINI & V. M. RODRIGUES

RP-109: "Ferramentas de Apoio ao Ensino de Autômatos e Máquinas de Turing", abril 1989.
A. M. A. PRICE & F. S. QUADROS

RP-108: "Modelagem da Definição de Dados do Sistema de Formulários Eletrônicos no Modelo de Entidades ("E"), abril 1989.
V. R. FICHMAN & D.D.A. RUIZ

RP-107: "Um Verificador de Tipos Polimórficos para PROLOG", março 1989
V. M. RODRIGUES

RP-106: "Biblioteca de Células do Projeto TRANCA em Regras do 2o PMU", janeiro 1989
F. SOMENZI, A. REIS, F. MORAES, M. LUBASZEWSKI, R. GOMES, R. REIS

RP-105: "Ferramentas para Especificação e Controle de Interfaces com o Usuário", janeiro 1989.
S.D. OLABARRIAGA, B. COPSTEIN & C.M.D.S. FREITAS

RP-104: "Matemática Intervalar: das origens ao estado da arte", janeiro 1989.
D.M. CLAUDIO

RP-103: "EXTRIBO - Um Extrator de Circuitos", janeiro 1989.
M.A. STEMMER & R.A.L. REIS

RP-102: "Pacote Gráfico para Editores do Sistema AMPLO", dezembro 1988.
M.S. PINHO, J.L.D. COMBA & S.D. OLABARRIAGA