# *Introdução*

ASN.1 é a sigla para **Abstract Syntax Notation One**, uma linguagem de alto nível criada única e exclusivamente para descrever estruturas de informações. O sufixo "1" indica que esta é a primeira notação padronizada. Esta é largamente utilizada na especificação de protocolos, em particular para a camada de aplicação da **OSI (Open System Interconnection)**. A notação ASN.1 é descrita internacionalmente pela ISO8824 e as regras para a codificação de estruturas de dados da ASN.1 em fluxo de bits para transmissão é dado pela ISO8825.

Com a ASN.1 o projetista de um protocolo não precisa se preocupar com a forma com que os dados serão transmitidos, e sim com a forma que deve descrever as estruturas de dados, sendo que esta facilita a definição de estruturas de dados bastante complexas.

A pergunta que surge naturalmente ao iniciarmos o estudo da ASN.1 é o que vem a ser uma sintaxe abstrata.

Para melhor compreender a que isso se refere, imaginemos um registro de um banco em que devemos ter como informações o número da conta, o tipo de conta (conta corrente ou poupança), a agência, o saldo, o nome e o CPF do cliente. Vamos considerar os dados organizados conforme a tabela abaixo:

| **Informação** | **Forma de Representação** |
|---|---|
| Nome do cliiente | String com um máximo de 30 caracteres |
| CPF | String com 13 caracteres, com dígitos e "-" |
| Número da conta | String com 10 caracteres, com dígitos e "-" |
| Tipo da conta | Pode assumir os valores conta corrente ou poupança |
| Agência | Dois campos: o nome(string) e o número(inteiro) |
| Saldo | Um valor representado por um número real |

A representação dos dados dessa forma, já seria por si mesma uma sintaxe abstrata, porque não temos nenhuma representação específica, e sim uma maneira de representar as informações a respeito da conta.

Vamos representar essas informações em tipos ASN.1(definidos na ISO8824) para mostrar como as definições da ASN.1 são por si só uma sintaxe abstrata. Poderíamos definir a estrutura **Registro** abaixo:

```
TipoDígitos  ::= PrintableString
                 ("0" | "1" | "2" | "3" | "4" | "5"
                  | "6" | "7" | "8" | "9" | "-")

TipoAgência  ::= SET
    {
    NomeAgência     ::= PrintableString (SIZE(10)),
    NumeroDaAgência ::= INTEGER (0..9999)
    }

Registro ::= SET
    {
    Nome           ::= PrintableString (SIZE(30)),
    CPF            ::= TipoDígitos (SIZE(13)),
    NumeroDaConta  ::= TipoDígitos (SIZE(10)),
    TipoDaConta    ::= ENUMERATED {contacorrente(1), poupança(2)},
    Agência        ::= TipoAgência,
    Saldo          ::= REAL
    }
```

Ao analisar a ASN.1 surge também o conceito de sintaxe de transferência. Esse conceito está ligado a transmissão dos dados. De posse de um tipo ASN.1 definido (como no exemplo acima) devemos atribuir valores para que sejam conduzidos de um computador a outro. Definidos os valores, estes são codificados(seguindo as normas da ISO8825) para então serem transmitidos. A forma como esses dados são transmitidos é a sintaxe de transferência. Em ASN.1 essa codificação segue regras conhecidas como **Basic Encoding Rules (BER - Regras básicas de codificação)** que nada mais fazem que representar tipos e valores por octetos, como são chamados os bytes pela **OSI**.

Um valor a ser atribuído poderia ser o da cliente Cristina abaixo:

```
cristina Registro ::=
   {
   "Cristina Resende" ,
   "6666666-66",
   "88888-8",
   poupança,
   {"Laranjeiras",0232},
   {2918189707,2,0}
   }
```

E a sintaxe de transferência seria:

```
11 45₁₆
   14 10 4372697374696E6120526573656E6465₁₆
   14 0A 363636363636362D3636₁₆
   14 08  38383838382D38₁₆
   0A 01 02₁₆
   11 10₁₆
      14 0B 4C6172616E6A6569726173₁₆
      02 01 E8₁₆
   09 06 8000ADF00A8B₁₆
```

# *1 Princípios da ASN.1*

Ao longo desta seção introduziremos os formatos básicos que envolvem a ASN.1, tais como os tipos, os valores, os sub-tipos e os módulos. Faremos aqui apenas uma breve introdução de como essas sintaxes estão estruturadas, visando dar uma visão geral da ASN.1, para somente na seção seguinte descrever mais detalhademente os vários tipos e subtipos existentes.

- Tipos e Valores
- Subtipos
- Módulos

# *1.1 Tipos e Valores*

Em se tratando de uma linguagem feita exclusivamente para definir uma sintaxe abstrata, ou seja, sendo usada para descrever a estrutura e o conteúdo de uma mensagem, é fácil perceber a importância da definição de tipos para a ASN.1.

Os tipos podem ser simp les ou estruturads. Os simples são as estruturas básicas presentes na linguagem, como o tipo **INTEGER** e **BOOLEAN**, enquanto que os tipos estruturados são definidos em termos de outros tipos, sendo que estes podem ser tipos simples ou até mesmo um outro tipo estruturado, formando estruturas aninhadas, muito comuns em linguagens de programação como C e Pascal.

Os tipos estruturados são de grande importância para a ASN.1 visto que, sendo uma forma de respresentação abstrata de uma informação, a maior parte dos dados acabarão sendo representados por este tipo de estrutura.

Na definição de um tipo temos basicamente três componentes: um nome que irá representar o novo tipo; o símbolo "::=", que pode ser lido como "é definido como"; e o fomato do tipo em si. Para ilustrar o que já foi dito até o momento, observemos o tipo abaixo, criado para transmitir informações sobre um determinado país:

```
Pais ::= SEQUENCE
{
     nomePaís            PrintableString (SIZE(15)),
     idiomaPaís          PrintableString (SIZE(10)),
     populaçãoPaís       INTEGER
}
```

Assim temos um tipo estruturado sendo formado por tipos simples próprios da ASN.1.

É interessante ressaltar que esses tipos simples se mostram bastante flexíveis de forma a facilitar o trabalho de abstração dos dados. Observando o formato usado para os campos nome e idioma, notamos que estes têm seu tamanho limitado em 15 e 10 caracteres respectivamente, sendo que isto é feito com o identificador **SIZE**.

Não é necessário se preocupar com tais formas de definição no momento, já que estas serão explicadas em ocasião oportuna, e só foram mencionadas para demonstrar a versatilidade e formato geral da definição de tipos na linguagem.

Podemos agora mostrar como são atribuídos valores para os tipos, e já enterdermos de que forma são enviadas as informações.

A atribuição de valores é bastante similar a uma definição de tipo, sendo que a única diferença está em um componente adicional na sintaxe, que aparece entre o nome usado para o valor e o símbolo "**::=**", que é o próprio tipo do valor.

Para melhor entender como isso é feito, observemos a atribuição de valores ao tipo definido acima:

```
brasil País ::=
{
     nomePaís            "Brasil",
     idiomaPaís          "Português",
     populaçãoPaís       150000000
}
```

Com isso definimos um valor do tipo **País** chamado **brasil**.

A definição de tipos e valores é praticamente a única coisa que usuários farão na ASN.1, sendo que a definição de tipos é predominante. Isto ocorre porque uma sintaxe abstrata é por ela mesma organizada em tipos. Os valores são

usados apenas em exemplos e alguns poucos valores assumidos como *default*. É fácil observar que na abstração de dados o tipo inteiro é muito mais importante que um valor particular como 13. Em compensação, quando da transmissão dos dados, os valores é que terão importância relevante, pois afinal somente eles que serão transmitidos. Os tipos são necessários para que os valores sejam decodificados de acordo com o formato desejado.

Os tipos definidos no protocolo estarão presentes no receptor, no transmissor, e na mensagem transmitida, definindo assim a importãncia de valores e tipos. Assim um valor é codificado segundo seu tipo (por regras de codificação próprias da ASN.1). Ao ser recebida, a mensagem é referenciada ao seu tipo e o valor é recuperado. Essa referência é facilmente aplicada já que de acordo com o formato de dados a ser enviado, o receptor sabe a ordem em que estes foram enviados.

Outro fato importante, e que é comum em linguagens de programação, está ligado às regras que delimitam a forma como os nomes de tipos e valores devem ser criados. Como em C e Pascal, exitem regras para que não se criem nomes ilegais.

O primeiro fato a ser observado na ASN.1 é que letras maiúsculas e minúsculas são interpretadas diferentemente. Assim o tipo **Nome** e o tipo **nome** são considerados distintos pela linguagem. Vale ressaltar também que não podem ser usadas palavras-chave da linguagem como nomes para variáveis, tipos e identificadores. Abaixo temos uma lista das palavras reservadas da ASN.1:

| | | | |
|---|---|---|---|
| **ABSENT** | **ANY** | **APPLICATION** | **BEGIN** |
| **BIT** | **BOOLEAN** | **BY** | **CHOICE** |
| **COMPONENT** | **COMPONENTS** | **DEFAULT** | **DEFINED** |
| **DEFINITIONS** | **empty** | **END** | **ENUMERATED** |
| **EXPLICIT** | **EXPORTS** | **EXTERNAL** | **FALSE** |
| **FROM** | **IA5String** | **identifier** | **IDENTIFIER** |
| **GeneralizedTime** | **GeneralStrinG** | **GraphicString** | **IMPLICIT** |
| **IMPORTS** | **INCLUDES** | **INTEGER** | **ISO646String** |
| **MACRO** | **MAX** | **MIN** | **MINUS-INFINITY** |
| **NOTATION** | **NULL** | **NumericString** | **number** |
| **OBJECT** | **ObjectDescriptor** | **OCTET** | **OF** |
| **OPTIONAL** | **PLUS-INFINITY** | **PRESENT** | **PrintableString** |
| **PRIVATE** | **REAL** | **SEQUENCE** | **SET** |
| **SIZE** | **string** | **STRING** | **TAGS** |
| **TeletexString** | **TRUE** | **type** | **TYPE** |
| **T61String** | **value** | **VALUE** | **VideotexString** |
| **VisibleString** | **UNIVERSAL** | **UTCTime** | **WITH** |

Os caracteres que podem ser usados nos nomes são as letras maiúsculas, as letras minúsculas, os dígitos decimais e o hífen**("-")**.

Além destas, devemos observar algumas outras regras:

**1**- O primeiro caracter de um nome deve ser uma letra;
**2**- O último caracter nunca poderá ser um hífen;
**3**- Um hífen não pode ser seguido por um outro hífen;

Seriam então exemplos de nomes ilegais os mostrados abaixo:

```
1560
1TipoDado
TipoDado-
Controle--de--acesso
OBJECT
```

Sendo que o último é uma palavra reservada.

O caso em que são usados dois hífens **("--")** é uma situação especial da ASN.1, pois eles definem o formato de comentário, ou seja, tem um significado análogo as chaves ("**{**") em Pascal ou a combinação"**/*"** em C, como abaixo:

```
brasil País ::=
{
  nomePaís        "Brasil",     --campo para  nome  do País--
  idiomaPaís      "Português",  --campo para idioma do País--
  populaçãoPaís   150000000     --tamanho da população--
}
```

Também temos que os tamanhos dos nomes não são limitados, no entanto para facilitar a depuração é comum seguirem-se algumas regras na criação dos mesmos. Observemos os exemplos abaixo:

```
tipo            : TipoDado
valor           : tipoValor
identificadores : segunda,terca,quarta,
```

Assim, não se utilizam nomes excessivamente grandes; os nomes de tipos começam com letras maiúsculas, os nomes de valores com letras minúsculas e os identificadores aparecem sempre em letras minúsculas. Procura-se também usar nomes que tornem claro o motivo da criação do tipo ou do valor.

Visto que as palavras-chave na ASN.1 geralmente são formadas por letras maiúsculas, o usuário poderia facilmente criar nomes idênticos com letras minúsculas substituindo algumas das maiúsculas. No entanto isso também é evitado, para que a depuração seja facilitada.

# *1.2 Subtipos*

A partir dos tipos simples existentes ou até mesmo de outros estruturados, podemos definir outros que são um subconjunto do original. Imaginemos que precisamos criar um tipo para representar o campo dos minutos em de forma estruturada que represente a hora de determinada ocorrência (com minutos e segundos). Poderíamos definir o tipo **Minutos** conforme mostrado abaixo:

```
Minutos ::= INTEGER(0..59)
```

Assim o tipo **Minutos** é um sub-tipo do tipo **INTEGER**, com uma faixa limitada de valores.

Consideremos agora a estrutura mostrada abaixo:

```
FimDeSemana ::= DiasDaSemana (sabado | domingo)
```

Onde o símbolo "|"pode ser pronunciado como "**ou**". Neste caso definimos quais os valores dos dias da semana que são também valores do fim de semana.

Usando a palavra chave **INCLUDES**, podemos utilizar a estrutura acima, incluindo também um tipo. Com isso,

poderíamos criar o tipo **FimDeSemanaProlongado** com a forma mostrada abaixo:

```
FimDeSemanaProlongado ::= DiasDaSemana
                    (INCLUDES FimDeSemana | segunda)
```

Podemos também criar um sub-tipo limitando as letras do alfabeto que podem ser utilizadas para representar os dados. Uma estrutura com essas características teria o formato abaixo:

```
Vogais ::= PrintableString (FROM("a" | "e" | "i" | "o" | "u" |
                                 "A" | "E" | "I" | "O" | "U"))
```

Onde a palavra-chave **FROM** tem essa característica especial de permitir escolher um grupo de caracteres dentro de um tipo existente.

Um sub-tipo que seria formado por uma cadeia de caracteres de tamanho limitado usando a palavra-chave **SIZE** seria:

```
PlacaDeCarro ::= PrintableString (SIZE(7))
```

Onde o tipo **PlacaDeCarro** é formado por 7 caracteres.

# *1.3 Módulos*

Um módulo é uma coleção de tipos, valores e macros (que serão explicadas mais a frente). Um módulo agrupa um conjunto de definições tais como as usadas para definir uma sintaxe abstrata. Contudo as definições dos módulos estão inteiramente nas mãos dos usuários da ASN.1, que podem colocar todas as suas definições dentro de um único módulo ou em vários deles.

O módulo seria a forma encontrada por um projetista na ASN.1 para organizar as definições por ele criadas. Uma característica que isto vem a criar é que módulos criados por um usuário podem ser facilmente utilizados por um outro. Esta característica dispensaria o usuário do trabalho de criar grupos de tipos que já estão definidos, bastando-lhe apenas acessar uma biblioteca de módulos.

Dentro de um módulo as definições podem ser colocadas em qualquer ordem, não sendo necessária a preocupação de definir os tipos antes que eles sejam usados. Pode-se então definir uma estrutura 1 que tem em seu interior uma estrutura 2 e somente depois declarar a estrutura 2. Linguagens de programação como C e Pascal não possuem este tipo de característica, sendo o uso de tipos e estruturas limitados pelos previamente declarados.

Na definição de um módulo temos os seguintes componentes: um identificador do módulo; a palavra-chave **DEFINITIONS**; opcionalmente um **tag** (que será oportunamente citado); o símbolo "**::=**"; e o corpo do módulo, sendo este delimitado pelas palavras-chave **BEGIN** e **END**.

O exemplo abaixo ilustra esse tipo de definição:

```
MóduloInformaçõesWolkswagen {4 8 10 34 12} DEFINITIONS ::=

BEGIN
       Logus   InformaçõesDoCarro ::= {...}
       InformaçõesDoCarro         ::= SEQUENCE {...}
       Gol    InformaçõesDoCarro  ::= {...}
END
```

O módulo acima tem em seu interior dois valores e um tipo. Vale notar a declaração do valor **Logus** antes da definição do tipo do valor.

A regra para criar nomes de módulos é a mesma dos tipos: as primeiras letras das palavras que formam o nome em letras maiúsculas.

Como já foi citado é comum que módulos façam uso de tipos ou valores declarados em outros módulos, sendo esta referência entre módulos característica da ASN.1. No entanto o usuário deve indicar dentro do módulo quais as definições que podem ser exportadas e quais são importadas de outros módulos.

Essa indicação é feita através das palavras-chave **EXPORTS** e **IMPORTS**. Por meio da palavra-chave **EXPORTS** pode-se definir quais os tipos e/ou valores que podem ser exportados dentre todos os componentes do módulo, sendo que se esta não for incluída na definição do módulo, qualquer uma das definições poderá ser importada por outro módulo.

Considerando o exemplo criado quando da especificação de módulos, poderíamos querer que somente o tipo **InformaçõesDoCarro** pudesse ser importado. Para tanto, colocaríamos a linha abaixo logo após o **BEGIN**:

```
EXPORTS Informações;
```

Com isso teríamos o módulo definido como abaixo:

```
MóduloInformaçõesWolkswagen {4 8 10 34 12} DEFINITIONS ::=

BEGIN

EXPORTS InformaçõesDoCarro;


       Logus   InformaçõesDoCarro ::= {...}
       InformaçõesDoCarro         ::= SEQUENCE {...}
       Gol     InformaçõesDoCarro ::= {...}
END
```

Se for desejável que mais de um tipo e/ou valor possa ser importado por outro módulo, estes aparecem separados por vírgulas. No caso de não se desejar que nenhuma das descrições do módulo seja acessível, podemos incluir a linha abaixo:

```
EXPORTS ;
```

Para que um módulo faça uso de uma definição de um outro módulo, se esta estiver declarada para tal uso, devemos usar a palavra **IMPORTS** na mesma posição da palavra **EXPORTS** em conjunto com a palavra chave **FROM**. Assim a linha teria o formato abaixo:

```
IMPORTS [tipo a ser importado]  FROM  [módulo com tag],
        ...,
        [outro tipo]  FROM [módulo deste tipo com tag];
```

Exemplificando temos o módulo abaixo:

```
MóduloInformaçõesFord {4 8 10 34 12} DEFINITIONS ::=

BEGIN
```

```
IMPORTS InformaçõesDoCarro FROM
        MóduloInformaçõesWolkswagen{4 8 10 34 12};

        Escort  InformaçõesDoCarro ::= {...}
        Verona  InformaçõesDoCarro ::= {...}

END
```

Caso seja necessária uma referência externa a uma das definições de um módulo, podemos usar a forma que aparece no exmplo abaixo:

```
uno MóduloInformaçõesWolkswagen.InformaçõesDoCarro ::= {..}
```

Que define o valor **uno** do tipo **InformaçõesDoCarro** definido dentro do módulo **MóduloInformaçõesWolkswagen.**

# *2 Tipos da ASN.1*

Podemos dividir os tipos da ASN.1 em três grupos:

- Tipos Simples
- Tipos Estruturados
- Useful Types

# *2.1 Tipos Simples*

Os tipos simples são as construções básicas da ASN.1, e são usados para representar desde valores numéricos a grupos de caracteres.

Abaixo aparecem os tipos de dados simples:

- Integer
- Real
- Null
- Boolean
- Enumerated
- Bit String
- Octet String
- Character Strings
- Object Identifier

# *2.1.1 Integer*

Este tipo é utilizado para representar números inteiros e é semelhante ao tipo *int* da linguagem C e ao tipo *integer* da linguagem Pascal, sendo, no entanto, bem mais flexível, pois permite a utilização de certos formatos que facilitam muito o trabalho dos programadores.

Um primeiro fato interessante de ser observado é que não existem valores limites para este tipo, como em seus correspondentes nas linguagens C e Pascal, abrangendo também valores negativos. Todavia, sub-tipos que limitam as faixas dos valores podem ser facilmente criados, sendo esta uma prática recomendada. Como a faixa de valores de uma variável em casos reais será quase sempre limitada, isso torna-se facilmente aplicável como uma boa regra na ASN.1.

Podemos também incluir um grupo de identificadores para cada um dos valores como no exemplo abaixo:

```
Inteiro  ::= INTEGER

        {
            um(1), dois(2), três(3), quatro(4), cinco(5),
            nove(9), oito(8), sete(7), seis(6), zero(0),cem(100),
            mil(1000)
        }
```

Assim teríamos um novo tipo que incluiria todos os inteiros sendo alguns destes representados por identificadores. Vale ressaltar que embora somente tenham sido criados identiicadores para alguns dos valores, este tipo ainda inclui todos os inteiros, não formando portanto um subtipo.

Outro característica que pode ser observada no exemplo acima é que este tipo de atribuição não exige que os identificadores apareçam em ordem numérica. Para limitarmos os valores que farão parte do novo tipo, podemos definir um sub-tipo como na estrutura abaixo:

```
Numeros ::= INTEGER

      {
              um(1), dois(2), três(3), quatro(4), cinco(5),
              seis(6), sete(7), oito(8), nove(9), zero(0)
      }
      (0..9)
```

que pode também ser definida como na forma abaixo:

```
Numeros ::= INTEGER

      {
              um(1), dois(2), três(3), quatro(4), cinco(5),
              seis(6), sete(7), oito(8), nove(9), zero(0)
      }
      (zero..nove)
```

Sendo que o segundo caso se aproveita dos próprios identificadores que criou. Neste tipo de definição o sub-tipo criado só pode assumir valores entre 0 e 9.

Criando esses identificadores poderíamos usar definições como as que aparecem abaixo para os valores:

```
Valor Numero ::= dois
Valor Número ::= 2
```

Podemos também usar o caracter "<" ou as palavras chave **MAX** e **MIN** para criar sub-tipos. Os exeplos abaixo ilustram como:

```
InteirosPositivos   ::= INTEGER (0
```

# *2.1.2 Real*

Os números reais em ASN.1 tem uma sintaxe especial quando da definição de valores, sendo diferente do formato das linguagens como C e Pascal. Eles aparecem no formato abaixo:

$$MxB^{E}$$

onde **M** é a mantissa, **B** a base (2 ou 10) e **E** o expoente.
Para expressar um valor, devemos fazê-lo da forma abaixo:

```
valor REAL ::= {M, B, E}
```

ou seja, para atribuir o valor 2.564 teríamos que fazê-lo como em seguida:

```
valor REAL ::= {2564, 10, -3}
```

Estão incluídos também os valores especiais "mais infinito" e "menos infinito" representados pelas palavras-chave **PLUS-INFINITY** e **MINUS-INFINITY** respectivamente. Isso pode ser observado nos exemplos abaixo:

**ReaisPositivos ::= REAL (0**

# *2.1.3 Null*

O tipo null é um tipo com apenas um valor, diferente de qualquer outro tipo da ASN.1. No entanto a notação **NULL** serve tanto para definir valores como tipos.

Essa definição não é muito utilizada, limitando-se a situações onde um tipo ou valor deve ser criado, mas sem conter nenhuma informação, visando reservar espaço para futuras alterações no formato do tipo. A estrutura abaixo é um exemplo disso:

```
ValorInteiroOpcional ::= CHOICE{INTEGER, NULL}
```

A estrutura da palavra-chave **CHOICE** está explicada adiante, mas o tipo acima funciona da seguinte forma: se for atribuido um valor inteiro, essa será a informação, caso contrário nehuma informação será válida. Essa estrutura pode ser substituída com vantagens por tipos **set** e **sequence** com campos opcionais.

# *2.1.4 Boolean*

O tipo boolean é idêntico ao tipo boolean do Pascal, podendo assim assumir os valores representados pelas palavras-

chave **TRUE** e **FALSE** e sendo utilizado em informações lógicas. Vale ressaltar que seu uso pode ser substituído por **bit strings** com vantagens em situações onde são necessárias várias variáveis de dois valores, pois reduz os octetos gerados na codificação.

# *2.1.5 Enumerated*

Um usuário pode definir seu tipo enumerado como abaixo:

```
Cores ::= ENUMERATED

   {
    azul(0), amarelo(1), verde(3)
   }
```

ou

```
Prioridade ::= ENUMERATED {menor(1), medio(2), maior(3)}
```

Essa definição é semelhante àquelas do tipo inteiro, mas com algumas diferenças:

- os valores estão limitados àqueles definidos;
- os números não tem nenhum significado, só servindo para definir representações distintas para os valores;
- não existe nenhuma relação de ordem entre os identificadores;
- somente os identificadores podem ser usados como valores.

Os valores colocados poderiam ser retirados, mas são mantidos simplesmente para facilitar possíveis alterações, podendo incluir valores entre os já existentes. Portanto a estrutura abaixo é válida:

```
Prioridade ::= {menor, medio, maior}
```

# *2.1.6 Bit String*

Este tipo serve para criar outros, cujos valores serão representados por cadeia de bits e é representada pela palavra-chave **BIT STRING**. Pode-se definir o tipo diretamente ou criar identificadores para os bits. Estas duas formas aparecem representadas abaixo:

```
CadeiaDeBits ::= BIT STRING

CadeiaIdentificadaBits ::= BIT STRING
     {
     bitZero(0), bitUm(1), bitDois(2)
     }
```

No primeiro caso temos um tipo que pode receber valores com qualquer número de bits. No segundo caso, o tipo levaria a definição de valores com apenas três bits.

Quanto a numeração colocada, ela é obrigatória e sempre teremos os bits de zero a n-1 para um tipo definido com n bits, mesmo que ele não esteja referenciado por um identificador.

A maneira para declarar valores para tipos definidos com identificadores é a forma abaixo:

```
Valor CadeiaIdentificadaBits ::= {bitZero,bitDois}
```

Onde os bits que aparecem entre as chaves assumirão valor 1 e os demias zero.

Para atribuir valores para tipos sem identificadores podemos usar valores em hexadecimal, com o caracter especial **H**, e valores em binário, com o caracter especial **B**:

```
valorBinario BIT STRING ::= '0011111010001'B
valorHexadecimal BIT STRING ::= '4B1'H
```

Vale ressaltar que se não completarmos os octetos, os bits menos significativos, na hora da codificação, serão considerados zero de forma a completar os octetos, em ambos os casos. Além disso, a segunda atribuição só é valida se o tipo tiver um número de bits múltiplo de quatro.

Utilizando a palavra-chave **SIZE** podemos limitar o número de bits de um determinado sub-tipo conforme os exemplos abaixo:

```
Palavra ::= BIT STRING (SIZE (0..7))
```

Na definição acima o tipo **Palavra** tem 8 bits.

# *2.1.7 Octet String*

Este tipo é semelhante ao tipo anterior, no entanto temos que a informação deve ter um número de bits múltiplos de oito. As formas abaixo são exemplos válidos:

```
palavra OCTET STRING    ::= '0100010111001111'B
word OCTET STRING       ::= 'AE7'H
PalavraTamanhoReservado ::= OCTET STRING (SIZE (0..3))
exemplo PalavraTamanhoReservado ::= '34EA32'H
```

Uma outra diferença a ser observada é que ao usar a palavra-chave **SIZE** estamos nos referindo ao número de octetos e não bits.

Um detalhe importante a ser observado é que os valores abaixo são os mesmos:

```
valor OCTET STRING ::= '097DE'H
valor OCTET STRING ::= '097DE0'H
```

Ou seja, caso não complete um octeto, a segunda parte do mesmo será considerada zero quando da codificação.

# *2.1.8 Character Strings*

Este tipo em especial é usado para trabalhar com informações na forma de texto, sendo que existem oito tipos de dados para representar cadeias de caracteres em ASN.1, sendo que dois deles podem ser representados por dois nomes. Os tipos e os caracteres que cada tipo representa aparecem abaixo:

| | |
|---|---|
| **NumericString** | 0 - 9 e espaço |
| PrintableString | a - z A - Z 0 - 9 ' " ( ) + , - . / : = ? e espaço |
| TeletexString (T61String) | definido pela CCITT na recomendação T.61 |
| VideoTextString | definido pela CCITT nas recomendações T.100 e T.101 |
| VisibleString (ISO646String) | caracteres da IA5 (Versão internacional da ASCII) e espaço |
| IAString | caracteres da IA5 |
| GraphicString | todos os caracteres gráficos registrados e espaço |
| GeneralString | todos os caracteres gráficos e de controle registrados, espaço e delete |

Os exemplos abaixo são declarações válidas:

```
matrícula NumericString   ::= "5690"
nome      PrintableString ::= "Rosa"
grego     GraphicString   ::= "ß"
```

Novamente pode se usar a palavra-chave **SIZE** para limitar o tamanho:

```
PlacaDeCarro ::= Printable String (SIZE(9))
```

Definimos então um subtipo com tamanho de nove caracteres.

Já a palavra-chave **FROM** pode ser usada para criar um sub-tipo com apenas alguns dos caracteres de um tipo existente conforme exemplificado a seguir:

```
Vogais ::= PrintableString (FROM ("a" | "e" | "i" | "o" | "u"))
```

Podemos usar também os dois operadores separadamente de duas formas distintas:

```
NumericoPuroLimitado ::= NumericString
(Size(9))
(FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"))
```

ou

```
LimitadoCaracteresOuTamanho ::= NumericString
(Size(9)
| FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"))
```

No primeiro caso temos que o tamanho do tipo é limitado e este só contém os caracteres especificados. Já no segundo caso, temos que o tipo ou tem nove caracteres do tipo **NumericString** (incluindo espaço) ou (operador "|") tem tamanho livre somente com os caracteres especificados.

Alguns valores válidos para o segundo exemplo são:

```
exemplo1 LimitadoCaracteresOuTamanho ::= "1 2 3 4 5"
(9 caracteres - primeiro caso)

exemplo2 LimitadoCaracteresOuTamanho ::= "0123456789"
(10 caracteres - segundo caso)

exemplo3 LimitadoCaracteresOuTamanho ::= "01234"
(qualquer um dos casos)
```

Já um valor ilegal aparece abaixo:

```
exemplo4 LimitadoCaracteresOuTamanho ::= "1 2 3 4 5 6"
(se tiver espaço só até 9 caracteres)
```

Para representar aspas devemos colocar um par de aspas como abaixo:

```
aspas PrintableString ::= " "" "
```

# *2.1.9 Object Identifier*

Os valores do tipo correspondem aos nós da árvore de identificação de objeto **(OIT - Object Identifier Tree)**.

A notação é feita simplesmente usando-se a palavra-chave **OBJECT IDENTIFIER**. Assim:

```
Atributos ::= OBJECT IDENTIFIER
```

A forma para descrever um objeto é a mostrada abaixo:

```
númeroDeTelefone Atributos ::= {2 5 4 20}
```

Esse objeto foi projetado e normalizado pela **ISO** e pela **CCITT** em conjunto e define os atributos do número de telefone. Um objeto é escolhido então pelos valores que aparecem entre as chaves.

# *2.2 Tipos Estruturados*

Os tipos estruturados da ASN.1 são:

- Set
- Sequence
- Set of
- Sequence of
- Choice
- Any
- Tag

# *2.2.1 Set*

O tipo set é definido por uma coleção de tipos, todos eles diferentes. Com isso, um valor de um tipo set deve ter um valor para cada um dos componentes, sendo que a ordem na qual esses valores são atribuídos é independente da ordem em que estes componentes foram declarados, já que são todos de tipos diferentes.

No exemplo abaixo, onde definimos um tipo e um valor para esse tipo, podemos observar como isso é feito:

```
TipoA ::= SET
{
  a  INTEGER,
  b  BOOLEAN,
  c  OCTET STRING
}

valorA TipoA ::=
{
  c   '5F'H,
  b   TRUE,
  a   -1
}
```

Se quisermos utilizar tipos iguais dentro de uma dessas estruturas devemos colocar uma etiqueta (**tag**) em cada um deles. Esse conceito será discutido mais a frente, mas faz com que, ao ser codificado, o tipo seja considerado diferente do tipo original. Para evitar o trabalho de verificar sempre se esta ou aquela estrutura devem ou não usar essas etiquetas, é comum usá-las sempre. Isso evita que se verifique se um dado importado de um outro módulo é ou não de um tipo já usado. Assim, a estrutura abaixo passa a ser válida:

```
TipoB ::= SET
{
  d [0] INTEGER,
  e [1] INTEGER
}
```

Os identificadores de campo podem ser retirados, mas isso pode levar a ambiguidades. Usando a palavra-chave **OPTIONAL** podemos definir que pode ou não ser referenciado quando da atribuição de valores. Consideremos o tipo abaixo:

```
TipoC ::= SET
{
  a  [0] INTEGER,
  b  [1] BOOLEAN,
  c  [2] OCTET STRING OPTIONAL
}
```

Ambas as atribuições de valores abaixo são válidas:

```
valorC1 TipoC ::=
{
  a  12,
  b  TRUE,
  c  '9A'H
}

valorC2 TipoC ::=
{
  a  12,
  b  TRUE
}
```

Se no entanto quisermos que, caso um campo seja exluído quando da definição de valores, assuma um valor específico, podemos usar a palavra-chave **DEFAULT** conforme exemplificado abaixo:

```
TipoD ::= SET
{
  a  [0] INTEGER,
  b  [1] BOOLEAN,
  c  [2] OCTET STRING DEFAULT '32'H
}
```

Se não for atribuído valor ao campo **c** ele assumirá o valor '32'H. Com a palavra-chave **COMPONENTS OF** podemos incluir uma estrutura dentro da outra:

```
TipoE ::= SET
{
  d INTEGER,
  e INTEGER,
  COMPONENTS OF TipoD
}
```

Com esse tipo de definição os campos **a**, **b**, **c**, assim como os campos **d** e **e**, fazem parte do **TipoD** criado. Existe porém uma maneira de criar um novo tipo com apenas alguns dos campos de um tipo existente com a palavra-chave **WITH COMPONENTS** em conjunto com as palavras-chave **PRESENT** e **ABSENT**. Considerando o tipo **TipoC** definido acima, vamos exemplifica r o uso deste formato:

```
SubTipo ::= TipoC
    {
       WITH COMPONENTS
            {
               a  (0..20),
               b
            }
    }


SubTipo ::= TipoC
    {
       WITH COMPONENTS
            {...,
               a  (0..20),
               b,
               c  ABSENT
            }
    }
```

As duas maneiras acima são equivalentes, onde temos no tipo **SubTipo** os campos **a** e **b**, sendo que o campo **a** foi também delimitado, o que é permitido nesse tipo de estrutura. A palavra-chave **ABSENT** faz com que um campo antes definido com **OPTIONAL** fique fora da estrutura. Já a palavra-chave **PRESENT** faria com que ele estivesse sempre presente.

# *2.2.2 Sequence*

É muito semelhante a estrutura do tipo **SET**, mas os componentes aparecem em ordem fixa e não precisam ser de

tipos diferentes.

Assim, para a estrutura abaixo:

```
TipoA ::= SEQUENCE
{
  a   INTEGER,
  b   BOOLEAN,
  c   INTEGER OPTIONAL
}
```

A atribuição abaixo não é válida:

```
valorA TipoA ::=
{
  a  10,
  c  15,
  b  FALSE
}
```

Vale ressaltar que, como a ordem é importante, os identificadores de campo podem ser retirados sem que haja risco de ambiguidade. Assim, a estrutura a seguir é válida:

```
TipoB ::= SEQUENCE
{
   INTEGER,
   BOOLEAN,
   INTEGER OPTIONAL
}
```

Como os campos dessa estrutura podem ser de tipos iguais, o uso da palavra-chave **OPTIONAL** pode trazer ambiguidades. Para que isso não ocorra, quando um ou mais campos dessa estrutura forem consecutivos, eles não devem ser do mesmo tipo, e, após um campo definido como opcional, não podemos ter uma campo comum do mesmo tipo do opcional.

As palavras-chave **COMPONENTS OF**, **WITH COMPONENTS**, **ABSENT** e **PRESENT** podem ser usadas da mesma forma que no tipo **SET**.

# *2.2.3 Set-of*

Este tipo é definido em termos de um outro simples, conforme os exemplos abaixo:

```
TipoA := SET OF INTEGER
TipoB := SET OF TipoDefinido
```

Para valores, temos:

```
valor1 SET OF INTEGER ::= {6,4,1,0,0,3}
valor2 SET OF INTEGER ::= {0,3,4,1,6,0}
```

Como a ordem não importa, os dois valores acima são idênticos.

A palavra-chave **SIZE** pode ser usada para alterar o número de componentes da estrutura ou até mesmo o tamanho dos componentes (em caso de strings). Os exemplos abaixo demonstram isso:

```
TipoC ::= SET OF PrintableString (SIZE(5))
TipoD ::= SET SIZE(5) OF PrintableString
```

No primeiro caso **TipoC** tem apenas 5 componentes , que são strings sem tamanho limitado. Já no segundo caso, **TipoD** pode ter um número teoricamente ilimitado de strings de até 5 caracteres.

# *2.2.4 Sequence-of*

Tem exatamente as mesmas características de **SET OF**, sendo que a ordem dos componentes passa a ser importante.

# *2.2.5 Choice*

É uma coleção de campos, todos de tipos distintos, onde o valor de um componente desse tipo definido será apenas um dos campos. Essa estrutura é semelhante ao tipo **union** da linguagem C.

Por exemplo:

```
TipoEscolha ::= CHOICE
    {
        x  [0]   REAL,
        y  [1]   INTEGER,
        z  [2]   BOOLEAN
    }
```

A notação para um valor desse tipo pode ser uma das alternativas abaixo:

```
valor1 TipoEscolha ::= x {2, 10, -1}
valor2 TipoEscolha ::= y 10
valor3 TipoEscolha ::= z  TRUE
```

O uso de etiquetas (**tags**) na definição dos tipos evita que haja problemas com tipos iguais dentro da estrutura. Abaixo temos um exemplo de uma estrutura ilegal e como ela pode ser feita legal através do uso das etiquetas.

```
TipoIlegal ::= CHOICE
     {
      a  CHOICE {t   INTEGER, u   BOOLEAN},
      b  CHOICE {v  INTEGER, w   REAL}
     }

TipoLegal ::= CHOICE
     {
      a  [0] CHOICE {t   INTEGER, u   BOOLEAN},
      b  [1] CHOICE {v  INTEGER, w   REAL}
     }
```

No primeiro há uma coincidência de tipo, pois **a** e **b** possuem o tipo **INTEGER**.O uso de **tags** faz com que ao serem codificados os tipos assumam valores representativos diferentes, por isso eles não são encarados como sendo iguais.

Podemos usar a seleção de tipo para criar novas estruturas a partir da descrita anteriormente. Isso é feito o nome do tipo **CHOICE** e o identificador do tipo, separados pelo símbolo "<", que pode ser lido como "**alternativa de**". A seleção de tipo pode ser usada em outra estrutura CHOICE ou sem uma estrutura **SET ou SEQUENCE**.

Temos, usando o tipo **TipoEscolha** definido acima:

```
TipoSelecao ::= SEQUENCE
     {
        x < TipoEscolha,
        z
```

Com o seguinte valor válido:

```
valorSeleção TipoSeleção ::= { x   {2, 10,-1} , z   FALSE}
```

Com esse tipo de definição os nomes dos campos permanecem os mesmos da estrutura tipo **CHOICE** original. Porém eles podem ser renomeados conforme abaixo:

```
TipoSeleçãoNovo ::= SEQUENCE
     {
        xnovo    x < TipoEscolha,
        znovo    z
```

Com o seguinte valor válido:

```
valorSeleçãoNovo TipoSeleçãoNovo ::= { xnovo   {2, 10,-1} , znovo   FALSE}
```

No entanto, se for criado um novo tipo **CHOICE** derivado de um outro é mais prático e usado o método baseado na palavra-chave **WITH COMPONENTS**, já citado quando da descrição dos tipos estruturados **SEQUENCE** e **SET**.

Assim podemos definir um novo tipo derivado de **TipoEscolha**:

```
TipoEscolhaNovo ::= TipoEscolha
     ( WITH COMPONENTS {
        x,
        z }
     )
```

Onde temos os campos **x** e **z**. Vale ressaltar que, como no caso descrito anteriormente, podemos criar um subtipo do tipo original incluído com o formato abaixo.

```
TipoEscolhaNovo ::= TipoEscolha
     ( WITH COMPONENTS {
        ...,
        y  (1..10),
        z }
     )
```

O operador "**...**" acrescenta os tipos acima do primeiro definido. Assim teremos os tipos **x**, **y** e **z**, sendo que **y** não é mais do tipo **INTEGER**, mas do subtipo **INTEGER (1..10)**.

Caso estejamos trabalhando em uma situação que excluímos apenas alguns elementos de uma lista de muitos, podemos usar a palavra-chave **ABSENT**, que aqui assume papel diferente do descrito para os tipos **SEQUENCE** e

**SET**.

```
TipoEscolhaNovo2 ::= TipoEscolha
    (WITH COMPONENTS {..., z ABSENT})
```

E nesse caso teremos os tipos **x** e **y** fazendo parte do tipo definido.

# *2.2.7 Any*

Um valor do tipo *ANY* pode assumir qualquer valor de qualquer tipo da ASN.1. Ele pode ser usado para simplesmente criar um buraco em uma definição de um tipo o qual ainda não foi especificado e que o será posteriormente.

Como representa valores de qualquer tipo a atribuição abaixo se torna válida:

```
v1 ANY ::= INTEGER 505
```

Como pode assumir valores de qualquer tipo, não pode ser usado em situações onde tipos diferentes são exigidos. Portanto, se não forem usadas etiquetas, ele não poderá ser usado nem em estruturas do tipo **SET** nem do tipo **CHOICE**.

Quando da apresentação do tipo **SEQUENCE** também foram expostos possíves problemas de ambiguidade quando a palavra-chave **OPTIONAL** é usada. Assim não se pode ter um tipo **ANY**

definido após um outro declarado como **OPTIONAL**; e caso um tipo **ANY** seja definido como opcional, ele só poderá aparecer como o último do grupo de tipos.

A palavra **DEFINED BY** permite que haja uma relação entre dois campos de um tipo **SEQUENCE** ou **SET**, de forma que o tipo de valor que será atribuido ao tipo **ANY** pode ser detreminado por um outro campo da estrutura.

Por exemplo:

```
TipoAtribuição ::= SEQUENCE
      {
          referenciador  INTEGER,
          referenciado   ANY DEFINED BY referenciador
     }
```

Nessa estrutura o tipo **referenciador** determinará o tipo do valor **referenciado**:

```
valor TipoAtribuição ::=
{
    referenciador  10,
    referenciado    REAL  {10,2,-1}
}
```

Assim, para um **referenciador** de valor 10, **referenciado** será do tipo **REAL**, e no caso, **referenciado** assume o valor especificado. No entanto, sempre que **referenciador** estiver com valor 10, referenciado é do tipo **REAL**, mas pode assumir outro valor.

# *2.3 Useful Types*

Existem alguns tipos que são definidos na especificação da ASN.1 usando eles mesmos a ASN.1, e que estão disponíveis ao usuário através de referências comuns de tipo. Esses tipos são conhecidos como tipos úteis **(useful types)** porque poderiam ter sido facilmente criados por um usuários comuns, mas foram definidos devido a sua grande utilidade.

A única diferença entre a sua definição e uma definição de tipo aparece no fato de que os tags a eles associados são universais, recurso este que não está disponível ao usuário comum da ASN.1. Na especificação da ASN.1 esse tipos aparecem definidos como no exemplo abaixo:

```
ObejectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString
```

Na referência a tags aparece o siginificado de tal definição e podemos observar que o tag original desse tipo é realmente 7 na tabela de tags.

Existem quatro desses tipos na ASN.1, dois relativos a datas e hora, um para embutir outros tipos de sintaxes abstratas na ASN.1, e um para relacionar uma string a um Object Identifier criado.

- Generalized time
- Universal time
- External
- Obeject descriptor

# *2.3.1 Generalized Time*

É usado na transmissão de hora e data em forma de texto. A representação usada é **AAAAMMDDHHMMSS.SSS**, ou seja, ano, mês, dia, hora, minuto e segundos, este último com precisão de até milisegundos.

Para representar o dia 6 de outubro de 1979, as 12:00 Hs e 38.5 segundos, podemos usar a notação abaixo:

```
momento GeneralizedTime ::= "19791006120038.5"
```

A sua definição na ASN.1 aparece abaixo:

```
GeneralizedTime ::= [UNIVERSAL 24] IMPLICIT VisibleString
```

# *2.3.2 Universal Time*

É definido com a palavra-chave **UTCTime** e possui duas diferenças em relação ao tipo **GeneralizedTime**: o ano só é representado por dois caracteres e só temos precisão de minutos.

O mesmo exemplo dado para **GeneralizedTime** ficaria:

```
momento UTCTime ::= "79100612001"
```

Este tipo foi criado para se adequar mais ao calendário cristão, onde é comum usar somente dois dígitos para o ano.

Sua definição na ASN.1 é:

**UTCTime ::= [UNIVERSAL 23] IMPLICIT VisibleString**

# *2.3.3 External*

Esse tipo pode ser usado para encaixar, dentro de valores de uma determinada sintaxe abstrata, um valor de uma outra sintaxe. Esse valor pode ser ou não um valor de um tipo da ASN.1. Por exemplo:

```
ConteúdoDeArquivo ::= EXTERNAL
```

Na prática cada um desses valores externos está associado a uma regra de codificação e uma referência que possa ser usada pelo receptor para fazer a devida interpretação do valor.

A sua representação na definição da ASN.1 aparece abaixo:

```
EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE
{
  direct-reference      OBJECT IDENTIFIER OPTIONAL,
  indirect-reference   INTEGER OPTIONAL,
  data-value-descriptor  ObjectDescriptor OPTIONAL,
  encoding
     {
       single-ASN!-type   [0] ANY,
       octet-aligned      [1] OCTET STRING,
       arbitrary          [2] BIT STRING
     }
}
```

# *2.3.4 Object Descriptor*

É usado para associar uma string a um **Object Identifier** criado para facilitar o sua utilização para o usuário.

O exemplo abaixo demonstra como essa associação é feita:

```
whizzoSpread ObjectDescriptor :=
  "Whizzo Corp's Spreadsheet File Format V3.09"
```

Na ASN.1 temos a seguinte declaração para este tipo:

**ObjectDescriptor ::= [UNIVERSAL 8] IMPLICIT GraphicString**

# *3 Macros*

As macros surgem como uma nova maneira de definir tipos em ASN.1. No entanto não introduzem novos tipos

falando em termos de codificação, sendo todos equivalentes a tipos pré-definidos pela ASN.1.

Não existe analogia com as macros da Linguagem C, já que estas simplesmente funcionam como uma substituicão no texto. Aqui o programador da macro a define de forma a devolver um valor de um tipo qualquer da ASN.1. O usuário da macro é que definirá o tipo de valor retornado pela macro mediante os tipos e valores que utilizou ao referenciá-la.

A fim de melhor esclarecer essa idéia, que a princípio parece confusa, consideremos o exemplo abaixo:

```
EXEMPLO MACRO ::= BEGIN

TYPE NOTATION ::=
  type (Tipo) ::=

VALUE NOTATION ::=

  "OMITIDO"    |

  value (valor Tipo)

END
```

Uma primeira observação a ser feita é quanto a criação do nome de uma macro; é usual criar nomes com todas as letras maiúsculas.

Quanto ao novo formato por ela introduzido, a macro nos permite escrever tipos como **EXEMPLO INTEGER** ou **EXEMPLO REAL**, ou qualquer outro tipo pré-definido da ASN.1. Para provar isso consideremos a definição de tipo abaixo:

```
TipoTeste ::= EXEMPLO SEQUENCE OF INTEGER
```

Na prática essa macro simplesmente serviria para criar um tipo como o definido abaixo:

```
TipoTeste ::= CHOICE {SEQUENCE OF INTEGER, NULL}
```

A maneira como isso funciona é a seguinte: a definição de uma macro assume que o tipo passado na definição de tipo referenciando-a está ligado a referência Tipo que aparece entre parênteses após a palavra-chave **type**. Além disso vemos que entre os sinais de maior e menor característicos também da definição as palavras-chave **present** e **omitted** significam que o tipo ali presente está ligado a referência **Tipo** (**present**) ou não (**omitted**), de onde a definição feita acima simplemente criar um tipo **CHOICE** conforme exposto acima. O nome definido **TipoDoExemplo** será substituído pelo nome **TipoTeste** usado na definição de tipo. Isso é importante para a definição de valores.

Isso funciona como se na macro tivéssemos as seguintes pré-definições:

```
Tipo ::= SEQUENCE OF INTEGER
TipoDoExemplo ::= CHOICE {present Tipo, omitted NULL}
```

Agora vamos a atribuição de valores para a macro. Abaixo aparecem dois exemplos:

```
valor1  TipoTeste ::= OMITIDO
valor2  TipoTeste ::= {1, 2, 0}
```

Olhando para a gramática definida no interior da macro, vemos que temos duas possibilidades separadas pelo caracter especial "|" já usado em outras definições e que aqui separa as possíveis formas de definição de valores. A

primeira é a string **"OMITIDO"**. Quando da definição de um valor eu escolho essa alternativa de referência eu entro na linha abaixo:

```
VALUE TipoDoExemplo ::= omitted NULL
```

Vale ressaltar aqui que **VALUE** é uma palavra-chave da ASN.1. Seguindo as pré-definições criadas quando da definição do tipo, temos que estamos criando um valor do tipo **NULL**. Assim vemos que a definição de tipo influencia na definição dos valores e que podemos usar strings próprias para refrenciar formas de definir nossos valores.

O valor valor1 criado nos leva a outra forma de criar um valor em uma macro. Ela significa que estamos criando um valor do tipo pré-definido na definição do tipo (no caso **Tipo**) associado ao nome de valor **valor1** usado na criação do valor. Isso é o que significam os nomes **valor** e **Tipo** no interior dos parênteses. Assim criamos aqui uma pré-definição **valor Tipo ::= {1, 2, 0}**, que é em sí uma definição de valor já conhecida.

A linha entre sinais de maior e menor associa então o valor **{1, 2, 0}** ao valor criado **valor2**.

O exemplo acima mostrou apenas algumas das possibilidades das macros. Mostrou também como elas são definidas: toda a estrutura delimitada pelas palavras-chave **BEGIN** e **END**.

As outras possibilidades quando da criação de uma macro aparecem na tabela abaixo:

| | |
|---|---|
| "nome" | a string "nome" sem as aspas, que poderá ser usada diretamente pelo usuário como referência direta, conforme exemplo acima |
| identificador | seguindo as regras para criaçao de nomes de identificadores, numerando uma sequência de dígitos |
| **empty** | palavra-chave. Refere-se a uma string vazia |
| **type** | palavra-chave. Refere-se a qualquer tipo pré-definido da ASN.1 |
| **type** (referência de tipo) | igual acima, só que o tipo aparece associado a referência de tipo entre parênteses |
| **value** (tipo) | qualquer string que possa ser reconhecida como uma definição de tipo para valor |
| **value** (referência de valor) | igual acima, mas valor é associado a expressão entre parênteses |

Um outro exemplo que demosntra as novas possibilidades expostas aparece abaixo:

```
EXEMPLO MACRO ::=

BEGIN

  TYPE NOTATION  ::= Argumento Resultado Erros
  VALUE NOTATION ::= value (VALUE Código)

  Argumento    ::=  "ARGUMENTO"  type | empty
  Resultado    ::=  "RESULTADO"  TipoResultado | empty
  Erros        ::=  "ERROS"  "{" NomeDoErro "}" | empty

  TipoResultado   ::=  type | empty
  NomeDoErro      ::=  ListaDeErros | empty
  ListaDeErros    ::=  Erro | ListaDeErros "," Erro
  Erro            ::= value(ERRO)

END

Código  ::= INTEGER
```

Um exemplo de uso é o mostrado abaixo:

```
valorExemplo  EXEMPLO
        ARGUMENTO  TipoAtributo
        RESULTADO    ValorDeArgumento
        ERROS              {semAtributo, acessoNegado}
    ::= 5
```

Podemos observar que muitas coisas acontecem ao mesmo tempo. No entanto, a nível de atribuição de valores estamos simplesmente atribuindo o valor 5 a **valorExemplo**.

É interessante observar como as strings escolhidas pelo criador da macro são diretamente referenciadas, mas sem as aspas. Sempre que são usadas, temos que considerar as solicitações que vêm após as mesmas. No exemplo anterior ela simplesmente referenciava uma linha, mas neste exemplo seu uso pede após a mesma, alguma definição conforme o modelo da macro.

Embora somente estejamos atribuindo o valor 5 a **valorExemplo** (o que é efetivamente transmitido) podemos ver que estamos definindo outras características ligadas à criação dos tipos. É claro que isso poderia ser feito diretamente, sem ser necessário que essa criação fosse feita na macro.

O problema principal das macros é que nada efetivamente será feito com os tipos criados dessa forma. Somente teremos tipos referenciados ao valor 5, mas eles não poderão ser usados fora da macro da maneira como foram definidos. A maneira que isso pode ser feita é a usada no exemplo anterior. Misturando as características mostradas, podemos criar macros que nos sejam úteis.

# *4 Basic Enconding Rules(BER)*

As regras de codificação aqui citadas estão presentes nas normas da X.409 definidas pela **CCITT**, que primeiro padronizaram a notação abstrata aqui estudada.

Essas regras definem como os valores definidos para os tipos criados serão transmitidos. Atualmente um valor é codificado de forma a ter três partes: o identificador (**I**), o tamanho (**T**), e o conteúdo (**C**).

Quando temos um valor de um tipo simples, a codificação aparece como abaixo:

| **I** | **T** | **C** |
|---|---|---|

Quando temos um tipo estruturado, a codificação tem um formato parecido com o exposto abaixo:

| **I** | **T** | **I** | **T** | **I** | **T** | **C** | **I** | **T** | **C** |
|---|---|---|---|---|---|---|---|---|---|

Para melhor compreender essa codificação, vamos analisar cada uma dessas três partes separadamente.

Um fato interessante é que identificadores de tamanho de tipos não influem na codificação. Se definirmos uma string com no máximo 10 caracteres, essa informação não aparece na codificação. O tamanho que aparece é o tamanho efetivo do valor. Essa limitação de tamanho e criação de sub-tipos é útil para o decodificador para que ele reserve espaço adequado na memória para o valor. Isso é discutido nos aspectos da implementação.

- Identificador
- Tamanho
- Conteúdo

# *4.1 Identificador*

Essa é a primeira parte da codificação, e tem um número inteiro de octetos, número este que varia de acordo com o tag associado ao tipo, que é um número sem máximo fixado.

De acordo com o tag, o identificador terá um ou varios octetos. Se o tag estiver da faixa de 0 a 30 (inclusive), teremos um único octeto no formato abaixo:

```
87654321
CCFTTTTT
```

Os dois bits **CC** representam a classe do tag associado e o bit **F** representa a forma do tag segundo as tabelas abaixo:

| **CC** | **Classe do tag** |
|---|---|
| 00 | Universal |
| 01 | aplicação-larga |
| 10 | contexto-específico |
| 11 | uso-privado |

| **T** | **Forma** |
|---|---|
| 0 | primitivo |
| 1 | construído |

A forma está ligada a construção do tipo, e será melhor explicada adiante.

Os bits **TTTTT** definem o número do tag.Se eles assumirem o valor **11111** isto significa que o identificador terá mais de um octeto. Por isso, para um valor maior que 30 de tag, cairemos neste caso.

Os valores dos tags do tipo universal estão na tabela apresentada quando da definição dos tags.

Para um identificador com vários octetos temos a estrutura abaixo:

```
87654321    87654321            87654321
CCF11111    1TTTTTTT     ...    0TTTTTTT
   1           2                   n
```

Podemos observar que o que identifica o último octeto é um valor zero no **bit 8** (mais significativo) e que uma concatenação dos sete bits dos octetos de **2** a **n** definem o valor do tag.

Um exemplo de identificador simples é:

**$01100110\ (66_{16})$**

que é do tipo aplicação-larga e forma construída, com número de tag 6.

Para um identificador com múltiplos octetos temos o exemplo abaixo:

**$00011111\ 10110101\ 01110110\ (1FB576_{16})$**

Neste exemplo temos um tag do tipo universal e primitivo com um valor definido da forma abaixo:

**$(1)0110101\ (0)1110110 = 011010\ 11110110 = 1AF6_{16} = 4096$**

Para diminuir o número de octetos na codificação, devemos, portanto, evitar valores elevados para os **tags**.

# *4.2 Tamanho*

Esta é a segunda parte da codificação, que também pode ocupar de um, a um número teoricamente ilimitado de octetos, de acordo com o tamanho do conteúdo do valor, que não tem limites na ASN.1.

Graças a esta parte da codificação o fim das mensagens podem ser encontrados, independente do tag associado.

O tamanho pode ser de três tipos: **curto**, **longo** e **indefinido**, sendo que os dois primeiros são também referidos como sendo do tipo tamanho **definido**.

O tamanho curto ocupa um octeto e é usado para conteúdos com tamanhos variando de 1 a 126 (inclusive) octetos, seguindo o formato abaixo:

```
87654321
0LLLLLLL
```

onde os bits **LLLLLLL** representam o tamanho.

O tamanho longo é codificado como abaixo:

```
87654321  87654321        87654321
1nnnnnnn  LLLLLLLL  ...   LLLLLLLL
    1         2             N+1
```

Aqui os bits **nnnnnnn** representam o valor de **N**, e podem variar de 1 a 127. Assim o tamanho é colocado nos octetos de **2** a **N+1**.

Vale resaltar que, ao contrario do identificador, estes octetos precisam ser decodificados antes da recuperação dos dados para que se saiba o tamanho do conteúdo.

O tamanho máximo que pode ser representado dessa última forma é da ordem de $10^{313}$ mas, teoricamente, podemos definir tamanhos ilimitados, usando o tipo indefinido. Para tanto o primeiro octeto tem o valor **10000000**. Isso irá informar ao decodificador que ele deve procurar a marca de fim de conteúdo representada pelos dois octetos mostrados abaixo:

```
87654321 87654321
00000000 00000000
    1        2
```

Quando apresentamos o conceito de tag mostramos que o tag universal 0 (representado acima) é reservado. Aparece aqui o motivo desta excessão. No entanto isso só é possível quando o conteúdo é construído. Isso é um caso muito especial e sem grande importância prática, mas serão mostrados exemplos adiante de como o codificador faz isso.

São exemplos de tamanhos os valores abaixo:

```
00001000 : tamanho 8
```

**10000010 00000001 11111111 : tamanho 511**

# 4.3 Conteúdo

O conteúdo representa o valor que realmente deve ser transmitido. Como este varia de acordo com o tipo do valor, apresentaremos este campo da codificação separadamente para cada valor.

- Integer
- Real
- Null
- Boolean
- Enumerated
- Bit, Octet e Character String
- Object Identifier
- Set, Sequence, Set of e Sequence of
- Tag
- Any
- Useful e demais tipos

# 4.3.1 Integer

A forma para um inteiro é primitiva, e os octetos representam o valor inteiro em complemento a dois, podendo, dessa forma, representar números positivos e negativos.

Consideremos a atribuição de valor abaixo:

```
valor1 INTEGER ::= -44
```

a codificação para **valor1** é:

**02 01 $D4_{16}$**

ou seja, tag (valor 2) universal e primitiva, com tamanho de conteúdo 1, e com conteúdo -44.
Outro exemplo, com tamanho 2, aparece abaixo:

```
valor2 INTEGER ::= 17980
```

e a codificação seria:

**02 02 $463C_{16}$**

# 4.3.2 Real

A codificação de um número real é primitiva e pode ser feita de quatro formas distintas:

**i)** o valor zero (Sem nenhum conteúdo);
**ii)** valores mais -infinito e menos-infinito;

**iii)** representação binária na base 2, 8 e 16;
**iv)** representação decimal baseada em caracteres;

O valor zero, representado como abaixo:

```
zero  REAL ::=0
```

é codificado como:

**$09\ 00_{16}$**

O conteúdo de mais-infinito é codificado como abaixo:

```
01000000
```

E o de menos-infinito como:

```
01000001
```

Então, se tivermos:

```
maisInfinito REAL ::= PLUS-INFINITY
```

a codificação será:

**$09\ 01\ 40_{16}$**

A representação binária é representada por um valor 1 no bit 8, e usa cinco campos para definir seu valor:

1) **S** : sinal
2) **B** : base
3) **F** : fator de escala
4) **E** : expoente
5) **N** : derivado da mantissa

Estes aparecem na seguinte forma:

**$S \times N \times 2^F \times B^E$**

Onde o fator $S \times N \times 2^F$ representa a mantissa. Esse tipo de representação visa facilitar a decodificação. Vale ressaltar que o expoente é representado em complemento a dois.

Existem quatro possibilidades de representação binária, de acordo com o tamanho do expoente (N ocupa os demais octetos indicados pelo tamanho):

```
1SBBFF00 EEEEEEEE
    0        1
1SBBFF01 EEEEEEEE EEEEEEEE
   0         1        2
1SBBFF10 EEEEEEEE EEEEEEEE EEEEEEEE
   0         1        2        3
1SBBFF11 nnnnnnnn  EEEEEEEE ... EEEEEEEE
   0         1        2            N+2
```

Os valores dos campos aparecem abaixo:

| bit 7 | Sinal(S) |
|---|---|
| 0 | +1 |
| 1 | -1 |

| bit 65 | Base(B) |
|---|---|
| 00 | 2 |
| 01 | 8 |
| 10 | 16 |
| 11 | reservado |

| bit 43 | Fator de Escala (F) |
|---|---|
| 00 | 0 |
| 01 | 1 |
| 10 | 2 |
| 11 | 3 |

Então, um valor definido como abaixo:

```
valor REAL ::= {171, 2, -3}
```

Teria a seguinte codificação:

```
09 03 80FDAB
```

A representação decimal por caracteres é representado por valores 1 nos bits 8 e 7 do primeiro octeto. Nesse formato os octetos seguintes contém basicamente caracteres ASCII, formando um campo como especificado na **ISO 6093**. Os demais bits do primeiro octeto definem a forma de representação:

```
000001 : ISO 6093 NR1
000010 : ISO 6093 NR2
000011 : ISO 6093 NR3


demais : reservados
```

# *4.3.3 Null*

A codificação de um valor null é primitiva. Como há apenas um valor, é respresentado sem octetos, na forma abaixo:

$$05\ 00_{16}$$

# *4.3.4 Boolean*

Esse tipo de valor é primitivo e ocupa um octeto, sendo que o valor **00000000** representa **FALSE** e os demais

representam **TRUE**.

Assim, temos as seguintes codificações:

**FALSE : 01 01 $00_{16}$**
**TRUE : 01 01 $01_{16}$**

Pelo fato de ocupar um octeto é que se levantou a hipótese de usar o tipo **BIT STRING** ao invés de **BOOLEAN** em alguns casos. Em tipos simples isso não seria vantagem, pois o **BIT STRING** deve ocupar pelo menos um octeto. Mas em situações onde os outros seis bits pudessem ser usados como flags isso seria interessante.

# *4.3.5 Enumerated*

A codificação de um valor enumerated é primitiva. A excessão do tag, a codificação é a mesma do valor do tipo inteiro.

Consideremos o tipo abaixo:

```
Tipo  ::= ENUMERATED {um(1), dois(2), tres(3)}
```

E o valor:

```
valorUm Tipo ::= um
```

A codificação seria:

**$0A\ 01\ 01_{16}$**

# *4.3.6 Bit, octet e character strings*

A codificação pode ser primitiva ou construída, escolha esta feita pelo codificador.

Na codificação primitiva, os octetos representam os elementos da string, exceto para **BIT STRING**, onde o primiero octeto representa o número de bits não usados nos demais octetos.

Dadas as definições abaixo:

```
valor1 BIT STRING ::= '001110111011'B
valor2 OCTET STRING ::= '0AE1C'H
valor3 IA5String ::= "Vela"
```

O valor **valor1** é codificado como:

**$03\ 03\ 043BB0_{16}$**

e **valor2** como:

**$04\ 03\ 0AE1C0_{16}$**

e por último **valor3**:

**16 04 56656C61$_{16}$**

As vezes o codificar opta por enviar mensagens segmentadas, permitindo assim enviar strings não só com tamanho indefinido mas também em casos em que ainda não se conhece o tamanho total da string. Isso retrocede à definição do campo tamanho da codificação em que foi citado um tamanho indefinido, que deve ser construído. Aqui temos um exemplo.

Consideremos o exemplo abaixo, enviado inteiro e segmentado:

```
estória IA5String ::= "Era uma vez ...(sequencia de strings)
                       ... felizes para sempre"
```

Esse valor pode ser representado de duas formas. Uma primeira forma primitiva:

**16 820345$_{16}$**
**45726120...2073656D707265$_{16}$**

E uma segunda com codificação construída:

**36 80$_{16}$**
**04 04 45726120$_{16}$**
**...**
**04 07 2073656D707265$_{16}$**
**00 00$_{16}$**

# *4.3.7 Object Identifier*

A codificação de valores do tipo **object identifier** é primitiva. Será transmitido o identificador de objeto que seria sua posição na árvore de objetos.

Somente sete dos oito bits dos octetos são usados. O bit 8 tem uma função especial: se for o ultimo octeto do valor a ser transmitido, assumirá zero, caso contrário, assumirá 1. Para reduzir a quantidade de octetos, e devido à natureza dos valores dos identificadores (2 primeiros números pequenos devido ao formato da árvore, com o primeiro variando de 0 a 2), os dois primeiros números são combinados em um único octeto seguindo a expressão 40x(primeiro) + segundo.

Dada a atribuição de valor abaixo:

```
relatórioDeTempo OBJECT IDENTIFIER ::= {2 6 6 247 1}
```

Teremos a seguinte codificação:

**06 05 5606817701$_{16}$**

# *4.3.8 Set, Sequence, Set-of e Sequence-of*

A codificação para valores desse tipo é construída e contém uma codificação para o valor de cada componente da estrutura. No caso dos tipos **SEQUENCE** e **SEQUENCE OF** os valores dos campos podem aparecer na ordem em que foram atribuídos, já para os tipos **SET** e **SET OF** os valores dos campos devem aparecer na ordem em que

foram definidos nos tipos, devido a natureza dos mesmos já definidas quando das explicações de sua sintaxe.

Podemos tomar como exemplos as definições abaixo:

```
Tipo1 ::= SET {a INTEGER, b OCTET STRING, c REAL}
valor1 Tipo1 ::= {PLUS-INFINITY, 10, '86'H}
```

O codificador poderia usar qualquer uma das codificações abaixo (além de outras):

```
11 09₁₆
09 01 40₁₆
02 01 0A₁₆
03 01 86₁₆

11 09₁₆
   03 01 86₁₆
   09 01 40₁₆
   02 01 0A₁₆

11 09₁₆
   09 01 40₁₆
   03 01 86₁₆
   02 01 0A₁₆
```

Em contrapartida, se tivermos a definição abaixo:

```
Sequencia ::= SEQUENCE OF (1..10)
valorSequencial Sequencia ::= {1,2,3,4,5,6,7,8,9,10}
```

Devemos ter a seguinte (e única) codificação:

```
10 1E₁₆
   02 01 01  02 01 02  02 01 03₁₆
   02 01 04  02 01 05  02 01 06₁₆
   02 01 07  02 01 08  02 01 08₁₆
   02 01 0A₁₆
```

# *4.3.9 Tag*

Quando temos um **tag** do tipo **IMPLICIT**, as regras de codificação são as mesmas consideradas até o momento, exceto pelo fato de que o número do **tag** é o número atribuído ao mesmo. Consideremos os três exemplos abaixo:

```
valor1 INTEGER := -27066

Tipo2 := [5] IMPLICIT INTEGER
valor2 Tipo2 := -27066

Tipo3 := [APPLICATION 5] IMPLICIT INTEGER
valor3 Tipo3 := -27066
```

As codificações para valor1, valor2 e valor3 aparecem baixo:

**valor1 : 02 02 $9646_{16}$**
**valor2 : 85 02 $9646_{16}$**
**valor3 : 45 02 $9646_{16}$**

Como vemos a diferença na codificação entre **valor1**, **valor2** e **valor3** é apenas o tipo de **tag**, devido as características que a palavra-chave **IMPLICIT** impõe, forçando o **tag** definido como sendo o **tag** do tipo.

Consideremos agora dois exemplos idênticos aos de **valor2** e **valor3** mas explícitos:

```
Tipo4 := [5] INTEGER
valor4 Tipo4 := -27066

Tipo5 := [APPLICATION 5] INTEGER
valor5 Tipo5 := -27066
```

As codificações seriam então:

**valor4 : A5 04 02 02 $9646_{16}$**
**valor5 : 65 04 02 02 $9646_{16}$**

Assim vemos que os dois tags são mantidos e que a pequena diferença entre ambos é apenas o tipo secundário de **tag**.

# *4.3.10 Any*

A codificação do tipo **any** é exatamente a do tipo de valor a ele atribuído. Se for, por exemplo um valor **TRUE** do tipo **BOOLEAN** termos a seguinte codificação:

**$01\ 01\ 01_{16}$**

# *4.3.11 Useful e demais tipos*

As regras para a codificação dos valores dos tipos aqui incluídos é a mesma associada ao tipos dos quais são derivados.

Por exemplo, para os tipos englobados nos **Useful Types**, a codificação seria sobre a sua definição interna da ASN.1. Isso aparece demonstrado abaixo:

```
Hora: 07:05:33.8   Data: 02/01/1982

momento GeneralizedTime := "19820102070533.8"
```

A codificação seria:

**$98\ 0F\ 3139383230323037303533332E38_{16}$**

Ou seja, é do tipo **VisibleString** mas com tag implícito. Portanto seus valores possuem caracteres pertencentes ao tipo **VisibleString** mas o tag é 24.

# 5 ASN.1 e a OSI

A OSI (**Open Systens Interconnection**) é um projeto internacional que visa padronizar o modo como os computadores irão se conectar uns com os outros. Esta dividiu o problema de interconexão entre computadores em sete camadas, cada uma delas envolvendo algum aspecto do problema, sendo que entre cada uma dessas camadas temos um interface que permite que umas camadas se comuniquem com as outras.

No modelo OSI temos definidas sete camadas:

**1 - Camada Física**
**2 - Camada de Enlace de Dados**
**3 - Camada de Rede**
**4 - Camada de Transporte**
**5 - Camada de Sessão**
**6 - Camada de Apresentação**
**7 - Camada de Aplicações**

Não cabe a este curso explicar toda a estrutura do modelo OSI. No entanto, a natureza da camada de apresentação faz dela um importante foco de estudo e, por isso, ela aparece aqui explicada.

- Camada de Apresentação
- Papel da ASN.1 na OSI

# 5.1 Camada de Apresentação

A principal tarefa da camada de apresentação é codificar dados estruturados de acordo com o formato interno do transmissor à um formato adequedo a transmissão dos mesmos e depois decodificá-los de acordo com o exigido no equipamento destino.

Dentre os objetivos desta codificação temos:

**1 -** Compatibilizar estruturas de representação de dados entre computadores baseados em diferentes processadores;
**2 -** Compactação de dados;
**3 -** Criptografia, visando a segurança na transmissão dos dados;

Como exemplo de incompatibilidade de representação interna de dados, podemos citar a forma como os microprocessadores da Intel e da Motorola representam seus dados: enquanto que os processadores da Intel (família 8086) representam seus bytes da direita para esquerda, ou seja, de forma invertida, os microprocessadores da Motorola (família 68000) os representam da esquerda para direita. Além desta diferença exitem outras incompatibilidades entre processadores menos difundidos.

Fica claro que se os dados fossem trasmitidos diretamente haveria uma grande confusão quando da comunicação e intrepretação dos dados. Para evitar isso, em algum lugar deve ser feita uma conversão desses dados.

Essa conversão poderia ser feita de duas maneiras: cada receptor decodificaria os dados recebidos ou o transmissor e o receptor codificariam os dados para um formato de transmissão e os decodificariam de acordo com sua representação.

A primeira solução seria inconveniente pois o receptor deveria ser capaz de identificar as diferenças entre ele e todos os demais processadores com os quais vai se comunicar para ser capaz de adaptar os dados recebidos a seu formato

interno. Já para a segunda solução terímaos um algorítmo bem mais simples: o codificador e o decodificador poderiam se basear em uma estrutura padrão para transmissão e o formato de representação interna dos dados seria irrelevante. É neste ponto que a ASN.1 mostra sua importância.

Quanto a compactção dos dados, fica clara a sua importância, haja visto o custo da transmissão dos dados. Como exemplo podemos citar a transmissão de um inteiro de 32 bits: como 95% desses inteiros estão entre 0 e 250 (os caracteres mais usados do padrão ASCII) poderíamos, por exemplo, trasmitir esse inteiros como um único byte sem sinal e utilizar o valor 255 para indicar que um valor de 32 bits está sendo enviado. É facil ver que teríamos um bom ganho em termos de tamanho de mensagem. No entanto, essa é só uma maneira como isso pode ser feito, sendo que existem formas bem mais eficientes, já que este tem sido alvo de estudos intensos, afinal envolve redução de custos.

Quanto a segurança é também nesta camada que os dados podem ser criptografados de forma a garantir a privacidade dos mesmos. Com isso documentos de real importância poderiam ser enviados por computadores sem que fossem lidos por pessoas não autorizadas.

Quando é feita uma criptografia dos dados, uma unidade de criptografia é inserida entre as duas máquinas, sendo que primeiramente os dados são codificados de forma a serem entendidos por ambas as máquinas e depois são criptografados de forma que só as duas máquinas vão conseguir interpretar os dados.

# *5.2 O Papel da ASN.1 na OSI*

Como citado durante as discussões sobre a camada de apresentação, o principal papel da ASN.1 está ligado a codifição de dados de forma a compatibilizar formatos internos de representação de dados do receptor e do transmissor.

No entanto a ASN.1 poderia ser usada no projeto de qualquer tipo de protocolo, não somente para esta finalidade. Existem várias razões pelas quais a ASN.1 aparece restringida a este papel, no entanto, a mais evidente está relacionada ao fato de que a maioria dos protocolos da OSI já existia antes da padronização da ASN.1. A estrutura da ASN.1, porém, permite que se façam quaisquer tipos de protocolos.

Para entender a importância da abstração imposta pela ASN.1 vamos analizar o que aconteceria com dados transmitidos diretamente de um processador da Motorola para um da Intel onde os bytes aparecem representados em ordem inversa. Para tanto, consideremos um valor inteiro de 32 bits (Intel 386 para Motorola 68030). No processador da Intel o número 43876 apareceria representado em hexadecimal como abaixo:

| 64 | AB | 00 | 00 |
|---|---|---|---|

Enquanto que no processador Motorola seria representado como abaixo:

| 00 | 00 | AB | 64 |
|---|---|---|---|

Com isso, se um processador da Intel fosse o transmissor, ao receber o valor, o processador da Motorola iria interpretá-lo como sendo o valor 1688928256.

Fica óbvio que essa diferença criaria uma grande confusão na comunicação entre as duas máquinas. é nesse ponto que a ASN.1 pode ser utilizada. Poderiamos definir o valor como abaixo:

```
valor INTEGER ::= 43876
```

Assim o valor seria codificado, transmitido, decodificado, e corretamente interpretado.

Um detalhe interessante a ser levado em conta é que se um dado valor é definido de um tipo, e a camada de apresentação recebe um valor de um outro tipo da camada de aplicação, a conexão entre as duas camadas é abortada.

À primeira impressão é fácil imaginar que deveria ser enviada uma mensagem de erro a camada de aplicação. No entanto, devemos lembrar que na transmissão a camada de apresentação somente receberá dados da camada de

aplicação, não podendo enviar nada a mesma.

# *6 Implementação*

Analisada toda a estrutura da ASN.1 e o seu papel na OSI, surge a pergunta sobre como a sintaxe abstrata é efetivamente usada em um protocolo, afinal essa definição abstrata dos dados está longe de ser funcional na prática. Para tanto conclui-se que de alguma forma ela deveria ser compilada a fim de ser usada efetivamente com um protocolo.

A compiliação deve então possibilitar ao usuário da ASN.1 criar, enviar, receber, codificar e decodificar valores de dados.

A maneira usual de implementar tal tarefa é transformar as estruturas declaradas em ASN.1 em um código para alguma linguagem de programação, tipicamente C/C++. O uso de tais compiladores não é foco deste curso, cabendo ao usuário escolher a ferramenta de sua preferência. Vamos, no entanto, dar uma vaga idéia de como esses compiladores funcionam, e para tanto, consideraremos compiladores ASN.1/C++.

Esses programas transformam uma especificação em ASN.1 em um código fonte em C++ e um arquivo **header**(*.h). Cada ítem definido na especificação com "**::=**" se torna uma classe em C++, e os membros da estrutura definida se tornam membros da classe, produzindo uma hierarquia de objetos.

Em paralelo a isso, o compilador deverá dispor ao usuário serviços que permitam a codificação e a decodificação dos valores recebidos, convertam os dados recebidos ou a serem enviados e os colocam em um buffer.

Isto nos dá, enfim, uma idéia de como usar as estruturas que criamos. Assim, podemos projetar, digitar, compilar e finalmente nos utilizarmos das estrutras criadas em ASN.1, já que o compilador nos fornecerá um código fonte em uma determinada linguagem de programação que nos fornecerá os valores codificados (ou decodificados) em um buffer definido. De posse dos dados em um formato conhecido e já adequdo a máquina, sendo essa adequação ao processador feita diretamente pelo compilador em uso. É fácil perceber que essa adequação é feita diretamente, já que o código fonte fornecido pela compilador será fornecido diretamente pela máquina.