



Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Computação e Automação Industrial



Compilador ASN.1 e Codificador/Decodificador para BER

Autora: Maria Inés Valderrama Restovic

Orientador: Prof. Dr. Manuel de Jesus Mendes

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Automação**.

Banca Examinadora

Prof. Dr. Manuel de Jesus Mendes (presidente) — DCA/FEEC/UNICAMP

Prof. Dr. Wanderley Lopes de Souza — DSC/CCT/UFPB

Prof. Dr. Mauricio Ferreira Magalhães — DCA/FEEC/UNICAMP

Campinas – SP
03 de Setembro de 1992

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

R313c	Restovic Valderrama, Maria Inés Compilador ASN.1 e codificador/decodificador para BER / Maria Inés Valderrama Restovic. --Campinas, SP: [s.n.], 1992. Orientador: Manuel de Jesus Mendes. Dissertação de Mestrado - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação. 1. Redes de computadores. 2. Protocolos. 3. Compiladores (Programa de computador). 4. Tipos abstratos de dados (Computação). I. Mendes, Manuel de Jesus. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.
-------	--

Título em Inglês: ASN.1 compiler and encode/decode for BER

Palavras-chave em Inglês: Computers networks, Protocols, Compilers (Computer programs),
Abstract data types (Computer)

Área de concentração: Automação

Titulação: Mestre em Engenharia Elétrica

Banca examinadora: Mauricio Ferreira Magalhães, Wanderley Lopes de Souza

Data da defesa: 03/09/1992

Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Nome do Candidato: Maria Inês Valderrama Restovic

Data da Defesa : 03/09/1992

Título da Tese : "Compilador ASN.1 e Codificador/Decodificador para BER"

Prof. Dr. Manuel de Jesus Mendes : _____

Prof. Dr. Maurício Ferreira Magalhães : _____

Prof. Dr. Wanderley Lopes de Souza : _____

Secretária Betty Ballesteros Bueno : _____

Coordenador PG Prof. Dr. Rui Fragassi Souza : _____

Resumo

Neste trabalho apresenta-se uma ferramenta chamada Compilador ASN.1, cujo objetivo é fornecer uma representação concreta para a sintaxe abstrata ASN.1, de forma que, as especificações das PDU's dos protocolos de aplicação, geralmente escritas em ASN.1, possam ser utilizadas computacionalmente.

Uma das funções prioritárias da camada de apresentação de um protocolo de comunicação é produzir uma codificação dos valores destas PDU's, baseando-se nas regras definidas pela norma BER. Assim, o compilador deve fornecer numa segunda tarefa, as rotinas de codificação e decodificação específicas para cada PDU compilada, utilizando um conjunto de funções que se encontram em duas bibliotecas auxiliares que realizam estas conversões.

Palavras-chave: *redes de computadores, protocolos, compiladores (programa de computador), tipos abstratos de dados (computação).*

Abstract

This work presents a tool called “Compilador ASN.1”, which main objective is to provide a concrete representation for the abstract syntax ASN.1, in order to translate the application protocol PDU’s specification, written in ASN.1, to the C language.

One of the main functions of the presentation layer is produce an encode-decode for the PDU’s data values, based on the BER norm. Therefore, a second compiler task is to provide the specific encode-decode routines for each compiled PDU, using a function set available in two complementary libraries that carry out these conversions.

Keywords: *computers networks, protocols, compiler (computer programs), abstract data types (computer).*

A

Antonio Atta.

Agradecimentos

Ao Brasil por me acolher e ter me oferecido esta oportunidade.

Ao CNPq, pelo apoio financeiro.

Ao meu orientador, Prof. Manuel de Jesus Mendes, pelo tempo dedicado e pelos comentários valiosos.

A minha família, por todo amor e apoio, em especial a meus pais, minha irmã Patricia e ao meu marido Antonio Atta.

Finalmente, às queridas amigas e colegas Cecília e Silvia.

Sumário

LISTA DE FIGURAS.....	XVII
LISTA DE TABELAS	XIX
LISTA DE ABREVIACÕES	XXI
CAPÍTULO 1	23
INTRODUÇÃO	23
1.1 APRESENTAÇÃO.....	23
1.2 MECANISMO DE UTILIZAÇÃO DO PAR ASN.1 – BER.....	27
1.3 MOTIVAÇÃO DA DISSERTAÇÃO.....	29
1.4 OBJETIVOS DO TRABALHO.....	30
CAPÍTULO 2	31
LINGUAGENS.....	31
2.1 SINTAXE ABSTRATA	31
2.1.1 Elementos Básicos	32
2.1.2 Tipos.....	33
2.1.3 Tipos Primitivos	34
2.1.4 Macros	44
2.2 SINTAXE DE TRANSFERÊNCIA	46
2.2.1 Estrutura Geral da Codificação.....	47
CAPÍTULO 3	57
ESPECIFICAÇÃO DO SISTEMA	57
3.1 DETALHES GERAIS	57

3.2 DESCRIÇÃO DOS PROCESSOS DO SISTEMA	60
3.3 ETAPAS NA UTILIZAÇÃO DO COMPILADOR	64
3.4 LIMITAÇÕES DO SISTEMA	67
CAPÍTULO 4	69
TRANSFORMAÇÃO DA LINGUAGEM ASN.1 PARA C	69
4.1 COMPARAÇÃO ENTRE ASN.1 E C.....	69
4.2 A GRAMÁTICA ASN.1	70
4.3 TRANSFORMAÇÕES DE TIPOS ASN.1 PARA C.....	72
4.3.1 Tipos Primitivos	73
4.3.2 Tipos Construtores	76
4.4 MACROS	82
CAPÍTULO 5	91
IMPLEMENTAÇÃO DO COMPILADOR.....	91
5.1 ASPECTOS GERAIS.....	91
5.2 ARQUITETURA DO SISTEMA	93
5.3 PRIMEIRO PASSO DO COMPILADOR (PASSO 1).....	95
5.3.1 Inversão do código de entrada e análise léxica.....	95
5.3.2 Geração do programa codificador	96
5.3.3 Geração de programa decodificador	98
5.4 SEGUNDO PASSO DO COMPILADOR (PASSO 2).....	100
5.4.1 Análise Sintática	100
5.4.2 Análise Semântica.....	101
5.4.3 Geração de Código C.....	101
5.5 BIBLIOTECA DE CODIFICAÇÃO E DECODIFICAÇÃO.....	103
5.5.1 Funções de codificação	103
5.5.2 Funções de decodificação	105
CAPÍTULO 6	109
CONCLUSÕES.....	109
6.1 CONCLUSÕES	109

6.2 TRABALHOS FUTUROS	110
REFERÊNCIAS BIBLIOGRÁFICAS	113
APÊNDICE A	115
REGRAS GRAMATICAIS ASN.1.....	115
APÊNDICE B.....	123
UM EXEMPLO COMPLETO	123

[

Lista de Figuras

Figura 1 – Cada emissor da informação efetua a conversão para o formato usado pelo receptor.....	25
Figura 2 – Cada emissor codifica para ASN.1 e cada receptor decodifica para o seu próprio formato.	27
Figura 3 – Transformação de ASN.1 em uma linguagem de alto nível.	28
Figura 4 – Árvore do OBJECT IDENTIFIER.....	37
Figura 5 – Módulo ASN.1 para MMS	43
Figura 6 – Simulação de um tipo Ponto Flutuante (real) pela Macro Floatpoint.....	45
Figura 7 – Ordenamento de “bits” para BER.....	46
Figura 8 – Campos na codificação de um valor	47
Figura 9 – Octeto de Identificação	47
Figura 10 – Codificação de um número de “tag” maior de trinta.....	49
Figura 11 – Codificação do tamanho de um dado na forma indefinida	49
Figura 12 – Codificação de tamanho na forma definida para valores menores de 128.....	50
Figura 13 – Codificação de tamanho em forma definida para valores maiores de 128.....	50
Figura 14 – Codificação de um valor TRUE para um tipo BOOLEAN	51
Figura 15 – Codificação de um número inteiro.....	51
Figura 16 – Exemplo de codificação BITSTRING ::= ‘101’B.....	52
Figura 17 – Exemplo de codificação de um OCTETSTRING ::= ‘A’.....	52
Figura 18 – Codificação do OBJECT IDENTIFIER	53
Figura 19 – Codificação de um tipo construtor.....	54
Figura 20 – Codificação do tipo a).....	55
Figura 21 – Codificação do tipo b).....	56
Figura 22 – Camadas OSI onde o sistema atua	59
Figura 23 – Processo de Compilação.....	60
Figura 24 – Fases da Compilação.....	61
Figura 25 – Processo de Codificação.....	63
Figura 26 – Processo de Decodificação	64
Figura 27 – Uso do Compilador Estática e Dinamicamente	66
Figura 28 – Notação BNF para o tipo SEQUENCE	71
Figura 29 – Produção que define os valores aceitos por “DEFAULT”	72
Figura 30 – Representação de SEQUENCE em uma “struct”	76
Figura 31 – Regra gramatical que define as variantes para um componente de tipo SEQUENCE ou SET	77
Figura 32 – Exemplo de OPTIONAL em um SET.....	78
Figura 33 – Exemplo de COMPONENTS OF, Dado1 se encontra na Figura 31.....	79
Figura 34 – Código C da lista que representa os tipos OF.....	80
Figura 35 – Parte da especificação da gramática do Passo 2 do compilador para YACC.....	93
Figura 36 – Arquitetura do Compilador ASN.1	94
Figura 37 – Inversão do código ASN.1 a) “top-down” b) “bottom-up”	97
Figura 38 – Esquema de um programa codificador para um módulo ASN.1	98
Figura 39 – Esquema de um programa decodificador para um módulo ASN.1	99
Figura 40 – Arquivo “ASN_prova1tiposc.c” com o código C correspondente ao do módulo ASN_prova1	102

Lista de Tabelas

Tabela I – Lista de palavras reservadas para ASN.132

Tabela II – “Tags” dos tipos *UNIVERSAL*.....40

Tabela III – Codificação das Classes de “tags”48

Lista de Abreviações

ASN.1 – *Abstract Syntax Notation One*

ASCII – *American Standard Code for Information Interchange*

BER – *Basic Encoding Rules*

BNF – *Backus Normal Form*

EBCDIC – *Extended Binary Coded Decimal Interchange Code*

ISO – *International Standard Organization*

LEX – *Lexical Analyzer Generator*

OSI – *Open System Interconnection*

PDU – *Protocol Data Unit*

SISDI-OSI – *Sistema Didático para o Modelo OSI*

YACC – *Yet Another Compiler Compiler*

Capítulo 1

Introdução

Neste capítulo são apresentados os aspectos relevantes da dissertação: o compilador ASN.1 e o Codificador/Decodificador BER, assim como a motivação, os principais objetivos a serem alcançados e a organização do trabalho.

1.1 Apresentação

No modelo de camadas especificado em *ISO Reference Model for Open System Interconnection* da OSI (ISO, 1989), as camadas superiores, aplicação e apresentação, devem prover ao usuário os meios para o intercâmbio de informações entre duas entidades de aplicação remotas. Como consequência disto, as entidades de aplicação envolvidas em prover um serviço de informação distribuída podem estar residentes em diferentes sistemas computacionais (Rose, 1989).

Para que a comunicação seja possível, os usuários de aplicação e apresentação devem estar cientes dos tipos de dados que serão intercambiados e das arquiteturas dos equipamentos

envolvidos. Esta condição pode ser compreendida mais facilmente através do exemplo a seguir (Blake, 1989):

Um usuário A transmite uma PDU (Protocol Data Unit) para o usuário B usando para os dados alfabéticos um código ASCII/IA5 e para os dados numéricos, ponto flutuante com dupla precisão. O usuário B, por sua vez, espera um código EBCDIC para os dados alfabéticos e ponto flutuante com precisão simples para os dados numéricos.

Obviamente, nesse exemplo, o processo de comunicação implica na existência de rotinas de conversão para traduzir os dados nas diferentes representações. Este processo de conversão seria consideravelmente simplificado se ambos os usuários adotassem, por exemplo, o uso do código ASCII para os dados alfabéticos e a precisão simples para os dados numéricos.

Por outro lado, este acordo entre os usuários não garante que a PDU que está sendo transmitida seja utilizada da mesma forma por ambos os usuários. Se, por exemplo, a PDU do usuário A tiver o campo X antecedendo o campo Y, e o usuário B esperar o campo Y antes do campo X, os valores dos campos serão usados de forma errada, produzindo informação inconsistente.

Outra diferença entre arquiteturas que deve ser observada é a posição do *bit* mais significativo em um *byte*. Alguns fabricantes consideram o *bit* mais significativo à esquerda do *byte*, enquanto outros adotam o da direita. A princípio, pode-se acreditar que a solução para este problema é imediata: basta inverter a posição dos *bits* com um simples programa de conversão. Isto na prática, porém, não é tão simples. A inversão de *bits* em campos de tipo *Boolean* não trará maiores problemas; em campos numéricos em notação científica, contudo, é necessário que o programa conversor conheça o tipo do campo para realizar a inversão adequadamente. Em resumo, descrever a ordem dos campos de dados, dos bits dentro dos

campos, assim como a estrutura e a sintaxe da PDU, é fundamental para a interpretação da informação que estes dados carregam.

Uma vez que as PDU's de alguns sistemas podem ser extensas, com tipos de dados complexos, muitas rotinas de conversão são utilizadas. Se considerarmos uma rede de N tipos diferentes de computadores, seriam necessárias de $N * (N - 1)$ conversões, como esquematizado na Figura 1 (Tannenbaum, 1988).

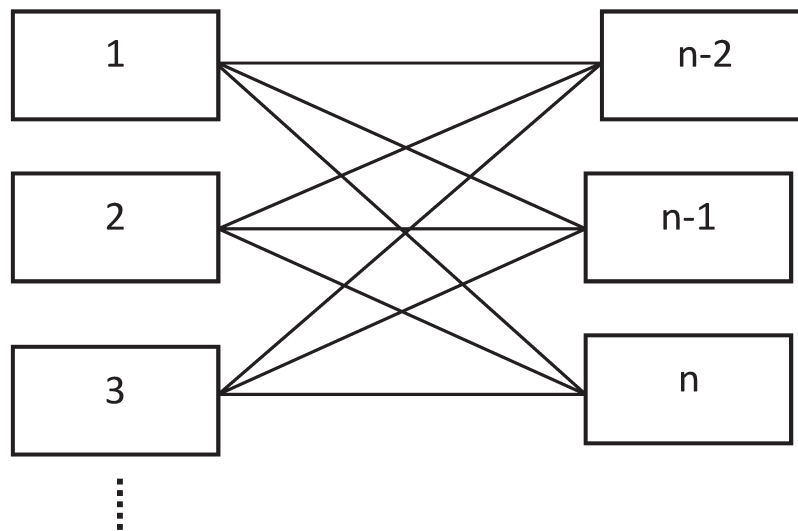


Figura 1 – Cada emissor da informação efetua a conversão para o formato usado pelo receptor.

Para reduzir o grande número de conversões de dados, garantindo sua integridade e semântica, é necessário identificar os problemas pertinentes ao processo: como definir estruturas de dados sem ambigüidades? e como transmitir o conteúdo de uma estrutura de dados sem perda da informação?

A solução para estas questões de representação e transmissão de estruturas de dados consiste em especificar uma forma flexível de descrever os dados, de modo que ela possa ser utilizada em uma ampla gama de aplicações.

O modelo OSI introduz dois conceitos com o objetivo de tornar as camadas de aplicação e apresentação independentes das características físicas dos sistemas onde elas são implementadas: Sintaxe Abstrata e Sintaxe de Transferência. Juntas, essas sintaxes formam o que se denomina o contexto da camada de apresentação, que deve ser negociado sempre que uma conexão é estabelecida (Halshall, 1988).

Uma sintaxe abstrata descreve os tipos de dados de uma maneira independente, ou seja, uma linguagem abstrata não contém restrições dependentes da máquina. Uma sintaxe de transferência, por sua vez, define um conjunto de regras para representação física dos valores dos tipos de dados especificados pela sintaxe abstrata sem ambigüidades.

Desta forma, foi definida a linguagem *Abstract Syntax Notation One* (ASN.1) no documento ISO8824 (ISO, 1987a) como uma linguagem formal de especificação de tipos de dados e definição dos valores para esses tipos. Para essa linguagem abstrata, foi estabelecida a sintaxe de transferência *Basic Encoding Rules* (BER), descrita no documento ISO 8825 (ISO, 1987b), que define as regras de codificação para os valores dos tipos de dados descritos em ASN.1.

A partir destas sintaxes é possível reduzir e padronizar as conversões de códigos da rede. Para o exemplo da Figura 1, em uma rede com N computadores diferentes, ter-se-ão $2 \cdot N$ conversões conforme mostra a Figura 2 (Tannenbaum, 1988).

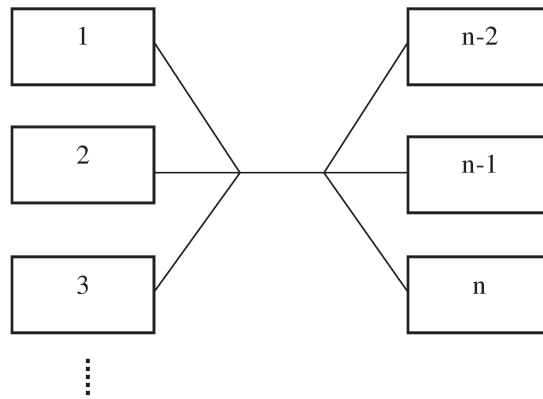


Figura 2 – Cada emissor codifica para ASN.1 e cada receptor decodifica para o seu próprio formato.

A potencialidade do ASN.1 para descrever os dados baseia-se também, no fato de a sintaxe de transferência permitir uma representação única, direta e independente dos dados.

1.2 Mecanismo de utilização do par ASN.1 – BER

A idéia básica consiste em definir todas as estruturas de dados necessárias para cada aplicação em ASN.1 e reuni-las em um único módulo. Quando a aplicação desejar transmitir uma PDU, ela passa a estrutura de dados à camada de apresentação através do seu nome em ASN.1. Usando a definição ASN.1 como guia, a camada de apresentação reconhece o tipo e a forma dos campos da estrutura e, a partir das regras contidas na norma BER, identifica como codificar os seus valores.

No outro lado da comunicação, ao receber os dados, a camada de apresentação examina a sua identificação e obtém a informação sobre o tipo e o tamanho do dado que está recebendo. Esse processo repete-se até que toda a PDU enviada tenha sido analisada. Com esta informação, a camada de apresentação pode efetuar qualquer conversão necessária, do formato externo usado na comunicação, para o formato interno usado pelo computador receptor.

Conceitualmente, estas notações abstratas e de transferência representam uma solução factível e eficiente para os problemas de conversão e transmissão de dados. A ASN.1, apesar de ser uma linguagem de alto nível, como Pascal e C, e ser precisamente abstrata, não passa de uma norma. Portanto, não possui representação computacional para as definições dos tipos de dados que as compõem. Dessa forma, é imprescindível transformar ASN.1 em uma linguagem mais “fortemente tipada”, para que ela possa ser utilizada na construção de ferramentas capazes de codificar e decodificar os valores dos tipos de dados expressos em ASN.1, como fica esquematizado na Figura 3 (Bochman, et al., 1991).

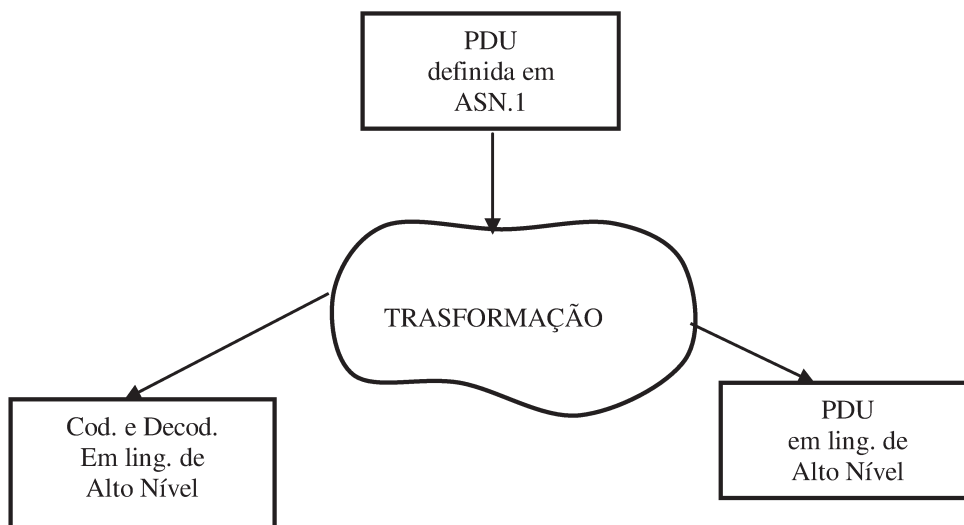


Figura 3 – Transformação de ASN.1 em uma linguagem de alto nível.

Considerando as características de complexidade e tamanho já mencionadas das PDUs e a existência de muitos protocolos de aplicação interagindo em um ambiente OSI, não é

recomendável que se realizem as transformações das PDUs ASN.1 em uma outra linguagem, de forma independente, por cada usuário de aplicação. Caso contrário, esta transformação poderia gerar diferentes interpretações de uma mesma estrutura, o que afetaria a sua padronização e portabilidade, já que, os dados intercambiados pelos protocolos baseiam-se nessa estrutura e as rotinas de conversão de codificação e decodificação devem utilizar os mesmos tipos de dados.

Uma ferramenta automática que realize esta transformação e que gere uma codificação e uma decodificação dos valores das PDUs é necessária em um ambiente aberto de comunicação, como o estabelecido pelo modelo OSI.

Como será visto nos próximos capítulos, o formalismo com o qual ASN.1 foi definido possibilita a construção desta ferramenta, mediante a utilização de técnicas tradicionais de implementação de compiladores.

1.3 Motivação da Dissertação

A partir da caracterização do contexto em que este trabalho está inserido, pode-se enumerar alguns pontos que motivaram o seu desenvolvimento:

ASN.1 vem sendo considerada a melhor solução para a transmissão dos dados nos protocolos OSI;

É necessário obter-se uma ferramenta que opere a transformação direta entre ASN.1 e uma linguagem de alto nível;

O volume e a complexidade das estruturas de dados intercambiadas no processo de comunicação, torna necessário a automação das funções de codificação e decodificação dos dados;

Dentro do contexto do projeto SISDI-OSI, desenvolvido por esta faculdade, existe a carência de um compilador ASN.1.

1.4 Objetivos do Trabalho

O principal objetivo desta dissertação é o estudo da sintaxe abstrata ASN.1 e sua correspondente sintaxe de transferência BER visando obter como maior resultado uma ferramenta que seja capaz de:

Ler uma especificação de um módulo ASN.1 de um certo protocolo de aplicação, compilá-lo e produzir um código equivalente a esta especificação na linguagem C, para que possa ser utilizado pelos usuários da aplicação e pelo próprio sistema de codificação e decodificação das PDUs;

Produzir um codificador e um decodificador específicos para este módulo, que possam ser utilizados pela camada de apresentação, para realizar as funções de produção de um conjunto de octetos binários, no caso da codificação, e, no caso inverso, reconstruir os dados, baseando-se especificamente nas regras BER definidas como a sintaxe de transferência.

O trabalho está dividido em seis capítulos. No próximo capítulo é apresentado um resumo dos aspectos mais relevantes de ambas as sintaxes estudadas, ASN.1 e BER. No capítulo 3 é feita uma descrição conceitual do sistema que será implementado. No capítulo 4 são avaliadas as diferenças entre ASN.1 e a linguagem C, os problemas inerentes ao processo de transformação e os tipos de dados envolvidos. No capítulo 5 é descrito o ambiente resultante do trabalho, assim como, a própria implementação. Finalmente, no capítulo seis, são apresentadas as conclusões as futuras extensões sugeridas a este trabalho.

Capítulo 2

Linguagens

Este capítulo contém uma revisão dos aspectos mais relevantes da linguagem ASN.1 e da norma de codificação BER.

2.1 Sintaxe Abstrata

A sintaxe abstrata ASN.1 (*Abstract Syntax Notation One*), é definida no documento ISO 8824 e complementada no documento *Draft Adendo 8824/DAD 1* (ISO, 1987b). ASN.1 pode ser definida como a primeira padronização estabelecida para representar dados em forma abstrata e foi inicialmente concebida sem uma representação concreta. Trata-se de uma linguagem formal, o que significa que está definida através de uma gramática com a qual se pode definir todos os componentes da linguagem, e que possui uma notação semelhante a BNF (*Backus Normal Form*). O princípio da linguagem é o mesmo adotado pela maioria das linguagens de alto nível, como, por exemplo, Pascal e C, em que, para cada variável declarada é definido um tipo de dado.

Quando um valor é associado a uma variável, sua sintaxe é regida pelo do tipo definido (como podemos ver ASN.1 serve para definir tipos e valores para esses tipos). Um terceiro e

importante elemento que esta notação provê é a Macro que consiste numa poderosa ferramenta capaz de estender a gramática da linguagem.

2.1.1 ELEMENTOS BÁSICOS

ASN.1 é uma linguagem “*Top-Down*”, isto é, a descrição de suas estruturas é do maior nível de abstração ao menor e a notação consiste em um conjunto de “*tokens*” (unidade mínima da linguagem) dos quais se distinguem quatro classes:

- ✓ **Palavras** que são formadas de letras maiúsculas ou minúsculas, dígitos, hífen e devem sempre começar com uma letra. Podemos distinguir três classes de palavras:
 - Tipos que sempre começam com letra maiúscula;
 - Identificadores que começam com letra minúscula;
 - Palavras reservadas (Tabela I).

Tabela I – Lista de palavras reservadas para ASN.1

BOOLEAN	OPTIONAL
INTEGER	DEFAULT
BITSTRING	COMPONENTS
OCTETSTRING	UNIVERSAL
NULL	APPLICATION
SEQUENCE	PRIVATE
SET	TRUE
OF	FALSE
CHOICE	BEGIN
IMPLICIT	END
ANY	DEFINITIONS
EXTERNAL	EXPLICIT
OBJECT	ENUMERATED
IDENTIFIER	EXPORTS
DESCRIPTOR	REAL

Não estão consideradas no conjunto representado pela Tabela I palavras reservadas da notação Macro, explicada adiante neste capítulo.

✓ **Números** que consistem em um ou mais dígitos

✓ **Strings** que são de três formas:

“ ” cadeia de caracteres;

‘0–9A–F’H cadeia de hexadecimais;

‘0–1’B cadeia de binários.

✓ **Pontuação** que consiste dos seguintes caracteres:

: = , { } < > () [] - ‘ “ |

2.1.2 TIPOS

A linguagem possui um conjunto de tipos primitivos e um mecanismo de formação de tipos mais complexos utilizando os tipos primitivos, chamados de tipos construtores.

A sintaxe utilizada para definir ambos os tipos é da forma BNF, onde adotamos as seguintes convenções: o símbolo $::=$ é uma atribuição; todos os nomes que começam com letra maiúscula correspondem a produções não terminais; os nomes escritos somente com letras maiúsculas são símbolos terminais da gramática, assim como, os caracteres entre aspas (“ ”). A forma de interpretar uma produção é segui-la seqüencialmente, levando em conta que uma regra gramatical pode ter uma ou mais opções, situação indicada com o caracter “|”. As regras apresentadas a seguir correspondem a declarações de um tipo e seu valor, seja este primitivo ou construtor.

Atribuição de tipo $::= \text{Nome do tipo} \text{ “}::\text{” Tipo}$

A sintaxe para definir o valor de um tipo é da forma:

Atribuição de valor $::= \text{Nome do valor Nome do tipo} \text{ “}::\text{” Valor}$

2.1.3 TIPOS PRIMITIVOS

Os tipos primitivos são os mais simples e atômicos. Eles serão descritos através de exemplos para cada tipo e correspondente atribuição de valor.

BOOLEAN é um tipo que só tem dois valores TRUE e FALSE. Sua gramática é da forma:

$\text{BooleanType} ::= \text{BOOLEAN}$

$\text{Booleanvalue} ::= \text{TRUE} \mid \text{FALSE}$

Exemplo:

$\text{EstadoCivil} ::= \text{BOOLEAN}$

$\text{casado EstadoCivil} ::= \text{TRUE}$

INTEGER é um tipo que tem por valores números cardinais. Uma extensão do tipo INTEGER é a que permite simular o tipo enumerado, o qual foi definitivamente incluído na sintaxe no adendo da norma. Um tipo enumerado limita o conjunto de valores que o tipo pode assumir a um conjunto finito e definido na declaração do tipo. As regras gramaticais que definem o tipo são as seguintes:

$\text{IntegerType} ::= \text{INTEGER} \mid \text{INTEGER} \text{ “} \{ \text{ “ Namednumberlist “} \text{”}$

$\text{Namednumberlist} ::= \text{Namednumber} \mid \text{Namednumberlist “}, \text{” Namednumber}$

$\text{Namednumber} ::= \text{IDENTIFIER} \text{ “} (\text{ “ Signednumber “}) \text{”}$

$\text{Signednumber} ::= \text{NUMBER} \mid \text{ “} - \text{” NUMBER}$

Exemplos:

Diames ::= INTEGER

primeiro Diames ::= 1

DiadaSemana ::= INTEGER { segunda (0), terça (1), quarta (2) }

DiadaSemana representa um tipo enumerado que pode assumir um dos três valores definidos para ele 0, 1 ou 2. O tipo ENUMERATED definido por adendo é exatamente igual:

DiadaSemana ::= ENUMERATED { segunda (0), terça (1), quarta (2) }

REAL é o tipo que pode assumir valores números reais. Este número é composto de três inteiros: mantissa, base, expoente. A mantissa e o expoente podem ser qualquer número cardinal mas a base só pode ser 2 ou 10, com a seguinte relação entre os três números.

$$\text{mantissa} * \text{base}^{\text{expoente}}$$

Exemplo:

Saldo ::= REAL

saldoconta Saldo ::= { 1052050, 10, -2}

BITSTRING é o tipo constituído por 0 ou mais “bits” como valores.

Exemplo:

AttributeGroups ::= BITSTRING

group AttributeGroups ::= '101'B

OCTETSTRING é o tipo constituído por valores 0 ou mais octetos (valores múltiplos de oito).

Exemplo:

Nome ::= OCTETSTRING

nomeusuario NOME ::= “INES”

ANY é a união de todos os tipos definidos em ASN.1. É normalmente usado quando ainda não se tem conhecimento do tipo em alguma especificação (protocolo). Seu uso será visto em maior detalhe nos capítulos seguintes.

Exemplo:

Anytype ::= ANY

– NULL é o tipo que não possui nenhum valor.

Nulltype ::= NULL

EXTERNAL é o tipo de dado utilizado para declarar os tipos definidos fora do módulo ASN.1 corrente (sejam eles tipos ASN.1 ou não). Sua gramática é a seguinte:

Externaltype ::= EXTERNAL

Tipos OBJECT

Existem dois tipos OBJECT em ASN.1 e a utilização desses tipos é importante porque permite a definição formal e informal dos módulos ou dos tipos utilizados nas especificações. São eles:

OBJECT DESCRIPTOR é um texto que referencia algum objeto só como informação adicional a título de esclarecimento – como um comentário – e não tem uma sintaxe formal.

Exemplo:

Ber ::= OBJECT DESCRIPTOR

ber Ber ::= “Basic Encoding Rules”

OBJECT IDENTIFIER é um tipo de dado que define totalmente e sem ambigüidade um objeto: é o nome com o qual se identifica um objeto, por exemplo uma PDU ou módulo ASN.1. Um OBJECT IDENTIFIER é uma seqüência de números não negativos que é obtida pelo caminhamento de uma árvore: cada nó da árvore corresponde a um número de um documento ISO, CCITT ou a ambos e é um componente do OBJECT IDENTIFIER.

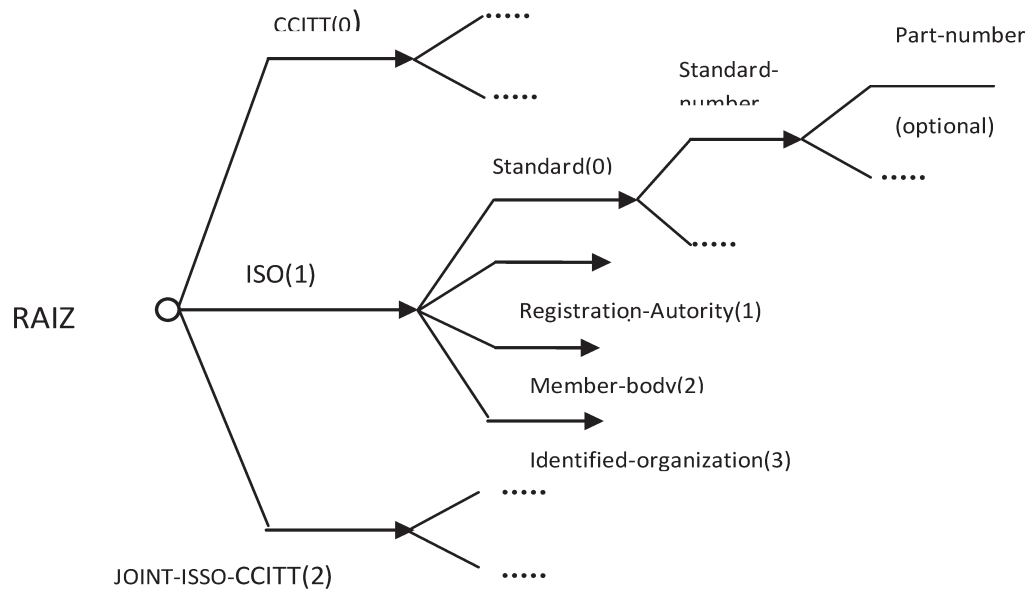


Figura 4 – Árvore do OBJECT IDENTIFIER

A Figura 4 (White, 1989) apresenta parte da árvore para definir os identificadores que compõem um OBJECT IDENTIFIER. Da raiz da árvore dependem as autoridades administrativas (ISO, CCITT, JOIN-ISO-CCITT), que são as responsáveis pela ramificação de cada nó. Seguindo esta árvore, assegura-se que para cada objeto existe um único nome.

Exemplo: O OBJECT IDENTIFIER que define o protocolo FTAM, é obtido pelo caminhamento da árvore: (raiz, ISO, Standard, FTAM), que corresponde à seqüência numérica (1, 0, 8571). Portanto, o tipo Ftam têm o seguinte valor associado:

Ftam OBJECT IDENTIFIER ::= { 1 0 8571 }

Existem várias formas previstas para representar um OBJECT IDENTIFIER com o objetivo de maior clareza e legibilidade, podendo-se atribuir um identificador a cada número que compõe a seqüência.

Exemplo:

Ftam OBJECT IDENTIFIER ::= { iso standard 8571 }

Tipos Construtores

São os tipos formados a partir de tipos primitivos. Um tipo construtor pode utilizar tipos simples e também tipos construtores para obter estruturas de tipos de complexidade e profundidade arbitrárias.

- ✓ SEQUENCE é um tipo que corresponde a uma lista ordenada de 0 ou mais elementos de qualquer tipo ASN.1

Exemplo:

Lista ::= SEQUENCE { nome OCTETSTRING,
valor INTEGER }

- ✓ SEQUENCE OF é uma lista ordenada de 0 ou mais elementos de tipos ASN.1 iguais.

Exemplo:

Nomes ::= SEQUENCE OF OCTETSTRING

- ✓ SET é uma lista não ordenada de 0 ou vários elementos de qualquer tipo ASN.1

Exemplo:

Lista ::= SET { nome OCTETSTRING,
valor INTEGER }

- ✓ SET OF é uma lista não ordenada de 0 ou mais elementos todos do mesmo tipo ASN.1

Exemplo:

Keywords ::= SET OF OCTETSTRING

- ✓ CHOICE é o tipo definido como a união de um ou mais tipos ASN.1: um valor desse tipo toma o valor de um e só um dos tipos definidos nele.

Exemplo:

```
FileIdentifier ::= CHOICE {  
  
    relativename OCTETSTRING,  
  
    absolutename OCTETSTRING }
```

Tipos TAGGED

A identificação de cada instância de um tipo ASN.1 deve ser única. Este problema apresenta-se, por exemplo, em tipos como SET que é uma lista não ordenada cujos valores só são distinguidos pelo tipo que possuem. Caso eles sejam de tipo similar distingui-los não é fácil. Os tipos TAGGED foram criados para eliminar essas ambigüidades na transmissão dos valores.

Os tipos TAGGED são divididos em quatro categorias:

- ✓ UNIVERSAL, para distinguir os tipos da norma ASN.1 e para todos os tipos primitivos e construídos (ver código na Tabela II);
- ✓ CONTEX-SPECIFIC, estão relacionados com o contexto específico em que são usados para eliminar ambigüidades dos tipos construtores (em um SET por exemplo); não têm validade fora do construtor em que foram definidos;
- ✓ APPLICATION, utilizado para identificar de forma única, os tipos que se encontram dentro de um módulo ASN.1;
- ✓ PRIVATE, definido pelo usuário mediante um acordo para identificar uma aplicação privada.

Exemplo:

```

Cliente ::= SET { nome [0] IMPLICIT OCTETSTRING,
                  numero [1] IMPLICIT INTEGER,
                  endereço [2] IMPLICIT OCTETSTRING OPCIONAL }

```

Quando se utiliza uma “tag” para um tipo que, por definição, já possui a “tag” universal, este ficará com duas “tags” para evitar uma codificação adicional no momento da transmissão do dado. A norma ASN.1 cria para tal as “tags” implícitas: o do tipo dado é mantido, mas sua “tag” é trocada pela especificada na sua definição, para maior eficiência no momento da codificação.

Tabela II — “Tags” dos tipos *UNIVERSAL*

Universal Tag	Tipos ASN.1
1	BOOLEAN
2	INTEGER
3	BIT STRING
4	OCTET STRING
5	NULL
6	OBJECT IDENTIFIER
7	OBJECT DESCRIPTOR
8	EXTERNAL
9	REAL
10	ENUMERATED
12-15	Reservado
16	SEQUENCE SEQUENCE OF
17	SET SET OF
18-28	String
29- ...	Reservado

Módulos

Todas as declarações ASN.1 que tenham uma relação comum, isto é, que consistam no conjunto completo de PDU's na especificação de um protocolo, formam um módulo. Cada protocolo possui um módulo independente para suas declarações de tipos, sendo possível através de mecanismos dentro da definição de um módulo, a comunicação entre os diferentes módulos. A sintaxe de um módulo ASN.1, é descrita em forma simplificada como:

```
"modulo" DEFINITIONS "tags" ::=
BEGIN
    "enlace"
    "declarações"
END
```

Onde "modulo" é o nome que identifica o módulo: pode ser um nome que deve começar com uma letra maiúscula ou opcionalmente uma identificação mais formal, como um OBJECT IDENTIFIER.

"tags": referencia todas as "tags" do módulo. Pode ser EXPLICIT ou IMPLICIT; se é EXPLICIT ou não aparece o campo "tags", consideram-se as "tags default" segundo a norma para todos os tipos. Se é IMPLICIT todos os tipos consideram a "tag" implícita.

"enlace" define a relação deste módulo com outros módulos, se é que existe: se de um módulo, são utilizados tipos definidos nele em outros módulos, estes tipos devem ser declarados EXPORT.

```
Exemplo:      modulo DEFINITIONS ::=
                BEGIN
                EXPORT tipo1,tipo2
                tipo1
                tipo2
                END
```

No caso contrário, se um módulo precisa de tipos definidos em outros módulos, deve importar os tipos através do nome do tipo e do nome de módulo onde este se encontra. Como mostra o seguinte exemplo:

Exemplo:

```
modulo DEFINITIONS ::=
BEGIN
    IMPORTS tipo1 FROM modulo1
    var ::= tipo1
END
```

Finalmente, “declarações” são todas as definições de tipos, valores e macros que compõem a especificação deste módulo. Na Figura 5 pode-se ver parte do módulo que descreve o protocolo MMS (Mendes, 1989).

```

ISO-9506-2-MMS DEFINITIONS ::=
BEGIN
  MMSpdu ::= CHOICE
  <
    confirmed-RequestPDU      [0] IMPLICIT Confirmed-RequestPDU,
    confirmed-ResponsePDU     [1] IMPLICIT Confirmed-ResponsePDU,
    confirmed-Error           [2] IMPLICIT Confirmed-ErrorPDU,
    unconfirmed-RequestPDU    [3] IMPLICIT Unconfirmed-requestPDU,
    rejectPDU                 [4] IMPLICIT RejectPDU,
    cancel-RequestPDU         [5] IMPLICIT Cancel-RequestPDU,
    cancel-ResponsePDU        [6] IMPLICIT Cancel-ResponsePDU,
    cancel-ErrorPDU           [7] IMPLICIT Cancel-ErrorPDU,
    initiate-RequestPDU       [8] IMPLICIT Initiate-RequestPDU,
    initiate-ResponsePDU      [9] IMPLICIT Initiate-ResponsePDU,
    initiate-ErrorPDU         [10] IMPLICIT Initiate-ErrorPDU,
    conclude-RequestPDU       [11] IMPLICIT Conclude-RequestPDU,
    conclude-ResponsePDU      [12] IMPLICIT Conclude-ResponsePDU,
    conclude-ErrorPDU         [13] IMPLICIT Conclude-ErrorPDU,
  >

  Confirmed-RequestPDU ::= SEQUENCE
  <
    Invoke I dUnsigneda2,
    ListofModifier SEQUENCE OF Modifier OPTIONAL,
    Confirmed-service-Request
  >

  Confirmed-ResponsePDU      ::= ...
  Confirmed-ErrorPDU         ::=
  UnconfirmedRequestPDU      ::=
  RejectPDU                  ::=
  "
  "
  "

  Conclude-ErrorPDU          ::= ...
END

```

Figura 5 – Módulo ASN.1 para MMS

2.1.4 MACROS

Uma ferramenta poderosa do ASN.1 são as Macros. Criada com o propósito de estender a linguagem original, uma macro pode ser construída para satisfazer um requerimento que os tipos e mecanismos definidos na norma original não conseguem atingir. Formalmente Macro é um mecanismo para construir e referenciar novos tipos em ASN.1, especificando uma nova notação através de um conjunto de produções. A definição de macro é a que se segue:

```
“MACRO” MACRO
BEGIN
TYPE NOTATION      ::= “sintaxe de tipo”
VALUE NOTATION     ::= “sintaxe de valor”
“sintaxe complementar”
END
```

Na sintaxe de declaração de uma macro “MACRO” é o nome único, composto unicamente de letras maiúsculas, que identifica a macro e através do qual se referencia o novo tipo criado.

TYPE NOTATION: define as regras gramaticais do novo tipo, associando um ou vários tipos ASN.1 já existentes ao novo tipo, ou criando uma nova sintaxe para o tipo através de uma “sintaxe de tipo”.

VALUE NOTATION: define a sintaxe do valor do novo tipo associando ao tipo valores já definidos em ASN.1, como por exemplo, valores de tipo INTEGER, ou definindo uma nova sintaxe para esse valor através da “sintaxe de valor”.

Finalmente, a “sintaxe complementar” define as regras gramaticais adicionais usadas por TYPE e VALUE NOTATION para as novas sintaxes: essas gramáticas são definidas mediante produções de forma similar à notação BNF.

A utilização de um novo tipo criado por uma macro é similar a qualquer invocação de tipo em ASN.1: é associando o nome da macro para uma referência de tipo e o comportamento

deste é definido pela nova sintaxe. Na Figura 6 encontra-se um exemplo de utilização deste mecanismo.

```
FloatPoint MACRO ::=
  BEGIN
    TYPE NOTATION ::= < type ::= OCTETSTRING
    VALUE NOTATION ::= value (VALUE Floatvalue)
    Floatvalue ::= Mantissa Exponent
    Mantissa ::= Sign number". "number
    Exponent ::= Precision Sign number
                | empty
    Precision ::= "e"
                | "E"
    Sign ::= "f"
            | "-"
            | empty
  END
```

Figura 6 – Simulação de um tipo Ponto Flutuante (real) pela Macro Floatpoint

A macro da Figura 6 corresponde à macro utilizada antigamente pelo protocolo MMS, uma vez que o tipo REAL não estava definido como um tipo básico da norma ASN.1 (Mendes, 1989), vindo a ser introduzido no adendo posterior. Neste exemplo pode-se notar que é atribuído um tipo (OCTETSTRING) como tipo da macro na linha: "TYPE NOTATION ::= < type ::= OCTETSRING", o que significa que um número "real" será uma cadeia de caracteres cujo formato é especificado pela VALUE NOTATION por meio da sintaxe definida pela produção "*Floatvalue*" e as demais produções dependentes de "*Floatvalue*". Portanto, temos que qualquer cadeia de caracteres composta de números e as letras: "e,E,f" e dos caracteres " , "-" que cumpram as regras definidas será considerada um tipo REAL. Por exemplo: a cadeia "104.99E2" é considerada um número REAL, embora a cadeia "104.e2" não satisfaça as regras definidas pela macro.

2.2 Sintaxe de Transferência

Para que os valores dos tipos de dados definidos em forma abstrata possam ser transmitidos pela rede, precisam de uma forma concreta de representação. Com esse objetivo, a ISO descreve no documento 8825 a sintaxe de transferência BER (*BASIC ENCODING RULES*) (ISO, 1987b) correspondente a ASN.1. BER é composta por um conjunto de regras de codificação, para cada tipo de dados ASN.1, com as quais cada valor de um tipo será transformado em uma sequência de octetos binários, prontos para serem transmitidos sem ambigüidades em seus valores e carregando a informação do tipo ao qual pertencem.

As regras de codificação são independentes das representações internas dos dados nos diferentes computadores que fazem parte da rede, estabelecendo de forma fixa a posição do “bit” mais significativo (o mais à esquerda) e o menos significativo (o mais à direita), em um octeto. A Figura 7 mostra a convenção BER para o “bit” mais significativo.

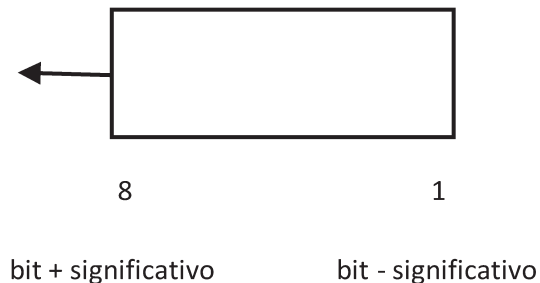


Figura 7 — Ordenamento de “bits” para BER

É importante considerar também que todos os números inteiros são codificados em complemento de dois e os caracteres segundo o código ASCII.

2.2.1 ESTRUTURA GERAL DA CODIFICAÇÃO

Os valores ASN.1 são codificados em três campos. Essa estrutura geral é válida para todos os tipos, sejam estes primitivos ou construídos, sendo a base da codificação dos dados por meio de BER e sempre através dos tipos primitivos.

O primeiro campo contém a codificação da identificação de tipo, o segundo indica o tamanho do valor e o último é destinado para o valor, como mostra a Figura 8. Numa codificação os octetos devem sempre seguir essa ordem.



Figura 8 – Campos na codificação de um valor

Identificação

Os octetos de identificação codificam a informação sobre um tipo por meio da “tag” desse tipo, considerando a classe da “tag” e seu número, podendo este ser codificado em mais de um octeto dependendo de seu tamanho. Outra informação codificada neste campo é se um tipo é primitivo ou é um construtor.

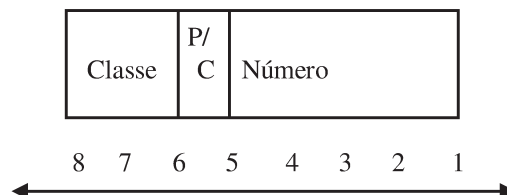


Figura 9 – Octeto de Identificação

A Figura 9 descreve a distribuição dos “bits” no octeto de identificação, onde a codificação da informação referente a um tipo é feita da seguinte forma:

- ✓ Classe de uma “tag” é codificada nos dois primeiros “bits” (8 e 7) segundo os códigos que aparecem na Tabela III.

Tabela III – Codificação das Classes de “tags”

Classe	bits 8-7
UNIVERSAL	0 0
APPLICATION	0 1
CONTEX SPECIF	1 0
PRIVATE	2 1

- ✓ A classe de um tipo é codificada no “bit” 6, sendo seu valor 0 se o tipo é da forma primitiva e 1 no caso contrário.
- ✓ O Número da “tag” será codificado como um binário sem sinal e de duas formas diferentes segundo o tamanho:
 - a) Para números de “tag” menor ou igual a trinta, os cinco “bits” restantes no octeto bastam para codificar o número; logo, a identificação de um valor seria em um octeto só.
 - b) Para números de “tag” maiores de trinta os cinco últimos “bits” tomarão o valor 11111 binário e o número da “tag” será codificado no menor número de octetos possível, sendo que o “bit” 8 de cada octeto tomará o valor 1, à exceção do último octeto, no qual o valor do “bit” será 0, para indicar que é o último octeto componente do número da “tag”; o número será obtido concatenando os 7 “bits” restantes do octeto. Esta situação pode ser vista na Figura 10.

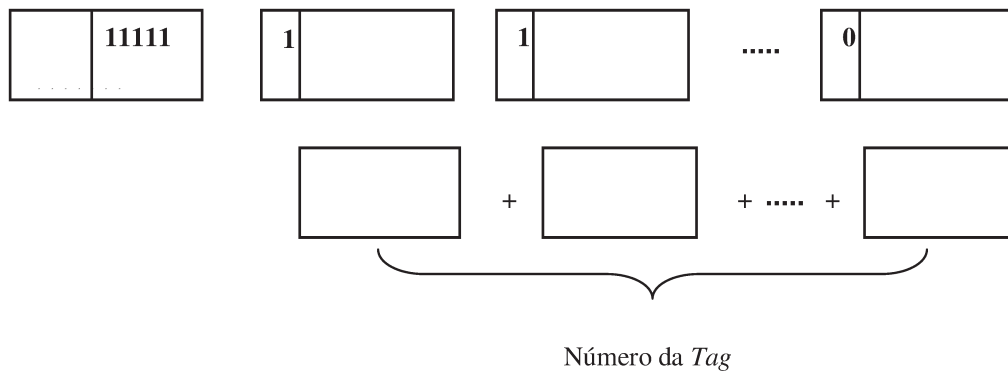


Figura 10 – Codificação de um número de “tag” maior de trinta

Tamanho

O campo tamanho corresponde à codificação do número de octetos que são necessários para codificar um valor, isto é, quantos octetos serão utilizados. Uma dificuldade da codificação deste campo é calcular precisamente o tamanho de um valor e enviar essa informação antes de codificar o valor. A norma BER apresenta duas opções para sua solução:

- ✓ **Forma Indefinida** – Na forma definida como seu nome indica, o tamanho de um valor não é conhecido, e portanto marca-se, por meio de octetos especiais, o início e o fim da codificação do valor. Desta forma, o octeto de tamanho é um só, onde o “bit” 8 toma o valor 1 e os 7 “bits” restantes o valor binário 0. O fim da codificação do valor é marcado mediante dois octetos de valor 0, como mostra a Figura 11.



Figura 11 – Codificação do tamanho de um dado na forma indefinida

- ✓ **Forma Definida** – Para calcular o número de octetos necessários para codificar um valor, o primeiro passo deve ser ler o valor e calcular seu tamanho e num segundo passo, codificá-lo.

Se o número de octetos necessários para codificar o valor é menor que 128, um só octeto de tamanho é suficiente, indicando-se com o número binário 0 no bit 8 esta condição, e nos 7 “bits” restantes o número de octetos como aparece na Figura 12.

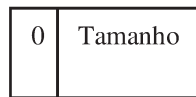


Figura 12 – Codificação de tamanho na forma definida para valores menores de 128

Na Figura 13, mostra-se a codificação para tamanhos maiores que 128. O primeiro octeto do campo tamanho indicará o número de octetos necessários para codificar esse número e seu “bit” mais significativo terá o valor 1. O número será obtido pela concatenação dos octetos seguintes.

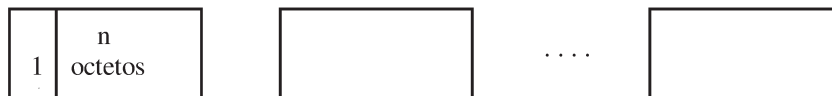


Figura 13 – Codificação de tamanho em forma definida para valores maiores de 128

Valor

A codificação do campo valor de um dado é feita em 1 ou mais octetos, e os tipos simples provêm a codificação fundamental usada por BER, uma vez que a codificação dos valores dos tipos construídos é feita com base nos seus componentes. A seguir será vista a codificação de alguns dos tipos simples mais utilizados:

- ✓ **BOOLEAN** – Um valor do tipo BOOLEAN é sempre codificado na forma primitiva: se esse valor é FALSE codifica-se o valor 0, caso contrário, qualquer valor positivo maior que 0 indica a condição TRUE. A Figura 14 é um exemplo da codificação de um valor TRUE.

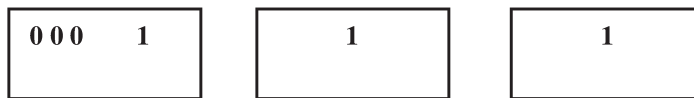


Figura 14 –Codificação de um valor TRUE para um tipo BOOLEAN

- ✓ **INTEGER** – O valores inteiros são codificados em complemento de dois e sempre em forma primitiva. A Figura 15 ilustra um exemplo de codificação de um número inteiro de valor 100.

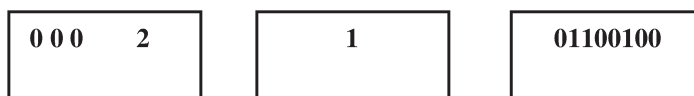


Figura 15 – Codificação de um número inteiro

- ✓ **BITSTRING** – O primeiro octeto do campo valor numa codificação de um BITSTRING indica o número de “bits” que sobrarão livres na codificação do valor, que é feita nos octetos seguintes. Um exemplo de BITSTRING, é a Figura 16.

Quando um valor BITSTRING apresenta um tamanho extenso pode ser mais prático transmiti-lo por partes: para isso, o valor pode ser codificado na forma construída.



Figura 16 –Exemplo de codificação BITSTRING ::= ‘101’B.

- ✓ **OCTETSTRING** – Um valor correspondente ao tipo OCTETSTRING pode ser codificado em um ou mais octetos de conteúdo. Por ser um valor binário múltiplo de oito, nunca se tem “bits” livres no octeto e os caracteres são codificados nos valores correspondentes ASCII. Um exemplo de codificação é mostrado na Figura 17.

Um valor muito comprido pode também ser codificado em forma construída.

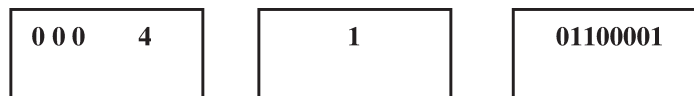


Figura 17 – Exemplo de codificação de um OCTETSTRING ::= ‘A’.

- ✓ **OBJECT IDENTIFIER** – Um OBJECT IDENTIFIER é composto de um conjunto de identificadores numéricos inteiros positivos e sem sinal, que são codificados como inteiros, da seguinte maneira:
- Os dois primeiros identificadores são convertidos em um só número pela fórmula $(X * 40) + Y$ sendo X o primeiro e Y o segundo identificador;
 - Os demais identificadores são codificados por separado formando uma seqüência de octetos onde: se um número precisa de mais de um octeto o “*bit*” 8 de todos os octetos componentes menos o último, tomará o valor 1. O “*bit*” 8 do último octeto indicará, com 0, o fim do identificador; o número é obtido com a concatenação dos 7 “*bits*” restantes em cada octeto, como é mostrado no exemplo da Figura 18.

Codificação do OBJECT IDENTIFIER Ftam ::= {1,0,8571}

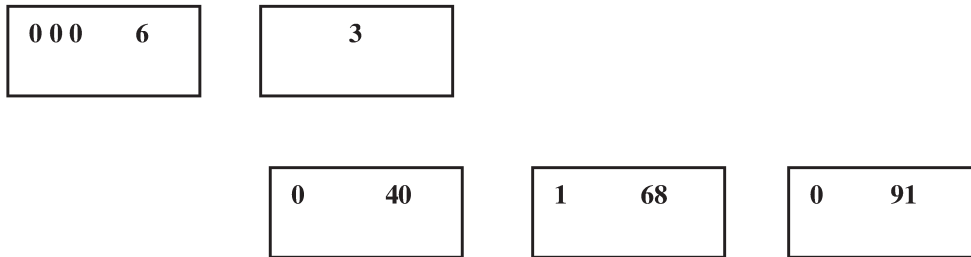


Figura 18 – Codificação do OBJECT IDENTIFIER

- ✓ **TIPOS CONSTRUÍDOS** – BER codifica os tipos construídos através dos tipos simples e utilizando recursão. No caso dos componentes de um tipo construído serem, por sua vez, tipos construídos, as mesmas regras são aplicadas.

A complexidade na codificação destes tipos está em conhecer o tamanho e geralmente é utilizada a forma indefinida descrita anteriormente.

Exemplo

```
Lista ::= SEQUENCE {
    TIPO 1,
    TIPO 2 }
```

Para cada elemento da sequência BER é aplicado como mostra a Figura 19.

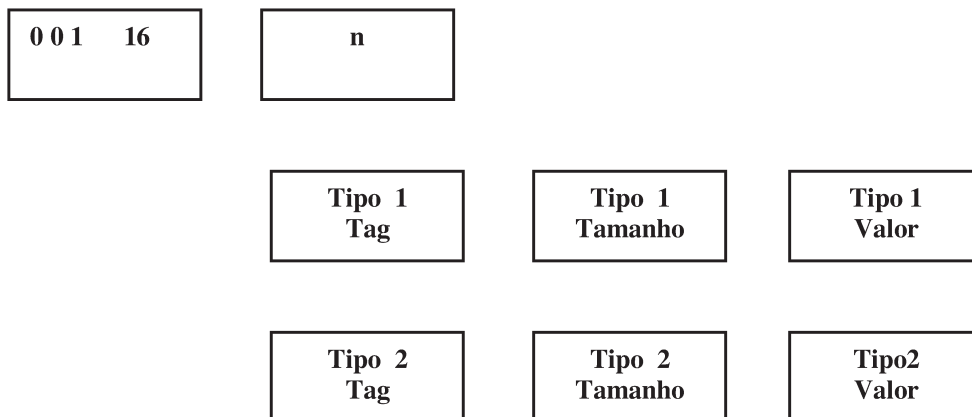


Figura 19 –Codificação de um tipo construtor

Os tipos SEQUENCE OF, SET e SET OF são codificados da mesma maneira, só modificando a “tag”.

- ✓ **CHOICE** – O tipo CHOICE representa uma escolha: logo, o valor a codificar segue a norma BER para o tipo do dado que foi selecionado.
- ✓ **TIPOS TAGGED** – A codificação de um tipo Tagged é derivada da codificação do tipo que aparece na sua definição (tipo básico). Se a palavra chave IMPLICIT não aparece na definição, o tipo será sempre codificado como se fosse um construtor. Caso contrário, o tipo será codificado como primitivo ou construído segundo o tipo da definição, embora a “tag” codificada não seja a do tipo básico.

A forma de codificar os tipos com “tag” adicional, fica mais clara através do conjunto de exemplos seguintes:

a) Tipo1 ::= [APPLICATION 0] Tipo2

No exemplo acima, pode-se observar que a palavra chave IMPLICIT não aparece. Logo a codificação será em forma construída, uma vez que essa definição de tipo “tagged” é similar a:

Tipo1 ::= [APPLICATION 0] IMPLICIT SEQUENCE (Tipo2)

Ambas definições são equivalentes porque, tanto a “tag” APPLICATION, como a “tag” do Tipo2 devem ser codificadas e a codificação é da forma ilustrada na Figura 20.

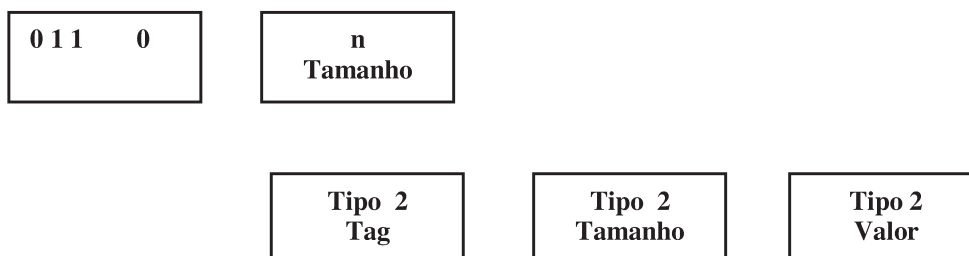


Figura 20 –Codificação do tipo a)

b) Se IMPLICIT é usado e assumindo que o Tipo2 é da forma primitiva temos que

$$\text{Tipo1} ::= [\text{APPLICATION } 0] \text{ IMPLICIT Tipo2}$$

É codificado da forma seguinte (Figura 21):



Figura 21 — Codificação do tipo b)

A utilização do IMPLICIT produz uma economia de octetos de identificação e de tamanho do tipo “tagged”, porque só a “tag” APPLICATION é codificada no octeto de identificação, não sendo necessário codificar a “tag” do Tipo2, uma vez que, ela está implícita na codificação básica nos octetos do campo valor.

Capítulo 3

Especificação do Sistema

Neste capítulo serão detalhados os aspectos funcionais de procedimentos levados em conta para modelar o compilador.

3.1 Detalhes Gerais

O Compilador ASN.1, como foi definido em seus objetivos, deve tornar possível a utilização da ASN.1 em um ambiente de protocolos OSI e fornecer uma representação da ASN.1 numa sintaxe concreta que, idealmente, deve ser a utilizada pela própria implementação do protocolo. Neste caso, a linguagem adotada foi C. Os motivos dessa escolha encontram-se no capítulo 5 deste trabalho.

Ao definir esta ferramenta, foi levado em consideração o ambiente com o qual ela interage: as camadas de aplicação e apresentação. Portanto, o fato de que todo protocolo de aplicação deve agrupar e descrever todas as PDU's que o compõem num só módulo ASN.1 e logo requerer os serviços de apresentação para transformar esta sintaxe abstrata numa sintaxe

de transferência, usando para tal uma sintaxe concreta como intermediária, determina que o sistema apresente dois aspectos funcionais diferentes. Na Figura 22, mostra-se estas operações.

O compilador foi visualizado como um sistema de utilização estática, isto é, que não é necessário compilar em tempo de execução do protocolo ou no intercâmbio de dados do mesmo. Porém os resultados da compilação são requeridos dinamicamente pelos processos de comunicação, como os codificadores e decodificadores, que mais tarde, serão apresentados neste capítulo.

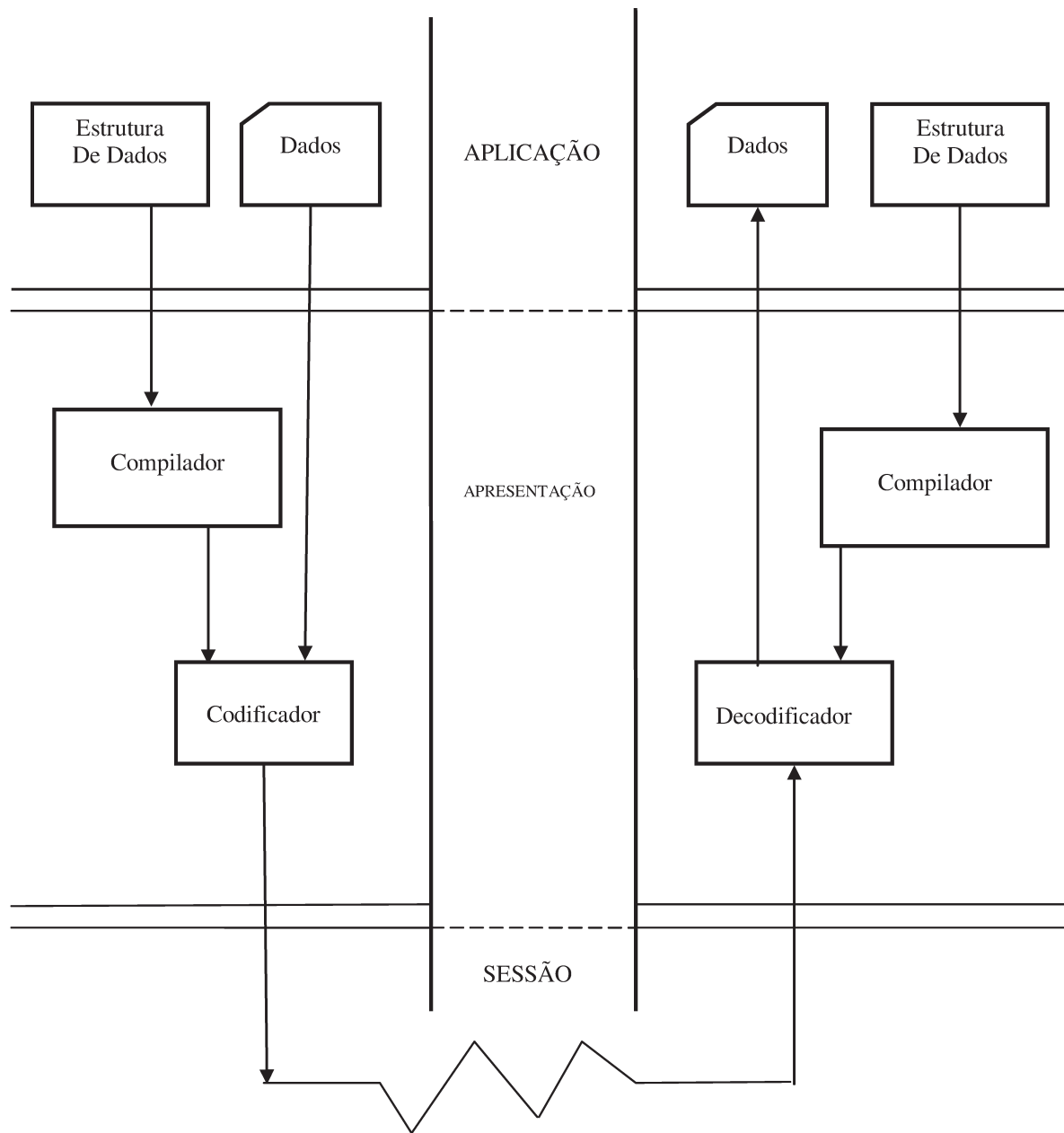


Figura 22 — Camadas OSI onde o sistema atua

3.2 Descrição dos Processos do Sistema

O Compilador ASN.1 é composto de três processos básicos:

- ✓ Compilação da linguagem ASN.1;
- ✓ Geração do codificador C-BER;
- ✓ Geração do decodificador BER-C.

Cada um destes processos realiza um conjunto de funções, destinadas a produzir os meios necessários para efetuar as transformações de ASN.1 no que se refere a seus tipos e a seus valores. A seguir, será visto cada processo e suas funções:

Compilação

Este processo é encarregado de produzir o código C equivalente ao ASN.1, mediante a conversão das estruturas de dados. No próximo capítulo veremos detalhadamente as estruturas de dados envolvidas nesta função.

A entrada para o processo de compilação é um conjunto de especificações ASN.1 agrupadas em um módulo; e a saída gerada pelo processo é um conjunto de especificações de tipos C mais a tabela de símbolos correspondente ao módulo. Esta situação está ilustrada na Figura 23.

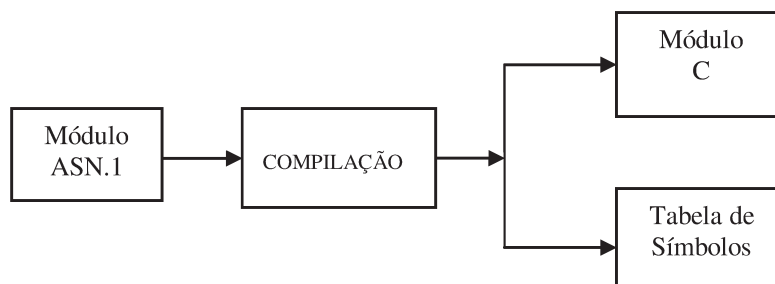


Figura 23 — Processo de Compilação

Conceitualmente, um compilador opera em fases, cada uma das quais, transforma o arquivo de entrada, de uma representação, para outra (Aho, et al., 1988). Na prática algumas destas fases podem ser agrupadas e as representações intermediárias, que deveriam existir entre as fases agrupadas, não necessitam ser construídas. A decomposição do compilador em fases é mostrada na Figura 24.

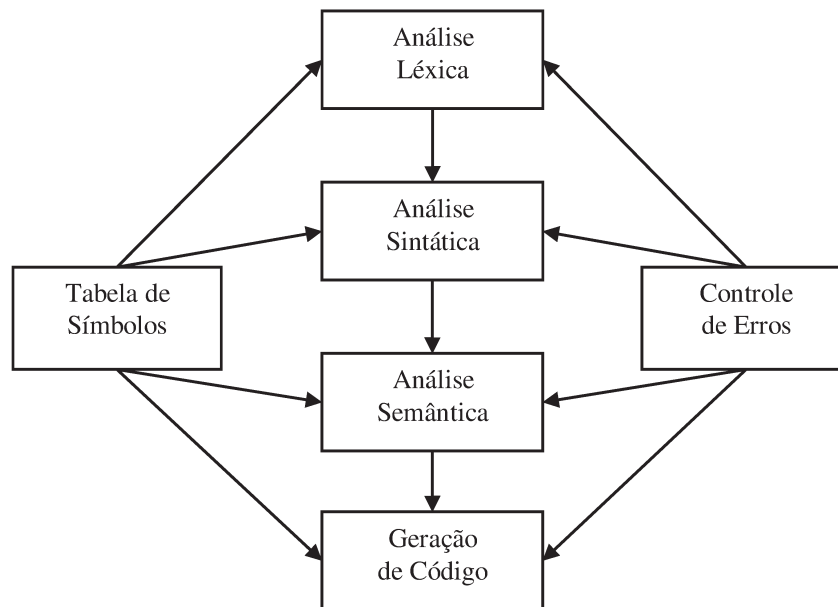


Figura 24 — Fases da Compilação

As três primeiras fases formam, em conjunto, a medula do analisador do compilador, e as fases de controle de erros e manutenção da tabela de símbolos interagem diretamente com as fases de análise e geração de código.

Análise Léxica: é a primeira fase do compilador e sua tarefa principal é ler os caracteres de entrada e produzir, como saída, uma seqüência de “tokens” que serão usados para realizar a análise sintática. A análise léxica deve assegurar que os “tokens” (unidade mínima da linguagem,

como identificadores, símbolos especiais, etc.), estejam escritos corretamente segundo as regras definidas para eles no documentos ISO 8824.

Análise Sintática: a sintaxe da linguagem ASN.1 é descrita por meio da notação BNF (*Backus Normal Form*). A análise sintática é realizada para verificar que todas as sentenças, que são compostas de “*tokens*”, correspondam a essa gramática, isto é, que essa sentença possa ser gerada através da gramática.

Análise Semântica: é feita para verificar a consistência da linguagem. A análise semântica utiliza a análise sintática para identificar os componentes da linguagem que deve avaliar, dependendo da linguagem, em particular o que se considera um erro semântico. No caso da ASN.1, por exemplo, verificar a consistência das “*tags*”, como classe e número delas, corresponde à análise semântica.

Controle de Erros: cada fase da compilação detectará erros diferentes, de acordo com a função que está realizando. Por exemplo, durante a análise léxica detecta-se os erros que correspondem a caracteres ou palavras que não façam parte dos conjuntos definidos para a linguagem. O compilador informará os erros que o código de entrada contém e interromperá sua execução.

Manutenção da Tabela de Símbolos: durante o processo de compilação o compilador cria e mantém uma tabela de símbolos que contém as informações necessárias para cada tipo de dados descrito no módulo ASN.1. A tabela de símbolos é uma estrutura de dados que possui uma entrada para cada identificador de tipo que apareça na análise léxica. As outras fases da análise interagem com a tabela de símbolos adicionando-lhe informações, já que nem todos os atributos de um identificador são conhecidos durante a análise léxica. Um outro tipo de interação com a tabela é o realizado durante a análise semântica e na geração de código, quando é preciso saber se os tipos de dados do módulo-fonte são usados corretamente.

No caso particular deste compilador a tabela não só será utilizada em tempo de compilação, mas também, nos processos de codificação e decodificação dos dados; a tabela de símbolos será um produto adicional da compilação.

Geração da Sintaxe Concreta: a fase final do compilador é a geração de um código executável. Normalmente, isto corresponde ao código de máquina suportado pela máquina onde se executa a compilação. Contudo, para este compilador específico, o código é gerado em

linguagem C, o que podemos chamar de sintaxe concreta porque é a linguagem adotada como intermediária entre a especificação ASN.1 e a codificação.

Geração do Codificador

O processo de geração do codificador consiste na criação de uma rotina de codificação específica para cada módulo ASN.1 compilado, com a qual se realiza a tarefa de codificar os valores dos tipos ASN.1 na forma de sequência de octetos binários, seguindo as regras BER.

A função específica deste módulo é apresentada na Figura 25, onde é possível ver que o processo deve realizar os seguintes passos:

Analisar os tipos ASN.1 no módulo de entrada para identificar os tipos de mais alto nível de abstração, com o objetivo de identificar o tipo principal e gerar uma chamada a uma função que o codifique.

Produzir uma rotina capaz de ler um conjunto de valores de uma PDU e gerar uma codificação para esses tipos. Esta rotina deve ser um programa executável que permita realizar estas funções durante a execução do protocolo.

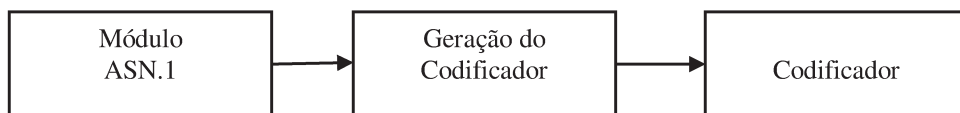


Figura 25 — Processo de Codificação

Geração do Decodificador

O processo inverso de toda codificação de dados é a decodificação. O processo de geração do decodificador é encarregado de entregar ao usuário uma rotina específica de decodificação que corresponda ao módulo ASN.1 que foi compilado. Esta rotina deve ser capaz de:

Analisar o módulo ASN.1 de entrada com o objetivo de identificar, na declaração de tipos, o de maior nível, para posteriormente poder gerar uma chamada que o decodifique;

Gerar uma rotina em linguagem C que seja capaz de ler os dados codificados segundo as regras BER, analisar os octetos de identificação, tamanho etc., e reconstituir o dado original. Esta rotina deve ser um programa executável que permita realizar estas funções independentemente. A Figura 26 representa este processo.

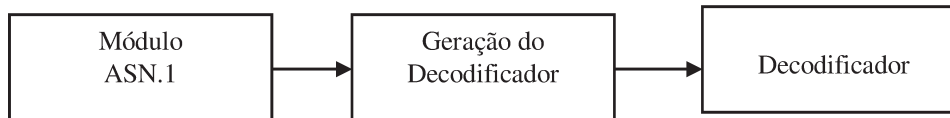


Figura 26 — Processo de Decodificação

3.3 Etapas na Utilização do Compilador

Como foi especificado no início deste capítulo, a utilização desta ferramenta tem dois aspectos: um estático e outro dinâmico. A parte estática foi descrita recentemente e refere-se à compilação “*off-line*” de cada módulo ASN.1 e à geração, de uma só vez, das rotinas codificadoras e decodificadoras em código C.

A segunda etapa, a dinâmica, refere-se à requisição, por parte do usuário, dos produtos da compilação do módulo ASN.1. As estruturas de dados, agora em linguagem C, serão incluídas diretamente, como as declarações das PDU’s, nas implementações de seus protocolos. Este ponto é fundamental, pois deve existir uma absoluta concordância entre os tipos de dados usados para definir os valores que serão entregues aos codificadores, já que um codificador específico está baseado nessas estruturas. São essas estruturas que serão utilizadas para obter os valores a codificar. Tudo isto é válido também para o decodificador específico, porque são as mesmas estruturas que o decodificador devolve ao usuário, uma vez concluída a tarefa.

As rotinas geradas na compilação serão utilizadas pela camada de apresentação, cada vez que um serviço de codificação ou decodificação seja necessário, sendo responsabilidade da camada verificar se o módulo ASN.1 que contém a especificação das PDU's já foi compilado. Em caso contrário, deve-se voltar à primeira etapa do sistema e compilar o módulo.

Na Figura 27, em relação ao processo “*off-line*”, representam-se as entradas e saídas do sistema, ou seja, os n módulos ASN.1 a serem compilados, assim como suas respectivas transformações em código C, codificador e decodificador.

Os blocos menores representam os valores dos tipos de dados a serem codificados assim como os dados a nível de código BER. Seguindo as linhas de fluxo, pode-se observar como o sistema se comporta quando é necessário processar os dados dinamicamente, sendo os codificadores e decodificadores requeridos “*on-line*” pela camada de apresentação.

Um ponto importante a ser considerado é o que se refere às interações do sistema com o meio em que ele deve atuar. Podemos distinguir claramente duas interfaces a serem desenvolvidas:

Interface com o compilador: refere-se à comunicação do protocolo usuário com o compilador, na forma em que as especificações ASN.1 do protocolo serão passadas para a compilação;

Interface com Apresentação: refere-se à coordenação e comunicação para os requerimentos de codificação dos valores, e decodificação dos mesmos, feita pela camada de apresentação tendo em vista a utilização das rotinas geradas pelo compilador.

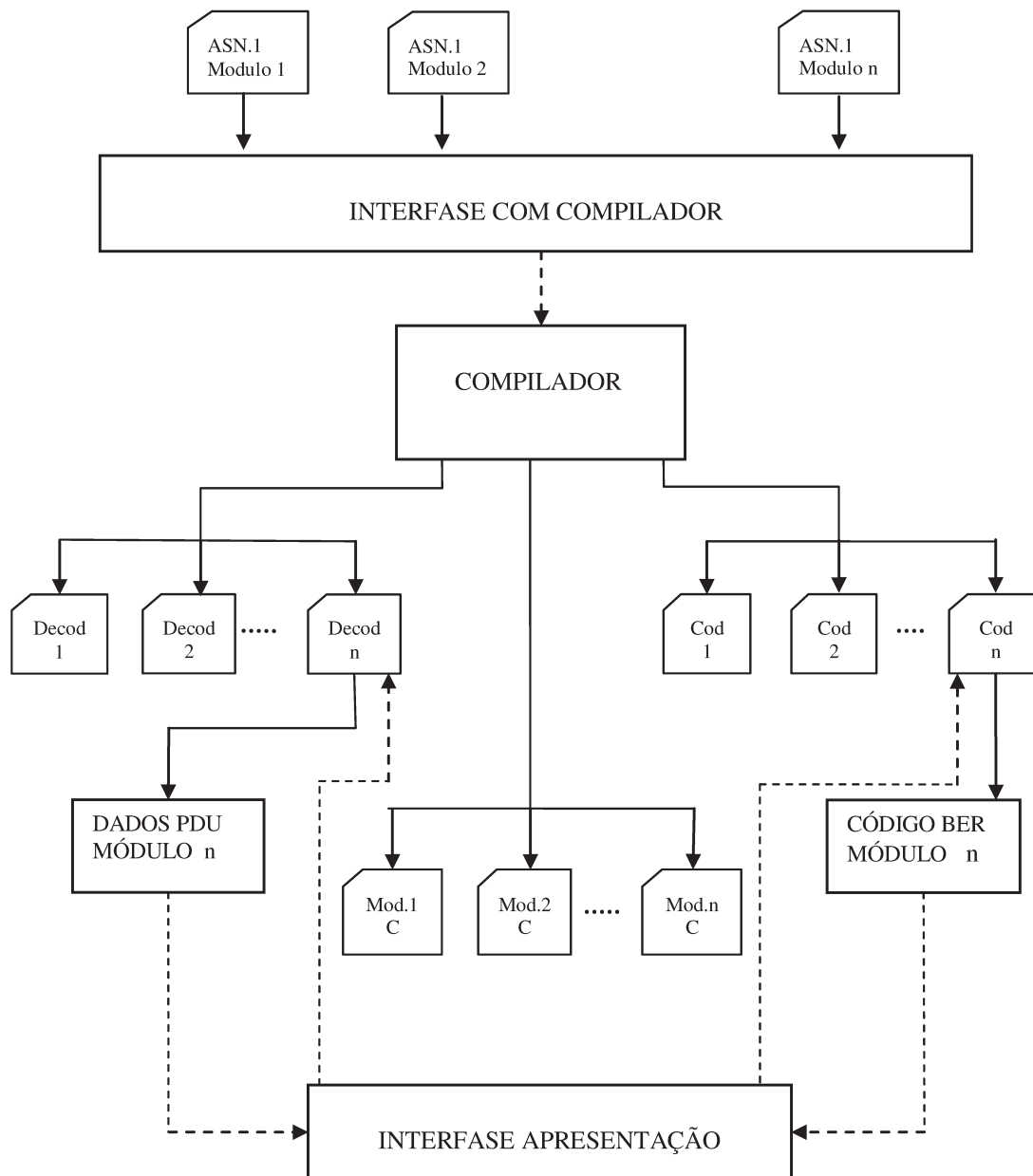


Figura 27 – Uso do Compilador Estática e Dinamicamente

3.4 Limitações do Sistema

Como será explicado em maior detalhe no próximo capítulo, o sistema possui algumas limitações referentes à gramática que deve compilar: no caso específico da notação MACRO, o compilador só tratará das macros do protocolo ROSE.

Pode-se considerar uma limitação o fato de se restringir ao uso da linguagem C para gerar a sintaxe concreta do ASN.1, não sendo estendidas estas facilidades para outras linguagens.

A comunicação do sistema com o meio é feita através de arquivos de texto para o módulo ASN.1 e arquivos de valores, com extensões “c”, em caso das rotinas de codificação, decodificação e código C. O usuário têm que executar as adaptações necessárias.

Capítulo 4

Transformação da Linguagem ASN.1 para C

Neste capítulo abordaremos os aspectos de transformação da linguagem ASN.1, para um conjunto de tipos de dados definidos em C, discutindo-se, para esse propósito, as dificuldades de compatibilizar ambas as linguagens

4.1 Comparação entre ASN.1 e C

Para realizar uma transformação adequada da linguagem ASN.1 para C, devemos ter em consideração as diferenças que existem entre as duas linguagens, de forma a obter um conjunto consistente de estruturas de tipos de dados em código C (Kernigham, et al., 1978).

Basicamente, a principal divergência entre ambas é a forma como são tratadas as declarações de tipos. C define seus tipos de dados “*bottom-up*”, isto é, do menor nível de abstração ao maior, começando sempre suas declarações pelos tipos mais atômicos, com o objetivo de provar uma maior facilidade para realizar a análise sintática e uma melhor

compreensão da formação de suas estruturas. ASN.1 faz suas declarações na forma contrária “*top-down*”, uma vez que, sendo o objetivo do ASN.1 descrever num módulo todas as PDU’s que um protocolo utiliza, o maior nível de abstração corresponde precisamente às declarações de conjunto de PDU’s. Torna-se então necessário para a geração do código C, realizar a inversão da especificação ASN.1 e executar a compilação sobre esse código ASN.1 “*bottom-up*”. A geração dos codificadores e decodificadores não é baseada no código invertido, que não é do conhecimento do usuário do sistema.

Outra consideração a ser feita diz respeito aos nomes utilizados por ASN.1, que não obriga que cada elemento dentro de uma estrutura, tenha um nome que o identifique. Já C considera cada tipo como identificado com um nome único. Logo, quando não existe um nome em uma estrutura SET ou SEQUENCE, será criado um nome concatenando o nome da estrutura, a palavra “*aux*” e um número que identifique o elemento univocamente. Uma segunda diferença com respeito aos nomes de tipos é que os nomes ASN.1 aceitam o caracter “-” durante a compilação.

Uma última diferença, é que ASN.1 não limita os tamanhos de seus tipos primitivos, como por exemplo INTEGER, REAL etc. Para alguns casos, os tamanhos serão limitados pelas especificações da arquitetura em que o compilador é hóspede, em particular o caso dos tipos BITSTRING e OCTETSTRING, resolvendo o problema por meio de estruturas ligadas por apontadores, descritos mais adiante.

4.2 A Gramática ASN.1

Sendo ASN.1 uma linguagem formal possui uma gramática para representá-la, que está definida no documento ISO 8824 (ISO, 1987a) e em seu Adendo posterior (ISO, 1987c). Esta gramática está na forma BNF e é utilizada para gerar o compilador.

A gramática ASN.1 é utilizada praticamente em sua totalidade. Algumas modificações foram introduzidas para tornar factível a sua implementação computacional, devido a alguns problemas de ambigüidade, por exemplo, nas produções mostradas na Figura 28.

```
Sequencetype ::= SEQUENCE { elementTypelist } | SEQUENCE { }  
SequenceofType ::= SEQUENCE OF Type | SEQUENCE
```

Figura 28 —Notação BNF para o tipo SEQUENCE

A produção `Sequencetype ::= SEQUENCE { }` foi eliminada já que produz um conflito: pode ser representada como uma seqüência vazia, o mesmo acontecendo no caso de SET e Setoftype.

No caso do tipo ANY, a produção aceita é a especificada na norma. No que se refere à modificação feita no adendo.

```
Anytype ::= ANY | ANY DEFINED BY identificador
```

só se utiliza a primeira parte da regra, indicando-se o tipo no momento da codificação por meio de um identificador.

No caso das produções que definem a gramática dos valores dos tipos, só serão usadas em casos muito restritos, como por exemplo, se a opção DEFAULT é usada. A maior parte será reconhecida e ignorada porque é mais prático obter os valores para os tipos de dados por meio de outros mecanismos próprios do protocolo que esteja usando o compilador. A gramática aceita no caso da opção DEFAULT é a que se representa na Figura 29.

```

Value : IDENT
      | NUMERO
      | objectidentifiervalue
      | octetstringvalue
      | characterstringvalue
      ;

```

Figura 29 — Produção que define os valores aceitos por “DEFAULT”

4.3 Transformações de tipos ASN.1 para C

A idéia básica da transformação consiste em traduzir um tipo ASN.1 em um tipo C, que permita armazenar o valor do tipo ASN.1: por exemplo, um tipo “*int*” em C na SUN está representado por quatro “*bytes*” que contêm os valores atribuídos ao “*int*”. Desta forma, associando o INTEGER ASN.1 ao “*int*” do E, dá-se representação para INTEGER. Como se sabe, cada tipo ASN.1 definido carrega consigo o tipo, ou criam um novo tipo. Estas propriedades não possuem equivalência, em nenhum tipo simples do C. Portanto, uma transformação direta de INTEGER para “*int*” não seria possível e uma estrutura deveria ser criada para representá-lo. Então para um INTEGER declarado da seguinte forma:

Numero ::= [APPLICATION 1] IMPLICIT INTEGER

seria necessário relacioná-lo com a estrutura:

```

struct INTEGER
{
    int Numero;
    int Implicit;
    int tag;
    int numerotag;
};

```

Se um tipo INTEGER estivesse declarado dentro de um construtor, ele teria outras opções a serem consideradas. É fácil ver que toda esta informação não precisa ser guardada, junto com o valor do dado, o que produziria repetições. Então o compilador, para resolver a situação, armazenará sempre todos os dados referentes a um tipo na sua tabela de símbolos, na qual existirá uma entrada para cada tipo declarado a ser preservada, inclusive para o tempo da codificação.

4.3.1 TIPOS PRIMITIVOS

A base da codificação e decodificação dos valores dos tipos ASN.1 encontra-se nos tipos primitivos que a linguagem proporciona. Felizmente, para alguns deles existe um tipo similar diretamente em C, como é o caso dos tipos INTEGER, REAL e NULL que são traduzidos em *“int”*, *“float”* e *“int”* respectivamente, já que o sentido e as propriedades atribuídas a eles são as mesmas em ambas as linguagens.

Para os outros tipos primitivos foram definidas estruturas de dados que representam cada um deles, como será explicado adiante.

O tipo primitivo BOOLEAN pode ser traduzido diretamente como um *“int”*, que tomará dois valores: 0 em caso de o valor ASN.1 ser FALSE e 1 no caso contrário:

```
int BOOLEAN;
```

Os tipos de dados primitivos mais característicos de ASN.1 são os tipos OCTETSTRING e BITSTRING que, como foram definidos, representam *“strings”* binários; para representá-los foram criadas duas estruturas similares, uma vez que, a diferença entre ambos os tipos só se faz presente no momento de codificar os dados contidos nelas. Como ASN.1 não especifica o tamanho dos valores que os tipos de dados podem armazenar, não é possível saber quanta memória é necessária para representar um destes tipos. Portanto, seria ineficiente traduzí-los como uma cadeia de caracteres, e é preferível representá-los como uma lista ligada, em que cada

nó possui um espaço de memória fixo múltiplo de oito. No caso de se precisar mais espaço de armazenamento, outro nó pode ser alocado.

As seguintes estruturas representarão os tipos OCTETSTRING e BITSTRING.

```
typedef char valor[80];
```

“valor” define uma cadeia com tamanho múltiplo de 8 para armazenar partes dos tipos OCTETSTRING e BITSTRING:

```
struct octetstring{
    valor dado;
    struct octetstring *next;
} octetstring;

struct bitstring{
    valor dado;
    struct bitstring *next;
} bitstring;
```

A estrutura *“octetstring”* possui um campo *“dado”*, de tamanho padrão, que conterà o valor total ou parcial de OCTETSTRING, e um campo apontador (*“next”*) para outra estrutura octetstring, em caso de ser necessário extensão. Para o tipo BITSTRING a estrutura é similar.

Para os tipos OBJECT, só foi definida uma estrutura para o tipo OBJECT IDENTIFIER, já que o tipo OBJECT DESCRIPTOR, como sua própria definição diz, pode ser representado como um OCTETSTRING.

Um OBJECT IDENTIFIER é definido como uma seqüência de números que correspondem às árvores de derivação, ISO, CCITT ou ambas. Os *“labels”* que acompanham os números são padronizados para sua representação. Portanto, só é necessário guardar esses números. A seguinte estrutura é utilizada para representar um OBJECT IDENTIFIER:

```
objectident struct {  
    int num [30];  
    int numtot;  
} objectident;
```

A estrutura possui dois campos: o primeiro é um vetor que armazena os números e possui tamanho trinta, já que se supõe que não existirão mais de trinta sub-índices em um OBJECT IDENTIFIER; o segundo campo da estrutura indica efetivamente quantos indicadores possui o tipo.

O tipo ANY representa um dos tipos ASN.1 mais complicados. Devido à sua própria definição, só se estabelecerá o tipo a ser usado na codificação, no momento em que o valor seja atribuído. Na prática, isso seria impossível de se deduzir. Logo, no momento da codificação deste tipo de dado, esta condição de ANY deve desaparecer, indicando-se de alguma forma ao codificador, o tipo que esse campo terá nesse instante. ANY será transformado numa estrutura que contém dois campos:

```
any struct {  
    valor nometipo;  
    void *ptrany;  
} any;
```

O primeiro campo “nometipo” representa um identificador do nome de tipo pelo qual o tipo ANY será substituído. Este nome deverá ser passado pelo usuário junto com os valores do tipo, no momento da codificação. O campo “ptrany” é um apontador para qualquer tipo de dado. Será associado a ele o tipo identificado pelo “nometipo”.

4.3.2 TIPOS CONSTRUTORES

Como foi visto no capítulo 2, ASN.1 apresenta facilidades para criar novos tipos de dados, que podemos chamar de estruturas de dados. A seguir descreveremos como eles serão representados em C.

SEQUENCE

Um tipo SEQUENCE é uma lista de campos de diferentes tipos, inclusive um tipo construtor. Já um tipo “*struct*” do C é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sob um único nome. Logo esses tipos de dados são semelhantes e um tipo SEQUENCE pode ser transformado numa “*struct*”, e seus tipos componentes mapeados diretamente no tipo C que seja correspondente. Um exemplo é o apresentado na Figura 30.

Em ASN.1:

```
Dado1 ::= SEQUENCE {  
    nome ::= INTEGER;  
    valor ::= OCTETSTRING;  
}
```

Em C:

A representação desta estrutura é

```
struct Dado1 {  
    int nome;  
    octetstring valor;  
} Dado1;
```

Figura 30 – Representação de SEQUENCE em uma “*struct*”

O tipo SEQUENCE outorga a seus campos componentes outras atribuições que são definidas através das produções gramaticais da Figura 31.

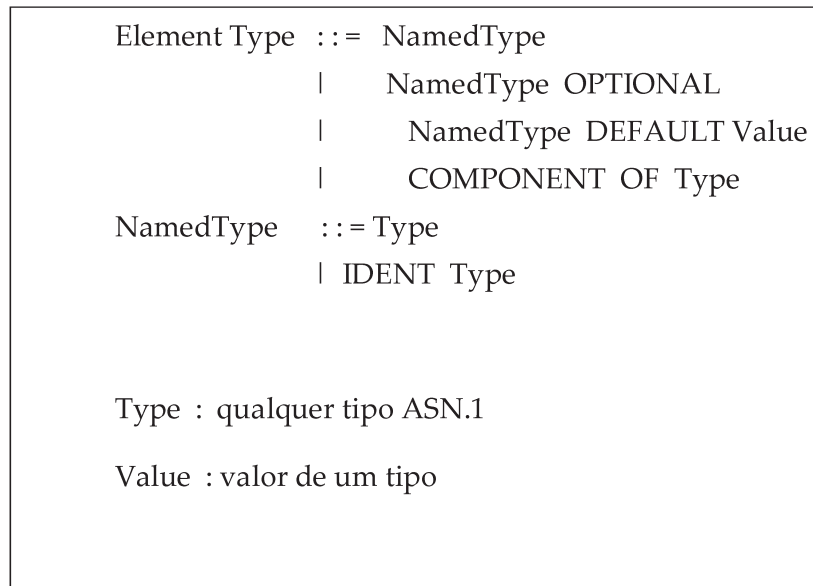


Figura 31 —Regra gramatical que define as variantes para um componente de tipo SEQUENCE ou SET

Analisando cada uma destas produções observa-se que a primeira alternativa da produção “ElementType” e por sua vez a produção “NamedType”, é composta de duas opções. A primeira “NamedType ::= Type”, implica em que só se indica o tipo do campo, faltando o seu nome, que deve ser criado seguindo a regra descrita no ponto 4.1 deste capítulo. A segunda “NamedType ::= IDENT Type”, é a única que corresponde diretamente a uma declaração de tipo em C, possuindo um tipo e um nome para esse tipo. Este caso é apresentado no exemplo da Figura 30.

A segunda opção da produção “Element Type” apresenta a palavra OPTIONAL que indica que o campo referido por NamedType pode ou não estar presente. Como não existe em C um tipo que tenha esta propriedade, será necessário simulá-lo através da criação de um campo adicional que será um novo componente da estrutura. Este campo atuará como um indicador, se o tipo que apresenta OPTIONAL está presente ou não, ou melhor, o seu valor no momento da codificação do dado.

O novo campo será um inteiro que tomará dois valores: 0 se o valor do tipo indicado como OPTIONAL não estiver presente, ou 1 em caso contrário. O nome atribuído a esse inteiro será o mesmo nome da variável mais o posfixado “*op*”. A Figura 32 é um exemplo desta situação.

Continuando a análise das regras gramaticais, encontra-se a opção DEFAULT, e a seguir um valor correspondente ao tipo descrito por “*NamedType*”, o que significa que um tipo pode, em um momento, não ter um valor atribuído e assumir o valor definido pela produção Value. A situação é similar à produzida por OPTIONAL, havendo que definir no momento da codificação se o valor do tipo está presente ou não. Novamente, a situação é resolvida adicionando um novo campo na estrutura em C: o nome do campo será criado igual ao OPTIONAL, com a diferença que termina com o posfixado “*def*”.

A última opção de “*ElementType*” é “*COMPONENTS OF Type*”, que representa uma inclusão de todos os campos de um tipo construtor indicado por Type dentro do SEQUENCE em que ela se encontra. É semelhante a uma união, em que os campos devem ser incluídos na estrutura como novos componentes dela. A Figura 33 apresenta esta situação.

```
Dado2 ::= SET {  
    var1 INTEGER;  
    var2 OCTETSTRING;  
    var3 BITSTRING OPTIONAL;  
}
```

Sua representação em C será:

```
struct Dado2 {  
    int var1;  
    octetstring var2;  
    bitstring var3;  
    int var3op;  
} Dado2;
```

Figura 32 —Exemplo de OPTIONAL em um SET

```
Dado3 ::= SEQUENCE
        { nome1  INTEGER;
          COMPONENTS OF Dado1 }

struct Dado3 {
    int nome1;
    int nome;
    octetstring valor;
} Dado3;
```

Figura 33 —Exemplo de COMPONENTS OF, Dado1 se encontra na Figura 31.

SET

SET representa um conjunto de elementos cuja interpretação em C é idêntica a SEQUENCE. Todas as regras de conversão para traduzir SEQUENCE são aplicadas para SET, sendo a única diferença o ordenamento dos componentes, que só necessita ser considerado no momento da codificação dos valores de ambos os tipos.

SEQUENCE OF e SET OF

Os tipos SET OF e SEQUENCE OF representam um tipo “array” em C, sendo que, pela definição de ambos, não é possível determinar o tamanho do “array” no momento da declaração do tipo. Por exemplo, um SEQUENCE OF de um tipo X pode ter, em determinados momentos, um número diferente de elementos. Assim, não é possível a sua tradução por um “array” de X. A forma mais prática de representar esses tipos, é uma lista ligada de apontadores para um tipo de dado, onde o tamanho da lista será determinado em tempo de execução. A lista da Figura 4.7 representa um exemplo de um tipo SET OF.


```

EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE
{
  direct – reference    OBJECT IDENTIFIER OPTIONAL,
  indirect-reference    INTEGER OPTIONAL,
  data-value-descriptor OBJECT DESCRIPTOR OPTIONAL,
  encoding              CHOICE
{
  single-ASN1-type      [0] ANY
  octet-aligned          [1] IMPLICIT OCTETSTRING
  arbitrary              [2] IMPLICIT BITSTRING } }

```

Cada campo do SEQUENCE contribui para identificar o contexto onde o dado EXTERNAL se encontra e para definir a sua codificação. Portanto, este tipo será traduzido como uma “*struct*” que corresponde diretamente à definição ASN.1. A definição C do EXTERNAL será então:

```

external struct
{
  objectident  diret_reference;
  int          indirect_reference;
  octetstring  data_value_descriptor;
  int encoding;

  union componentes
  {
    any          single_ASN1_type;
    octetstring  octet_aligned;
    bitstring    arbitrary;
  } componentes;
} external;

```

4.4 Macros

Como foi dito no capítulo 2 deste trabalho, as facilidades que as Macros proporcionam afetam diretamente a gramática da linguagem, já que permitem que esta seja entendida ao criar novas regras gramaticais para definir os novos tipos e valores que atendam aos requerimentos do usuário não alcançados com os tipos básicos ASN.1.

É essa propriedade das macros ASN.1 que gera o principal problema na sua compilação. Devemos considerar que a maioria dos compiladores modernos são divididos, teoricamente, em duas fases: a primeira é a que lê o arquivo de entrada e constrói um código interno que corresponde à gramática entendida pelo compilador; e a segunda fase, que toma essa forma interna e produz um conjunto de lexemas de saída como, por exemplo, um código “assembler”.

Para construir a primeira fase de um compilador, a gramática que será entendida por ele deve estar totalmente especificada. Lamentavelmente quando escrevemos um compilador ASN.1 baseado nesta teoria, não é possível conhecer quais e que tipo de macros serão encontradas, isto é, quando o compilador encontra uma macro, não entenderá o código e conseqüentemente produzirá uma falha.

Outro inconveniente que as macros apresentam, é que grande parte da informação necessária para sua interpretação, deve ser explicitada a nível de comentário, o que não é reconhecido por um compilador.

Uma forma de solucionar o problema poderia ser através da utilização de algum outro modelo de compilador, por exemplo, um compilador que possa aprender novas regras sintáticas quando estas aparecem em uma macro.

Um compilador desta natureza não faz parte dos objetivos deste trabalho. Por conseguinte, uma outra solução foi considerada estendendo a gramática atual ASN.1 com as regras gramaticais que definem um grupo específico e determinado de macros conhecidas. Esta é uma solução parcial, já que, para compilar uma macro fora das selecionadas, sua gramática deverá ser inserida explicitamente no compilador.

Neste trabalho adotamos esta solução e o conjunto de macros escolhido é o que faz parte da RO-NOTATION do protocolo ROSE (*Remote Operation Service Element*), definido no documento ISO 9072 (ISO, 1989).

A decisão de compilar essas Macros deve-se ao fato de que o protocolo ROSE tem sido implementado no ambiente do SID MAP, projeto em que está inserido este trabalho.

As Macros do ROSE são quatro: BIND, UNBIND, OPERATION e ERROR. Elas são utilizadas no processo de comunicação interativa entre entidades de aplicação nas chamadas operações remotas.

Cada uma destas macros tem uma função específica dentro deste contexto. Tipicamente, uma entidade de aplicação requer que uma operação particular seja executada: a entidade par atende à execução da operação e reporta o seu resultado.

A macro BIND define um tipo de operação BIND e os tipos para os dados do usuário a serem intercambiados na fase de estabelecimento de uma associação de aplicação. A macro UNBIND permite a especificação de um tipo de operação chamado “unbind” e especifica os tipos de dados de usuário a serem utilizados no desligamento de uma associação de aplicação. A macro OPERATION permite a especificação de uma operação e dos tipos de dados utilizados na requisição da operação e, finalmente, a macro ERROR destina-se à especificação dos tipos de dados do usuário a serem intercambiados, no caso de resposta negativa.

Para compilar essas macros foi necessário gerar uma gramática que as representasse especificamente e, conseqüentemente, uma estrutura de dados para cada uma delas.

A compilação será realizada reconhecendo o tipo de dados definido por alguma destas macros que foi inserido na gramática do ASN.1 como um novo tipo e não como uma macro – como macros elas não podem ser compiladas. O código ASN.1 destas macros será apresentado a seguir.

4.4.1 Gramática das Macros do ROSE

Para reconhecer cada uma das macros do protocolo ROSE foi gerado um novo conjunto de produções a partir da definição ASN.1 das mesmas. Este conjunto de produções define os novos tipos de dados a serem inseridos na gramática original.

Macro BIND

```
Bindtype    : BIND
              argumentbind
              resultbind
              errorbind
              ;
```

Macro UNBIND

```
Unbindtype : UNBIND
            argumentbind
            resultbind
            errorbind
            ;

argumentbind : /* vazio */
              | ARGUMENT
              TYPeref
              ;
```



```
resultbind      : /* vazio */  
                  | ARGUMENT  
                  TYPeref  
                  ;  
  
errorbind       : /* vazio */  
                  | BIND-ERROR  
                  TYPeref  
                  ;
```

Macro OPERATION

```
Operationtype   : OPERATION  
                  argumentbind  
                  resultoperation  
                  erroroperation  
                  linked  
                  ;  
  
resultoperation : /* vazio */  
                  | RESULT  
                  resulttype  
                  ;  
  
resulttype      : /* vazio */  
                  | namedtype  
                  ;
```

```
erroroperation : /* vazio */  
              | ERROR  
              CHAVEABERTA  
              errornames  
              CHAVEFECHADA  
              ;  
  
errornames    : /* vazio */  
              | errorlist  
              ;  
  
errorlist     : error  
              | errorlist  
              VIRGULA  
              error  
              ;  
  
error         : IDENT  
              | type  
              ;  
  
linked        : /* vazio */  
              | LINKED  
              CHAVEABERTA  
              linkedoperationnames  
              CHAVEFECHADA  
              ;
```

```
linkedoperationnames : /* vazio */  
                      | operationlist  
                      ;
```

```
operationlist : operation  
              | operationlist  
              VIRGULA  
              operation  
              ;
```

```
operation : IDENT  
          | type  
          ;
```

Macro ERROR

```
Errortype : ERROR  
          parameter  
          ;
```

```
parameter : /* vazio */  
           | PARAMETER  
           namedtype  
           ;
```

As novas regras gramaticais são inseridas na regra que define os tipos ASN.1 da seguinte forma:

```
Builtintype      : Booleantype
                  | Integertype
                  .
                  .
                  .
                  | Bindtype
                  | Unbindtype
                  | Operationtype
                  | Errortype
                  ;
```

4.4.2 Estrutura C para Macros do ROSE

As estruturas equivalentes em C para as macros seguem o mesmo princípio das estruturas dos tipos definidos ASN.1, isto é, a informação referente ao tipo encontra-se na tabela de símbolos, e na estrutura é necessário reservar espaço para o valor do tipo. No caso das macros do ROSE, elas contêm campos para armazenar o nome do tipo definido nos Argumentos, resultado ou error.

```
Macro BIND      typedef struct BIND {
                  valor argument;
                  valor result;
                  valor error;
                  void * ptrtipo;
                  } BIND;
```

```
Macro UNBIND    typedef struct UNBIND {  
                    valor argument;  
                    valor result;  
                    valor error;  
                    void * ptrtipo;  
                } UNBIND;  
  
                typedef union identoutipo {  
                    valor ident;  
                    void ptr_tipo;  
                } identoutipo;  
  
                typedef struct lista_op {  
                    int oper;  
                    identoutipo tipo_op  
                    struct lista_op *next;  
                } lista_op;
```

```
Macro OPERATION  typedef struct OPERATION {  
    valor argument;  
    int valor result;  
    identoutipo result;  
    struct lista_op *error;  
    struct lista_op *linked;  
    int identtipo;  
    union valorop {  
        int localvalue;  
        objectident globalvalue;} valorop;  
    } OPERATION;
```

```
Macro ERROR      typedef struct ERROR {  
    valor parameter;  
    int identtipo;  
    union valorerr {  
        int localvalue;  
        objectident globalvalue;} valorerr;  
    } ERROR;
```

Capítulo 5

Implementação do Compilador

Neste capítulo são discutidos os aspectos da implementação do compilador ASN.1 e das bibliotecas auxiliares de funções de codificação e decodificação.

5.1 Aspectos Gerais

O compilador ASN.1 foi construído tendo como guia a especificação para os componentes do sistema descrita no capítulo 3.

O sistema foi implementado em equipamento SUN SPARC STATION 370 sob o sistema operacional SUN OS 4.1.1 compatível com UNIX, utilizando as ferramentas LEX (*Lexical Analyser Generator*) e YACC (*Yet Another Compiler Compiler*), além da linguagem C disponível nessa plataforma.

A disponibilidade das ferramentas LEX e YACC no sistema UNIX para auxílio na implementação das fases de análise de um compilador, aliado à economia de tempo e à eficácia do código gerado, foram os principais fatores que motivaram sua utilização.

A ferramenta LEX é um gerador de programas orientado ao processamento léxico de caracteres. O programa produzido reconhece expressões regulares fornecidas pelo usuário sob a forma de regras. O arquivo de entrada é processado com base nessas expressões e transformado

em uma seqüência de unidades básicas “*tokens*” que é fornecida como entrada para o analisador sintático.

Já o YACC, é uma ferramenta de uso geral para reconhecer estruturas sintáticas. Para utilizar o YACC, o programador deve preparar uma especificação do processo, contendo regras que descrevem a estrutura de entrada, o código a ser executado quando as regras são reconhecidas, e uma rotina de baixo nível para fazer a entrada básica. Dessa forma, o YACC gera uma função que controla todo o processo de análise do arquivo de entrada. Essa função, chamada *parser*, efetua a análise a partir de itens básicos, fornecidos pelo analisador léxico. Os “*tokens*” devem estar organizados de acordo com a estrutura das regras que foram especificadas pelo programador, chamadas de regras gramaticais. Quando uma destas regras é reconhecida, uma ação fornecida pelo programador é executada. Caso contrário, uma mensagem de erro é emitida.

A linguagem que é reconhecida pelo *parser*, deve ser especificada por meio de uma gramática de livre contexto, sem regras ambíguas. O método de reconhecimento utilizado é o “*shift-reduce*” para gramáticas LR(1), difícil e complicado de ser implementado manualmente (Aho, et al., 1988).

A Figura 35 representa uma parte da gramática especificada para gerar o *parser* por meio do YACC, correspondente ao Passo 2 do compilador. A gramática está na forma BNF. As ações a serem efetuadas são executadas cada vez que um símbolo terminal (escrito em letras maiúsculas), é reconhecido. A variável *yytext* usada no exemplo, é utilizada pelo LEX para devolver uma cadeia de caracteres que corresponde ao “*token*” reconhecido.


```
assignment    : typeassignment
                ;
typeassignment : TYPeref { if( inicio == NULL )
                          actual = CREA_NO (yytext);
                          else {if (BUSCA(yytext) == NULL)
                                actual = CREA_NO (yytext);
                                else
                                { errores ("Var repetida");
                                  exit (1); }
                              }
                          }
                ASIGNAÇÃO
                Type
                ;
type : builtin_type
     | defintype
     ;
```

Figura 35 — Parte da especificação da gramática do Passo 2 do compilador para YACC

5.2 Arquitetura do Sistema

O sistema é composto de um compilador de dois passos para ASN.1, de um codificador para a sintaxe BER e de um decodificador de BER para C. Cada um destes programas utiliza bibliotecas auxiliares de rotinas básicas programadas em C, que contêm as funções de codificação, decodificação e os tipos C para ASN.1. A seguir, é feito um detalhamento de cada um dos componentes do sistema, descritos na Figura 36.

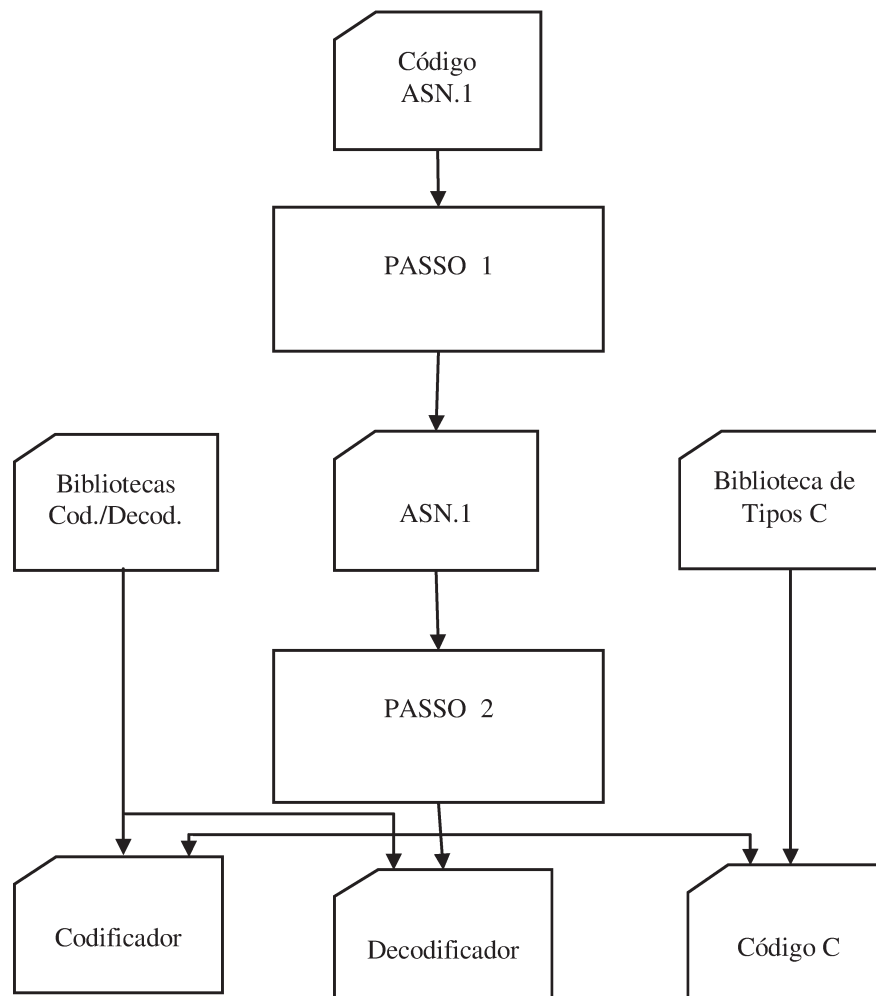


Figura 36 —Arquitetura do Compilador ASN.1

5.3 Primeiro Passo do Compilador (Passo 1)

As fases da compilação, expostas no capítulo 3, não podem ser implementadas em um único passo como seria a condição ideal de um compilador. Isto se deve, basicamente, à forma segundo a qual os tipos de dados são declarados em ASN.1, descrita no capítulo anterior.

O Passo 1 do compilador é responsável pela transformação do módulo de entrada, escrito em ASN.1, para sua forma invertida ainda em ASN.1. Esse passo reúne basicamente as funções que não estão diretamente relacionadas com a compilação.

Funções do Passo 1:

- ✓ Inversão do código de entrada e análise léxica
- ✓ Geração do programa Codificador
- ✓ Geração do programa Decodificador

A única fase do processo de compilação tradicional presente neste passo é a análise léxica. A justificativa para sua inclusão é explicada em seguida juntamente com o detalhamento das funções deste passo.

5.3.1 INVERSÃO DO CÓDIGO DE ENTRADA E ANÁLISE LÉXICA

Esta função efetua a transformação do arquivo de entrada contendo a especificação do protocolo escrito em ASN.1, na sua forma “*bottom-up*” (invertida) a ser usada no passo seguinte.

Para poder inverter o código de entrada, esta função realiza uma análise sintática reduzida visando identificar cada uma das declarações de tipo (por exemplo, onde começa e termina uma declaração SEQUENCE), incluindo todos os seus tipos componentes, sem fazer, contudo, a análise sintática de cada um deles. Embora esta análise sintática seja reduzida, é

necessário efetuar a identificação dos “tokens” do arquivo de entrada, o que justifica a inclusão do analisador léxico nesse passo.

O processo de inversão propriamente dito, é obtido com o auxílio de uma estrutura tipo pilha, onde são inseridas as declarações de tipos ASN.1, à medida que elas vão sendo identificadas no arquivo de entrada. Essa operação repete-se até que todo o arquivo tenha sido processado. Para gerar o arquivo invertido, basta efetuar acessos à estrutura de pilha na sua forma tradicional. O resultado desta operação é um arquivo denominado “*asninvert asn*”, que é usado como entrada para o segundo passo do compilador e como ponto de comunicação entre ambos os passos. Um exemplo de ambos os códigos (entrada e saída) deste passo é apresentado na Figura 37. Simultaneamente a estes processos, é feita a troca de caracteres, quando necessário, como pode-se observar no nome de módulo da Figura 37. O passo 1 está preparado, ainda, para detectar possíveis erros léxicos e alguns sintáticos e emitir mensagens contendo o possível tipo de erro e a linha do código onde ele foi detectado. A detecção de um erro implica no fim da execução do programa, não sendo feita nenhuma tentativa de recuperação automática do erro; essa tarefa é deixada para o usuário.

5.3.2 GERAÇÃO DO PROGRAMA CODIFICADOR

A codificação do ASN.1 para a sintaxe de transferência BER é uma das principais funções da camada de apresentação, e é responsável pela transformação de cada tipo e valor de um dado em uma sequência de octetos. Para realizar esta tarefa, o primeiro passo do compilador, à medida que compila as estruturas gramaticais do módulo ASN.1, vai montando simultaneamente uma rotina codificadora específica para o módulo. A idéia principal é que cada módulo possa ser codificado “*on-line*”, de modo que, cada vez que seja necessário enviar uma PDU, esta não necessita ser compilada novamente, uma vez que sua rotina codificadora já estará pronta.

```

a) Asn-prova1 DEFINITIONS ::=
BEGIN
Personelrecord ::= [APPLICATION 0] IMPLICIT SEQUENCE
{
  Name,
    title          [ PRIVATE 0 ] BOOLEAN,
    number         [1] EmployeeNumber OPTIONAL,
    dateofhire     [2] Date,
    nameofspouse   [3] Name
    children       [4] IMPLICIT SEQUENCE OF ChildInformation }
Childinformation ::= SET
{
  Name,
    dateofbirth    [0] Date }
Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
{
  givenName  OCTETSTRING,
  initial    OCTETSTRING,
  familyName OCTETSTRING }
EmployeeNumber ::= [ APPLICATION 2 ] IMPLICIT INTEGER
Date            ::= [ APPLICATION 3 ] IMPLICIT OCTETSTRING
END

b) Asn-prova1 DEFINITIONS ::=
BEGIN
Date            ::= [ APPLICATION 3 ] IMPLICIT OCTETSTRING
EmployeeNumber ::= [ APPLICATION 2 ] IMPLICIT INTEGER
Name ::= [ APPLICATION 1 ] IMPLICIT SEQUENCE
{
  givenName  OCTETSTRING,
  initial    OCTETSTRING,
  familyName OCTETSTRING }
Childinformation ::= SET
{
  Name,
    dateofbirth [0] Date }
PersonelRecord ::= [APPLICATION 1] IMPLICIT SEQUENCE
{
  Name,
    title          [ PRIVATE 0 ] BOOLEAN,
    number         [1] EmployeeNumber,
    dateofhire     [2] Date,
    nameofspouse [3] Name,
    children       [4] IMPLICIT SEQUENCE OF ChildInformation
  }
END

```

Figura 37—Inversão do código ASN.1 a) “*top-down*” b) “*bottom-up*”

Para construir esta rotina é necessário realizar chamadas a funções codificadoras específicas que se encontram em uma biblioteca que engloba todas as funções primitivas de codificação. Essa biblioteca é denominada “codbiblio.c”. A rotina codificadora receberá por nome “codnome_do_modulo.c”. A Figura 38 mostra um exemplo de rotina codificadora. A biblioteca de funções de codificação é detalhada mais adiante nesse capítulo.

```
include <stdio.h>
include "tiposasn.c"
include "tabela.c"
include "codbiblio.c"
main ( )
{ abre_arquivo("dados.c");
  abre_arquivotab("modulotab.c");
  abre_arquivocod ( );
  RECUPERA( );
  codifica(nometipo1);
  codifica(nometipo2);
}
```

Figura 38—Esquema de um programa codificador para um módulo ASN.1

5.3.3 GERAÇÃO DE PROGRAMA DECODIFICADOR

A função de gerar a rotina decodificadora é análoga à geração da função codificadora. A toda operação efetuada na geração da rotina de codificação, corresponde uma operação para a geração da rotina de decodificação. Quando uma sentença da gramática é analisada corretamente pelo *parser*, o compilador se encarrega de gerar a chamada correspondente a uma função decodificadora específica para o tipo de dado que está sendo compilado. A rotina

decodificadora utiliza as funções primitivas de decodificação de tipos que se encontram na biblioteca “decodbiblio.c”. Essa biblioteca será também melhor detalhada mais adiante. A rotina gerada pelo compilador é chamada “nome_do_modulodecod.c”, e é utilizada dinamicamente pelo usuário, sempre que seja necessário decodificar os dados correspondentes às PDU’s. Um exemplo dessa rotina é mostrado na Figura 39. O programa lê um arquivo denominado “codber.c”, onde se encontram os dados codificados e gera o arquivo “dados.c” onde grava os dados na sua forma decodificada, correspondente aos tipos de dados do arquivo “tiposasn.c”.

```
include <stdio.h>
include "tiposasn.c"
include "tabela.c"
include "decodbiblio.c"
main ( )
{ abre_arquivo("codber.c");
  abre_arquivotab("modulotab.c");
  abre_arquivodecod ( );
  RECUPERA( );
  decodifica(nometipo1);
  decodifica(nometipo2);
}
```

Figura 39 —Esquema de um programa decodificador para um módulo ASN.1

5.4 SEGUNDO PASSO DO COMPILADOR (PASSO 2)

O segundo passo do compilador é onde se realiza a compilação do módulo ASN.1. Ou seja, ele compreende o processo de geração de código C para as estruturas ASN.1 e reúne todas as fases da compilação (descritas no capítulo 3). São elas:

- ✓ Análise sintática da ASN.1
- ✓ Análise semântica da ASN.1
- ✓ Geração de código C

5.4.1 ANÁLISE SINTÁTICA

A análise sintática completa para a gramática ASN.1 que é aceita pelo compilador, descrita no Anexo A deste trabalho, é realizada neste último passo. A análise é efetuada por uma função chamada `yyparser()`, gerada mediante a utilização da ferramenta YACC. Esta função precisa de uma rotina de leitura e análise dos “*tokens*” do código de entrada (arquivo “*asninvert.c*”), que é provida pelo programa “*yy.lex.c*”, gerado pelo uso da ferramenta LEX sob um arquivo de especificações de expressões regulares que resumem as palavras da linguagem ASN.1. Cada vez que uma sentença gramatical é reconhecida pelo analisador sintático, um conjunto de ações que tratam da geração e manutenção da tabela de símbolos do compilador é executado. Cada tipo de dado sintaticamente correto é incluído nessa tabela, juntamente com o identificador do dado, além de outras informações, que auxiliam não só o processo de compilação, como também a geração de rotinas de codificação e decodificação. Para cada tipo são armazenadas as seguintes informações: classe do tipo, “*tag*” explícita ou implícita, número

da “tag”, característica do tipo (definido pelo usuário ou tipo básico), no caso de tipo construtor, informações sobre seus componentes, etc.

Para manipular esta tabela, foi desenvolvido um conjunto de funções de criação, inclusão, preenchimento e busca na tabela, que juntamente com a estrutura de dados que a define, encontram-se armazenadas no arquivo “tabela.c”. Além de compilador, essas funções são também utilizadas pelas rotinas geradas no passo 1.

Finalmente, cada vez que uma sentença gramatical não é reconhecida, um erro sintático é produzido, informando ao usuário o possível tipo do erro e a linha onde foi detectado. Em seguida, a execução do programa é interrompida e os produtos do primeiro passo do compilador são descartados. Não é feita nenhuma tentativa de recuperação dos erros.

5.4.2 ANÁLISE SEMÂNTICA

A análise semântica é responsável pela verificação da consistência de um módulo ASN.1. Cada vez que uma declaração é considerada sintaticamente correta, o compilador, auxiliado pela tabela de símbolos, faz a verificação dos tipos declarados, além de uma validação que é fundamental para os processos de codificação e decodificação. Ou seja, é feita a verificação das seqüências dos números das “tags” das classes APPLICATION e PRIVATE que devem ser únicas em todo o módulo que está sendo compilado, e das seqüências das “tags” de CONTEXT SPECIFIC dentro de cada tipo construtor que fora declarado, forçando que, se a “tag” existe, todos os campos devem possuir uma “tag” com um número único.

As ações executadas em caso de erro, durante a análise semântica, são similares às ações executadas nas etapas anteriores.

5.4.3 GERAÇÃO DE CÓDIGO C

Como foi amplamente explicado no capítulo 4, para cada tipo ASN.1 foi associado um tipo em C. Portanto, cada estrutura ASN.1 possui sua estrutura correspondente em C. Quando

uma declaração de tipo ASN.1 é compilada corretamente, é gerado um código correspondente em C que não é outra coisa senão uma declaração de tipo para as estruturas equivalentes aos tipos básicos ASN.1 descritas no capítulo 4. Essas estruturas encontram-se no arquivo “tiposasn.c”, que deve ser incluído no arquivo que contém as declarações do módulo. Esse arquivo receberá o nome “nome_do_modulotiposc.c”. Nele, encontram-se todas as informações sobre a estrutura dos tipos necessárias para os processos seguintes. Todos os nomes dos tipos ASN.1 são estritamente respeitados para gerar os nomes dos tipos C, sendo unicamente mudado o caracter “-” por “_”. Em caso de não existir o nome ASN.1, ele é criado seguindo a regra definida no capítulo 4 para esse fim. A Figura 40 mostra um exemplo dos tipos C para módulo da Figura 37.

```
“include tiposasn.c”
typedef octetstring Date;
typedef int EmployeeNumer;
typedef struct Name
{
    octetstring givenName;
    octetstring initial;
    octetstring familyName;
} Name;
typedef struct ChildInformation
{
    Name ChildInformationaux1;
    Number number;
    Date dateofhire;
    Name nameofspouse;
    typedef Sequenceof children;
} ChildInformation;
```

Figura 40—Arquivo “ASN_prova1tiposc.c” com o código C correspondente ao do módulo ASN_prova1

5.5 BIBLIOTECA DE CODIFICAÇÃO E DECODIFICAÇÃO

As rotinas de codificação e decodificação geradas pelo compilador formam um conjunto de chamadas a funções que tratam de cada tipo básico do ASN.1 e estão armazenadas em duas bibliotecas do sistema “codbiblio.c” e “decodbiblio.c”. As rotinas geradas juntamente com as bibliotecas específicas e com a tabela de símbolos de cada módulo realizam a codificação ou decodificação dos valores caso a caso.

5.5.1 FUNÇÕES DE CODIFICAÇÃO

Existem tantas funções de codificação quantos forem os tipos básicos do ASN.1 a serem codificados. Estas funções são simples, independentes umas das outras, e funcionam com o auxílio da tabela de símbolos do compilador. Além delas, existem duas funções genéricas que coordenam a codificação e as buscas na tabela. A seguir encontra-se a relação das funções que compõem a biblioteca de codificação:

Funções de codificação de tipos primitivos

codinteger: codifica valores de tipo INTEGER;

codboolean: codifica valores de tipo BOOLEAN;

codreal: codifica valores de tipo REAL;

codbits: codifica valores de tipo BITSTRING

codoctet: codifica valores de tipo OCTETSTRING, e também todos os tipos derivados de OCTETSTRING E OBJEC DESCRIPTOR, identificando-os através da “tag” na sua chamada;

codobject: codifica valores de tipo OBJECT IDENTIFIER;

codnull: codifica valores de tipo NULL;

codexternal: codifica valores EXTERNAL. Esta é a função mais complexa, devido à necessidade de recuperação da tabela de símbolos de um outro módulo já compilado e identificado através de um IMPORT. O acesso a essa tabela é feito para identificar o tipo e codificar seu valor;

codany: codifica valores de tipo ANY como padronização do uso do tipo ANY. O nome do tipo deve ser informado em tempo de codificação, antes da obtenção do valor do tipo. Logo, esta função lê primeiro o nome do tipo, busca-o na tabela de símbolos e em seguida codifica seu valor.

Basicamente, todas funções codificadoras de tipos primitivos operam de forma similar:

- codifica a “tag” do tipo do dado através da função “tag”_implicit. Esta função é encarregada de codificar as “tags” de acordo com a sua condição (implícita ou explícita). Outra atribuição dessa função é participar o número da “tag” caso este seja maior que 30;
- codifica o tamanho do valor através da função **tamanho** que decide se esse valor deve ser particionado;
- codifica o valor de acordo com o seu tipo.

Funções de codificação de tipos construtores

codifcons: codifica tipos SEQUENCE, SEQUENCE OF, SET, SET OF, indistintamente. Cada tipo estruturado deve terminar com a seqüência de caracteres “}}\\}””, que determina o fim dos tipos SEQUENCE OF e SET OF;

codifchoice: codifica tipos CHOICE, procurando o componente a ser codificado e chamando a função codificadora geral;

Funções gerais

codifica: através do nome de um tipo, procura na tabela de símbolos e identifica a “tag” universal deste para, em seguida, chamar a função específica;

coduniversal: chama a função específica de codificação para cada tipo segundo sua “tag” universal, indicada pela função codifica.

Independente dos tipos básicos ASN.1, foram incluídas, nesta biblioteca, quatro funções de codificação para os tipos definidos pelas macros do ROSE. Estes tipos serão codificados como se fossem tipos primitivos que da mesma forma serão tratados pelas funções de uso geral.

codbind: codifica valores que correspondem à macro BIND;

codunbind: codifica valores que correspondem à macro UNBIND;

codoperation: codifica valores da macro OPERATION;

coderror: codifica valores da macro ERROR.

Estas funções assumem, como “tag” universal os números de “tag” especificados na notação das macros do documento ROSE (os números de 29 a 32).

5.5.2 FUNÇÕES DE DECODIFICAÇÃO

A biblioteca de decodificação é composta das mesmas funções que a biblioteca de codificação, para os tipos básicos, possuindo também duas funções gerais de decodificação.

Funções de decodificação de tipos primitivos

decodinteger: decodifica octetos correspondentes a INTEGER;

decodbool: decodifica octetos correspondentes a BOOLEAN;

decodreal: decodifica octetos de tipo REAL;

decodbits: decodifica octetos de tipo BITSTRING;

decodoctet: decodifica octetos de tipos OCTETSTRING;

decodobject: decodifica tipos OBJECT IDENTIFIER;

decodnull: decodifica tipos NULL;

decodext: decodifica tipos EXTERNAL.

Funções de decodificação de tipos construtores

decodcons: decodifica os tipos construtores, indistintamente, à medida que percorre os componentes na tabela de símbolos, confrontando-os com a “tag” do octeto de identificação;

decodchoice: identifica qual componente do tipo CHOICE foi codificado a partir da “tag” de identificação e chama a rotina específica de decodificação.

As funções de decodificação de tipos primitivos utilizam as funções `decod_ident`, `decod_tag`, `compor_tag`, para recuperar as classes de “tag”, assim como, seus números e tamanho do valor codificado como guia na decodificação dos valores.

Funções de decodificadores gerais

decodifica: através de um nome de um tipo busca na tabela de símbolos a “tag” universal do tipo a decodificar;

decoduniversal: chama a função específica de decodificação para cada tipo, segundo sua “tag” universal, indicada pela função codifica.

Da mesma forma que na biblioteca de funções de codificação, existem quatro funções de decodificação das macros do ROSE. São elas: decodbind, decodunbind, decodoperation, decoderror.

Todas as funções que decodificam tipos básicos, obtêm seus dados do arquivo de octetos (resultado da codificação), através da leitura simples de um “byte” por vez. Este arquivo é utilizado como interface entre o compilador e seu usuário – neste caso, a camada de apresentação. O resultado é armazenado no arquivo “decodnome_do_modulo.c”. Estes arquivos são considerados a entrada e saída padrão do sistema. No futuro serão as entradas e saídas “default”, deixando ao usuário a responsabilidade de fornecer suas próprias rotinas de leitura e gravação dos dados, de modo a adaptar a parte dinâmica do sistema às suas necessidades.

Capítulo 6

Conclusões

As principais conclusões desta dissertação e o desenvolvimento de trabalhos futuros na área são discutidos neste capítulo.

6.1 CONCLUSÕES

Como foi dito na introdução deste trabalho, transformar manualmente as especificações das PDU's em estruturas C, torna-se um trabalho mais difícil à medida que as PDU's crescem em complexidade; logo, justifica-se a automatização deste processo. O compilador ASN.1 que foi implementado realiza automaticamente esta tarefa, para qualquer conjunto de PDU's, de qualquer protocolo de aplicação, contribuindo para a padronização das estruturas de dados do ambiente no qual futuramente atuará (Projeto SISDI-OSI). O fato de ele estar programado utilizando as ferramentas YACC e LEX facilita futuras alterações, modificações e inclusões de novas regras gramaticais que, eventualmente, possam aparecer em novos adendos da norma ou através de trabalhos futuros, por incremento do conjunto de gramáticas que serão reconhecidas (Macros).

Uma outra contribuição desta ferramenta é o serviço que presta à camada de apresentação, com a geração automática das rotinas de codificação e decodificação que serão utilizadas pela camada no momento de realizar as conversões correspondentes, tornando-a independente do tipo dos dados e das estruturas dos mesmos. Logo, se uma estrutura PDU é modificada, ao ser compilada novamente novas rotinas serão geradas.

6.2 TRABALHOS FUTUROS

Existem vários trabalhos futuros a serem desenvolvidos como continuação desta dissertação e eles se dividem basicamente em dois grupos: extensão da gramática ASN.1 compilada e integração com o projeto SISDI-OSI:

- ✓ **Extensão da gramática ASN.1.** Existem alguns tipos que não foram incluídos neste compilador, como por exemplo, o tipo *“real”*, em toda sua extensão devido à falta do adendo correspondente à codificação deste tipo. Futuramente todos os tipos *“characterstring”* serão também considerados.

Como foi dito nos capítulos anteriores, este compilador não trata das macros em forma genérica, compilando-se as macros do protocolo ROSE e assumindo que elas fazem parte do conjunto de tipos definidos ASN.1. Desta forma, é interessante destacar que a inclusão da gramática das macros leva a aplicar outras técnicas de compilação.

- ✓ **Integração com o Projeto SISDI-OSI.** O ambiente contemplado para a atuação deste compilador é o ambiente do projeto SISDI-OSI que está sendo desenvolvido em paralelo com este trabalho, não sendo ainda possível contar com a camada de apresentação e os protocolos de aplicação. Os pontos a seguir farão parte do projeto e estão atualmente em fase de desenvolvimento:

- Interface de gerenciamento da fase estática da compilação;
 - Definição de rotinas padrões de entrada/saída para os dados que serão utilizados pelos codificadores e os dados entregues pelos decodificadores;
-
- ✓ **Configuração do Compilador.** Cita-se a título de exemplo, uma extensão para Pascal, uma vez que alguns dos protocolos de aplicação como CCR (Silva, 1992) estão sendo implementados nesta linguagem;
 - ✓ **Integração com a camada de apresentação** visando definir o gerenciamento e o controle das funções de codificação e decodificação.

Referências Bibliográficas

- Aho A.V., Sethi R. e Ullman J.D.** Compiler Principles, Techniques and Tools [Livro]. - [s.l.] : Addison-Wesley, 1988.
- Blake W.** Data Network Concepts Theory and Practice, Prentice Hall, Englewood Cliffs, New Jersey, 1989. [Livro]. - Englewood Cliffs, New Jersey : Prentice Hall, 1989.
- Bochman D. e Quinet G.,V.** OSI Specifications using ASN.1 and other formal description techniques [Conferência] // 9º Simpósio Brasileiro de Redes de Computadores. - Florianópolis, Santa Catarina : [s.n.], 1991.
- Halshall F.** Data Communications, Computer Networks and OSI [Book]. - [s.l.] : Addison-Wesley, 1988.
- ISO** Information processing system - Open System Interconnection - Specification Of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1) [Report]. - [s.l.] : International Standard 885, 1987b.
- ISO** Information Processing System - Open System Interconnection - Specification of Abstract Syntax Notation One (ASN.1) [Relatório]. - [s.l.] : International Standard 8825, 1987a.
- ISO** Information processing System - Text Communications Remote Operations - Part 1, Model, Notation and Service definitions [Relatório]. - [s.l.] : International Standard 9072-1, 1989.
- ISO** Information processing systems- Open System Interconnection - Abstract Syntax Notation One (ASN.1). Extensions to ASN.1 [Relatório]. - [s.l.] : Draft Addendum 8824/DAD1, 1987c.
- Kernigham B.W. e Ritchie D.M.** The C Programming Language [Livro]. - Englewood Cliffs, New Jersey : Prentice Hall, 1978.
- Mendes M. de J.** Comunicação Fabril e o Projeto Map/TOP [Livro]. - Buenos Aires, Argentina : Kapelusz, 1989.
- Rose M. T.** The Open Book, A Practical Perspective on OSI [Livro]. - Englewood Cliffs, New Jersey : Prentice Hall, 1988.
- Silva F.** CCR // Dissertação de Mestrado. - Campinas : IMECC, UNICAMP, 1992.
- Tannenbaum A. S.** Computer Networks [Livro]. - Englewood Cliffs, New Jersey : Prentice Hall, 1988.
- White J. E.** ASN. and ROS: The Impact of X400 on OSI [Artigo] // IEEE Journal on Selected Areas in Communication. - [s.l.] : IEEE, 1989. - September. - Vol. 7.

Apêndice A

Regras Gramaticais ASN.1

É descrita neste apêndice a gramática utilizada pelo primeiro e segundo passos do compilador. Essa gramática foi utilizada para gerar ambos “parsers” por meio do YACC e se encontra em uma forma similar à notação BNF, sendo a mesma notação utilizada para o YACC: o carácter “:” significa atribuição e “|” corresponde a mais de uma opção, as palavras escritas em letras maiúsculas são símbolos terminais e as demais correspondem aos nomes das regras.

```
moduledefinitions: TYPeref DEFINITIONS ASIGNACAO
                  BEGIN
                  modulebody
                  END
                  ;
```

```
Modulebody : /* vazio */
            | assignmentlist
            ;
```

```
assignmentlist: assignment
               | assignmentlist
                 assignment
               ;
```

```
assignment: typeassignment
           | valueassignment
           ;
```

```
typeassignment : TYPeref ASIGNacao
               type
               ;
```

```
type: builtintype
     | definedtype
     ;
```

```
builtintype : booleantype
            | integertype
            | bitstringtype
            | octetstringtype
            | anytype
            | nulltype
            | sequencetype
            | settype
            | sequenceoftype
            | setoftype
            | choicetype
            | selectiontype
            | taggedtype
            | objectidentifiertype
            ;
```


booleantype: BOOLEAN
;

integertype : INTEGER
| INTEGER CHAVEABERTA
namednumberlist
CHAVEFECHADA
;

namednumberlist: namednumber
| namednumberlist
VIRGULA
namednumber
;

namednumber: IDENT
PARENTABERTO
signednumber
PARENTFECHADO
;

signednumber: NUMERO
| MENOS NUMERO
;

bitstringtype: BITSTRING
| BITSTRING CHAVEABERTA
namedbitlist
CHAVEFECHADA
;

```
namebitlist : namedbit
             | namedbitlist
             VIRGULA
             namedbit
             ;

namedbit    : IDENT
             PARENTABERTO
             NUMERO
             PARENTFECHADO
             ;

octetstringtype: OCTETSTRING
                ;

nulltype      : NULL
                ;

sequencetype: SEQUENCE
             CHAVEABERTA
             elementtypelist
             CHAVEFECHADA
             ;

elementtypelist: elementtype
                | elementtypelist
                VIRGULA
                elementtype
                ;
```

```
elementtype: namedtype
            | namedtype
              OPTIONAL
              | COMPONENTS OF
                type
            ;
```

```
namedtype : IDENT type
          | type
          ;
```

```
sequenceoftype: SEQUENCE OF
                type
                | SEQUENCE
                ;
```

```
settype: SET
        CHAVEABERTA
        elementtypelist
        CHAVEFECHADA
        ;
```

```
setoftype: SET OF
          type
          | SET
          ;
```

```
choicetype: CHOICE
           CHAVEABERTA
           alternativetypelist
           CHAVEFECHADA
           ;
```

```
alternativetypelist: nametype  
    | alternativetypelist  
    VIRGULA  
    namedtype  
    ;
```

```
selectiontype: IDENT MENOR  
    type  
    ;
```

```
taggedtype: tag type  
    | tag IMPLICIT  
    type  
    ;
```

```
tag: CUADROABERTO  
    class  
    classnumber  
    CUADROFECHADO  
    ;
```

```
class: /* vazio */  
    | UNIVERSAL  
    | APPLICATION  
    | PRIVATE  
    ;
```

```
classnumber: NUMERO  
    ;
```

anytype: ANY
;

objectidentifiertype: OBJECT
IDENTIFIER
;

definedtype: TYPeref
;

Apêndice B

Um Exemplo Completo

Descrição do Exemplo

Na camada de aplicação, o ACSE – *Association Control Service Element* –, é o encarregado do estabelecimento e a finalização de uma associação lógica entre dois SASE's – *Specific Application Service Element* –. Uma associação é estabelecida em resposta ao requerimento de um usuário (AP) para um serviço de aplicação, que cria uma PDU de iniciação de associação (*connect-request*), para utilizar os serviços do ACSE.

O ACSE gera a PDU AARQPDU (*Associate Request PDU*) a partir de informações da primeira AAR.

Os módulos que contêm as PDU's envolvidas nesta situação serão utilizados para ilustrar o exemplo de compilação de uma especificação ASN.1 e a codificação correspondente.

Será compilado um módulo que contém parte das PDU's do ACSE [SIL92] e utilizar-se-á como protocolo que solicita a associação o TP. A seguir os módulos compilados:

Módulo ASN.1 do ACSE

```
ISO-2-ACSE1  DEFINITIONS ::=
BEGIN

AARQapdu ::= [APPLICATIONS 0] IMPLICIT SEQUENCE
{ protocolVersion          [0] IMPLICIT BISTRING
  { Version (0) },
calledAETitle              [1] ApplicationTitle,
  OPTIONAL,
callingAETitle             [2] ApplicationTitle,
  OPTIONAL,
applicationContexName      [3] ApplicationContexName,
userInformation            [4] AssociationData
  OPTIONAL,
}

ApplicationTitle           ::= OBJECT IDENTIFIER

ApplicationContexName      ::= OBJECT IDENTIFIER

AssociationData            ::= EXTERNAL

END
```


Módulo ASN.1 do TP

```
ISO-TP MODULE DEFINITIONS ::=
BEGIN
TP-ASSOCIATION-ESTABLISHMENT-RI ::= SEQUENCE
{
  protocol-Version          [1] Protocol-Version
                           { version (1) },
  conection-winner-assignment [2] BOOLEAN,
                           DEFAULT TRUE,
  bid-mandatory             [3] BOOLEAN,
                           DEFAULT TRUE,
  recovery-contex-handle    [4] Recovery-Contex-Handle,
                           OPTIONAL,
}

Protocol-Versions          ::= BITSTRING

Recovery-Contex-Handle     ::= OCTETSTRING

END
```

Valores de cada Módulo

A seguir são descritos alguns dos valores que as PDU's podem ter para efeito de codificação.

Módulo ISO8650-1-ACSE1

protocolVersion: considera-se o valor 1, já que não se precisa de informação da implementação, devido ao fato de o ACSE ser único.

calledACTitle: considera-se o OBJECT IDENTIFIER do RDA

{ ISO STANDARD RDA(9579) part2-sql(2) Tp-ac(4) version (1) }

calledAETitle: neste caso, como é um campo opcional, a título de exemplo não terá valor.

applicationContextName: é um inteiro que identifica um contexto de aplicação, e terá como valor o número 23.

userInformation: a título de exemplo EXTERNAL, na sua opção any. No momento da codificação deverá ser informado que o tipo correspondente é a PDU do TP e serão os valores dessa PDU os codificados.

Valores da PDU do PT

protocol-version: será considerado o valor 0.

connection-winner-assignment: seu valor será o “default”

bid-mandatory: terá o valor FALSE

recovery-context-handle: se refere ao nome de uma função, como exemplo “recovery”.

Arquivos de definições de tipos C gerados pela compilação

Arquivo correspondente ao módulo ISO-2-ACSE1 : ISO_2_ACSE1tiposc.c

```
“include tiposan.c”
typedef external AssociationData;
typedef objectident ApplicationContextName;
typedef objectident ApplicationTitle;
typedef struct AARQapdu
{
    bitstring protocolVersion;
    int calledAETitleop;
    ApplicationTitle calledAETitle;
    int callingAETitleop;
    ApplicationTitle callingAETitle;
    ApplicationContextName applicationContextName;
    int userInformationop;
    AssociationData userInformation;
} AARQapdu;
```

Arquivo correspondente ao módulo ISO-TP : ISO_TPtiposc.c

```
"include  tiposan.c"
typedef  octetstring Recovery_Context_Handle;
typedef  objectstring Protocol_versions;
typedef  struct TP_ASSOCIATION_RI
{
    Protocol_Versions protocol_version;
    int conection_winner_assignmentdef;
    BOOLEAN conection_winner_assignment;
    int bid_mandatorydef;
    BOOLEAN bid_mandatory;
    int recovery_context_handleop;
    Recovery_Context_Handle recovery_context_handle;
} TP_ASSOCIATION_ESTABLISHMENT_RI;
```

Funções de Codificação e Decodificação

Função de codificação do Módulo ISO-2-ACSE1:

```
include <stdio.h>
include "tiposan.c"
include "tabela.c"
include "codibiblio.c"
main( )
{ abre_arquivo("dados.c");
  abre_arquivotab("ISO_2_ACSE1tab.c");
  abre_arquivocod( );
  RECUPERA( );
  codifica(AARQapdu);
}
```

Função de decodificação

```
include <stdio.h>
include "tiposan.c"
include "tabela.c"
include "decodibiblio.c"
main( )
{ abre_arquivo("codber.c");
  abre_arquivotab("ISO_2_ACSE1tab.c");
  abre_arquivodecod( );
  RECUPERA( );
  decodifica(AARQapdu);
}
```

As funções de codificação do módulo ISO-TP não serão utilizadas uma vez que sua PDU será codificada através do campo "userInformation".

Arquivo “dados.c” onde se encontram os valores a serem codificados

1	1	6	1	0
	9579		2	4
	1			
0	23	1	1	
	“TP”_ASSOCIATION_ESTABLISHMENT_RI”			
0	0	1	0	1
	“recovery”			

Arquivo “codber.c” resultado da codificação em Hexadecimal.

80				
80	02	07	80	
81	06	06	28	EA
	6B	02	04	01
83	02	01	23	
84	30	80		
02	01	01		
80	30	80		
81	03	02	07	00
83	01	01	00	
84	04	06	72	65
	63	76	65	72
	79			
00	00	00	00	00
	00			