### 1.5.3.1. Objetos de Dados

Como já comentado, cada variável possui uma área de armazenamento amarrada a si durante a execução do programa, e o seu o tempo de vida é o período de tempo no qual a amarração existe. Esta área de armazenamento é usada para guardar o valor codificado da variável. O termo objeto de dado é usado para denotar a área de armazenamento em conjunto com o valor da variável [GHE 97].

Sendo assim, um objeto de dado é definido por (*L*, *N*, *V*, *T*), onde *L* é a localização, *N* o nome, *V* o valor e *T* o tipo de objeto. Todos estes componentes devem ser amarrados. A figura 1.6 mostra a visualização de um objeto de dado e suas amarrações, que são representadas por linhas que vão do objeto de dado aos objetos correspondentes aos quais são amarrados.

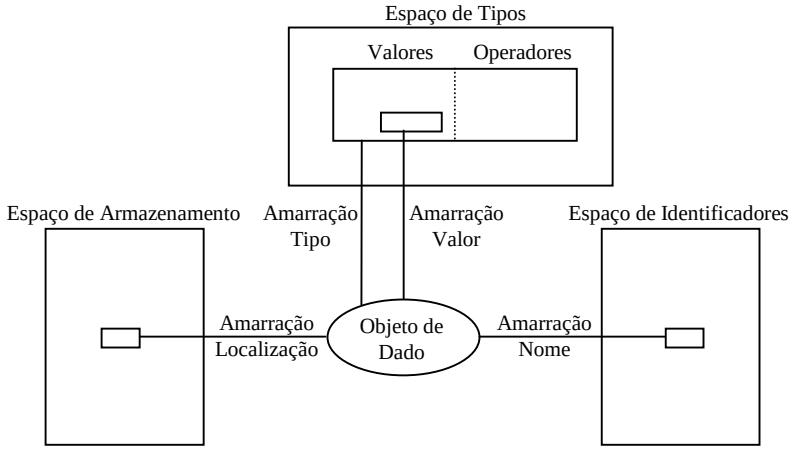


Figura 1.6 – Objeto de dado e suas amarrações [DER 90]

Neste caso, o espaço de armazenamento, que é "invisível" para o programador, consiste no conjunto de localizações virtuais de armazenamento disponíveis no computador. O espaço de identificadores de uma linguagem é a coleção de todos os nomes possíveis que podem ser dados aos objetos de dados. E finalmente, o espaço de tipos é o conjunto de todos os tipos possíveis que podem ser amarrados a um objeto de dado [DER 90]. Na próxima seção, os tipos mais comuns disponíveis nas linguagens estão descritos.

### 1.5.3.2. Tipos de Dados

Programas de computador podem ser vistos como funções que devem ser aplicadas a certos dados de entrada com objetivo de produzir determinados resultados. Essas funções, por sua vez, possuem uma seqüência de instruções que produzem dados intermediários que são armazenados em variáveis de programa. Existem diferenças básicas de uma LP para outra quanto às espécies de dados que usam, às espécies de operações disponíveis sobre os dados e à maneira como os dados podem ser estruturados e usados.

As LP geralmente fornecem um conjunto fixo de tipos de dados elementares (ou embutidos), e mecanismos para definição de tipos de dados mais complexos a partir dos elementares, tais como tipos de dados estruturados. Todas as LP possuem tipos elementares, cujos valores são atômicos, e tipos agregados, que consistem numa composição dos tipos elementares. Algumas linguagens também possuem tipos recursivos, cujos valores podem ser compostos de outros valores do mesmo tipo. Assim, nesta seção os tipos de dados elementares, agregados e recursivos são detalhadamente descritos.

Os **tipos de dados elementares** são aqueles que contêm somente um único valor de dado e sobre o qual são definidas algumas operações. Este tipo de dado é sempre manipulado como uma unidade e não pode ser decomposto em valores mais simples. Assim, os inteiros podem ser encarados como o conjunto de valores ..., -2, -1, 0, 1, 2, ... que podem ser manipulados pelos operadores familiares +, -, * e /. Na verdade, inteiros e reais são suportados por hardware na maioria dos computadores convencionais, que também fornecem aritmética de ponto fixo e ponto flutuante [GHE 97, SIL 88, WAT 90].

Existem tipos de dados elementares similares em diferentes LP com diferentes nomes. Por exemplo: *Pascal* tem os tipos *Boolean*, *Integer* e *Real*; *ML* tem os tipos *bool*, *int* e *real*; e *C++* tem os tipos *bool*, *int* e *float*. Os

nomes não têm muito significado, o que é importante notar é que cada um dos tipos mais comuns, tais como boolean, inteiro, real e caracter, são definidos na implementação; isto significa que eles possuem um conjunto de valores que podem ser definidos diferentemente por cada implementação da LP. Por exemplo, inteiros consistem num intervalo de todos os números possíveis e reais são um subconjunto dos números reais. Também existem limitações de *hardware*, uma vez que os computadores fornecem tamanhos diferentes para os tipos aritméticos suportados. Na linguagem *C* isto pode ser observado pelos prefixos *long* e *short* nas declarações de *int* e *float* [SIL 88, WAT 90, GHE 97].

Em *Pascal*, *Ada* e *C*, é possível um novo tipo elementar através da enumeração dos seus valores, mais precisamente, através da enumeração de identificadores que irão denotar seus valores. Tal tipo é chamado de enumeração. Considere a definição do seguinte tipo em *C* e *Pascal*, respectivamente:

```
enum Month {jan, feb, mar, apr, may, jun,
            jul, aug, sep, oct, nov, dec};
```

```
type Month = (jan, feb, mar, apr, may, jun,
              jul, aug, sep, oct, nov, dec)
```

Os valores para este novo tipo são doze enumerandos, que são distintos dos valores de qualquer outro tipo. É possível notar que deve haver uma distinção entre os enumerandos, que por conveniência foram escritos como "*jan*", "*fev*", etc. O que realmente ocorre é que valores deste tipo podem ser mapeados, um a um, para valores inteiros, no intervalo de 0 a *n*-1, sendo *n* a cardinalidade do enumerador. Em *Pascal* também é possível definir um subconjunto de um tipo existente, através do tipo *subrange*. Por exemplo, o tipo *subrange* '28..31' tem o conjunto de valores {28, 29, 30, 31}, um subconjunto de *Integer* [WAT 90].

Os **tipos de dados agregados** são aqueles cujos valores são compostos ou estruturados a partir dos tipos elementares. Os objetos de dados que fazem parte de um tipo agregado são chamados de componentes. Cada componente pode ser um tipo de dado elementar ou agregado. Os tipos de dados agregados possuem um único nome, mas a manipulação pode ser feita no objeto como um todo ou em um componente de cada vez através de uma operação adequada de seleção.

LP suportam uma grande variedade de tipos de dados agregados: registros, uniões, vetores, *strings*, listas, árvores, etc. Esta variedade pode parecer confusa, mas na verdade estes tipos podem ser entendidos através de alguns conceitos estruturais que são descritos a seguir: a) Produto Cartesiano; b) Mapeamento Finito; c) Seqüências; d) União Discriminada; e) Conjunto Potência [GHE 87, SIL 88, WAT 90].

a) Produto Cartesiano

O produto cartesiano de *n* conjuntos $A_1, A_2, ..., A_n$, denotado por $A_1 \times A_2 \times ... \times A_n$, é um conjunto cujos elementos são *n*-tuplas ordenadas $(a_1, a_2, ..., a_n)$, onde $a_i$ pertence ao conjunto $A_i$. Por exemplo, os polígonos regulares podem ser caracterizados por um inteiro (o número de lados) e um real (o comprimento dos lado). Um polígono regular é então um elemento do produto cartesiano "inteiro x real".

O produto cartesiano é implementado nas linguagens através de agregados do tipo registro, denominado de *record* em *Pascal*, *Cobol*, *Modula-2* e *Ada*, e *struct* em *C*, *Algol* e *PL/I*. Esses agregados são declarados de forma que cada componente seja especificado pelo seu nome e seu tipo de dado. No exemplo anterior, variáveis do tipo polígono podem ser declaradas como compostas de um campo inteiro contendo o número de lados, e um campo real guardando o tamanho de cada lado. Os campos do produto cartesiano são selecionados especificando-se os seletores, ou nomes de campo. A seguinte declaração caracteriza um *record* em *C++* e *Pascal*, respectivamente:

```
struct POLIGONO {
        int num_lados;
        float comprimento_lado;
}
```

```
var POLIGONO: record
                num_lados: integer;
                comprimento_lado: real;
              end;
```

A seleção de um campo é feita segundo a seguinte sintaxe: <nome do objeto>.<nome do campo>. Então, considerando-se o exemplo ficaria "*POLIGONO.num_lados*", e "*POLIGONO.comprimento_lado*".

b) Mapeamento Finito

Uma representação convencional de mapeamento finito consiste em alocar um certo número de unidades de armazenamento (por exemplo, palavras) para cada componente. As linguagens de programação implementam o mapeamento finito através do construtor *array* (ou vetor). As declarações em *C* e *Pascal*

| | |
|---|---|
| ***float*** *a[10];* | ***var*** *a:* ***array*** *[1 .. 10]* ***of*** *real* |

podem ser vistas como um mapeamento do intervalo de inteiros entre 0 e 9, e 1 e 10, respectivamente em *C* e *Pascal*, no conjunto dos reais.

Dependendo da linguagem em questão, um vetor multidimensional pode ser interpretado de duas maneiras: como um mapeamento finito em que o domínio é um produto cartesiano dos índices do vetor ou como uma composição de mapeamentos finitos, ou seja, um vetor de duas dimensões seria um vetor de vetores. O *Fortran* utiliza a primeira alternativa, enquanto muitas outras linguagens (*C, Pascal* e *Ada*) generalizam o conceito do construtor vetor utilizando a segunda interpretação. Apesar de utilizarem a composição de vetores, a maioria das linguagens (*C* é uma exceção) permite que a declaração de um vetor multidimensional seja feita de uma forma condensada equivalente ao mapeamento finito onde o domínio é um produto cartesiano. Por exemplo:

***var*** *a* ***array*** *[1..5]* ***of array*** *[1..10]* ***of real;***

pode ser declarada como

***var*** *a* ***array****[1..5,1..10]* ***of real;***

A seleção de um componente do vetor é feita através de indexação, isto é, fornecendo como índice um valor apropriado no domínio. Assim, por exemplo, o elemento do vetor "*a*", mapeado pelo valor "*i*" no domínio, seria denotado por "*a[i]*". Linguagens que interpretam um vetor multidimensional como uma composição de mapeamentos, mas permitem uma definição condensada, obrigam que a seleção de um elemento seja feita de maneira compatível com a forma de sua definição. Por exemplo, o elemento mapeado pelo primeiro índice com um valor "*i*" e o segundo com um valor "*j*" seria "*a[i][j]*" ou "*a[i,j]*", dependendo de como foi declarado. Deve-se ter muito cuidado ao fazer a indexação, pois, indexar com um valor fora do domínio resulta em um erro que usualmente só pode ser detectado em tempo de execução.

c) Seqüências

Uma seqüência consiste em um número qualquer de ocorrências ordenadas de dados de um certo tipo. Este mecanismo de estruturação tem a propriedade importante de deixar em aberto o número de ocorrências do componente e, portanto, requer que a implementação subjacente seja capaz de armazenar objetos de tamanho arbitrário. Em outras palavras, objetos criados por seqüência devem possuir um tamanho variável, ou seja, são dinâmicos.

Arquivos seqüenciais são um exemplo deste mecanismo. Outra implementação usual corresponde aos *strings*, no qual o tipo do componente é o caracter. Em outras palavras, *strings* são apenas vetores de caracteres. Entretanto, os *strings* implementados nas linguagens nem sempre são dinâmicos ou possuem tamanho arbitrário. Muitas linguagens exigem que em sua definição seja especificado um tamanho, podendo esse tamanho ser interpretado como o número máximo ou o número constante de caracteres que o *string* pode ter. Operações convencionais sobre *strings* incluem concatenação, seleção de um componente ou extração de um *substring* especificando-se as posições do primeiro e do último caracter desejado.

É difícil abstrair um comportamento comum aos exemplos de seqüências fornecidos pelas LP. Por exemplo, *Pascal* e *C* tratam *strings* como vetores de caracteres, sem operações primitivas especiais para sua manipulação. Já *PL/I* e *Ada* fornecem operações primitivas para manipulação de *strings*, mas, para reduzir o problema de alocação dinâmica de memória, exigem que o tamanho máximo de um *string* seja especificado na sua declaração.

d) União Discriminada

A união discriminada, que consiste numa extensão do produto cartesiano, é um mecanismo de estruturação que especifica que deve-se fazer uma escolha entre diferentes estruturas alternativas. Cada estrutura alternativa é chamada de "variante". Exemplos de implementações desse mecanismo são a *union* em *C* e o *variant record* em *Pascal*, que são apresentados a seguir.

```
Struct { int a;
         union { int c;
                 struct { int d; float e; } f;
               } g;
       } x;
```

```
var x: record a: integer;
           case b: boolean of
                   true: (c: integer);
                   false: (d: integer; e: real)
end;
```

Neste exemplo, a seleção de uma alternativa na *union* é feita qualificando-se o objeto com o nome da alternativa selecionada, por exemplo: *x.g.c*"; e no *variant record* é feita atribuindo-se um dos valores possíveis ao campo de seleção "*b*".

e) Conjunto Potência

Em muitas aplicações é útil definir-se variáveis cujos valores representam um subconjunto de um conjunto de valores de um certo tipo T. O tipo dessas variáveis é denominado "conjunto potência" - o conjunto de todos os subconjuntos de elementos do tipo T, o qual é chamado de "tipo base". Por exemplo, considerando o conjunto de cores existentes em um terminal gráfico como sendo: verde, vermelho, azul, branco, amarelo e magenta, os conjuntos {verde, vermelho} e {vermelho, azul, magenta} são cores possíveis de desenho apresentados nesse terminal. Uma variável que representasse cores de desenhos nesse terminal poderia ser declarada em *Pascal* da seguinte maneira:

***var*** *Cores_Desenho:* ***set of*** *(verde, vermelho, azul, branco, amarelo, magenta);*

As operações permitidas sobre tais valores são operações de conjuntos, tais como união, interseção e teste de pertinência de um elemento [GHE 87, SIL 88].

Apesar de conjuntos (e conjuntos potência) fazerem parte do conceitos matemáticos básicos, somente algumas linguagens, principalmente *Pascal* e *Modula-2*, fornecem este tipo. Na maioria das outras linguagens eles são fornecidos através de bibliotecas. Por exemplo, a biblioteca padrão do *C++* fornece muitas estruturas de dados, incluindo conjuntos [GHE 97].

Já um **tipo de dado recursivo** T contém em sua definição componentes do mesmo tipo T. Para se definir um tipo recursivo pode-se usar o nome do próprio tipo na sua definição. Por exemplo, o tipo "árvore_binária" pode ser definido ou como sendo vazio ou como uma tripla composta de um elemento atômico (raiz), uma "árvore_binária à esquerda" e uma "árvore_binária à direita".

A recursão é um forte mecanismo de estruturação implementado nas LP. LP convencionais suportam a implementação de tipos de dados recursivos através de ponteiros. Assim, através da recursão pode-se representar estruturas dinâmicas complexas, tais como árvores e listas encadeadas, de uma forma mais elegante. No caso de listas o número de componentes é indeterminado, podendo inclusive ser nulo, isto é, a lista pode estar vazias. Uma lista é considerada homogênea, se todos os seus componentes são do mesmo tipo. Assim, agregados cujo tamanho pode crescer arbitrariamente e cuja estrutura pode ter complexidade arbitrária, também podem ser definidos. Em contraste com o mecanismo de seqüências, a recursão permite ao programador criar caminhos de acesso arbitrários para a seleção de componentes [GHE 87, SIL 88, WAT 90, GHE 97]:

Os pedaços de código em *C/C++* e *Ada* a seguir, definem um tipo de uma lista de inteiros e uma variável que pode apontar para o cabeçalho da lista. Implementações similares de tipos recursivos podem ser fornecidas em *Pascal* e *Modula-2* [GHE 97].

```
typedef struct {
        int val;
        int_list* next;
} int_list;

int_list* head;
```

```
type INT_LIST_NODE;
type INT_LIST_REF is access INT_LIST_NODE;
type INT_LIST_NODE is
    record
        VAL: INTEGER;
        NEXT: INT_LIST_REF;
    end;
HEAD: INT_LIST_REF;
```

Para finalizar, deve-se salientar que existe um conjunto de regras usado por algumas linguagens para estruturar e organizar os seus tipos de dados, cujo entendimento é um grande passo no entendimento da semântica da linguagem. Antes da apresentação destas regras, é importante saber que os erros podem ser classificados em duas categorias: erros de linguagem e erros de aplicação. O primeiro corresponde a erros sintáticos e semânticos no uso da LP. O segundo correspondem a desvios do comportamento do programa em relação às especificações do programa. A **detecção de erros** pode ser acompanhada de diferentes maneiras, que também podem ser classificadas em duas categorias: **estática** e **dinâmica**. A verificação dinâmica requer que o programa seja executado com entrada de dados, e a estática não. Em geral, é preferível que uma verificação seja executada estaticamente (ou em tempo de compilação), ao invés de esperar para ser feita em tempo de execução [GHE 97].

Um linguagem que permite que todos os testes de tipo sejam efetuados estaticamente, é dita **fortemente tipada**. Neste caso, o compilador pode garantir a ausência de erros de tipo nos programas. *Algol 68* é um exemplo de uma linguagem fortemente tipada, já o *Pascal* não é uma linguagem fortemente tipada, pois, por exemplo, em *Pascal* intervalos e o emprego correto dos registros com variantes não podem ser testados estaticamente.

As regras de **compatibilidade de tipos**, por sua vez, devem dar uma especificação exata de como o mecanismo de testes de tipos deve ser aplicado. É possível definir duas noções de compatibilidade de tipos:

- **Equivalência nominal**: duas variáveis possuem tipos compatíveis se possuem o mesmo nome de tipo, definido pelo usuário ou primitivo, ou se aparecem na mesma declaração.
- **Equivalência estrutural**: duas variáveis possuem tipos compatíveis se possuem a mesma estrutura. Para se verificar a equivalência estrutural, os nomes dos tipos definidos pelo usuário são substituídos pelas suas definições. Este processo é repetido até não sobrarem mais nomes de tipos definidos pelo usuário. Os tipos são então considerados estruturalmente equivalentes se possuem exatamente a mesma descrição.

Para ilustrar as regras de compatibilidade de tipos considere o segmento de código a seguir escrito numa LP hipotética:

```
type s1 is struct {
    int y;
    int w;
};
type s2 is struct {
    int y;
    int w;
};
type s3 is struct {
    int y;
};
s3 func(s1 z) { ... };
...
s1 a, x;
s2 b;
s3 c;
int d;
...
a = b;   --(1)
x = a;   --(2)
c = func(b);    --(3)
d = func(a);   --(4)
```

De acordo com a equivalência nominal a instrução (2) está correta, desde que "*a*" e "*x*" possuem o mesmo nome de tipo. A instrução (1) contém um erro porque "*a*" e "*b*" possuem tipos diferentes. Similarmente, as instruções (3) e (4) contêm erros de tipo. Considerando a equivalência estrutural, as instruções (1), (2) e (3) estão corretas, e a instrução (4) contém um erro, pois o tipo "*s3*" não é compatível com *int*.

Freqüentemente também é necessário converter um valor de um tipo em um valor de outro, como, por exemplo, quando se quer somar uma variável inteira à uma constante real. Na maior parte das linguagens essa **conversão é implícita**. A conversão implícita é tornada explícita pelo tradutor, que gera código para conversão baseado no tipo dos operadores e na hierarquia de tipos da linguagem. Esta conversões automáticas, seguindo a terminologia do *Algol 68*, são chamadas de coerção. Entretanto, algumas linguagens permitem apenas a **conversão explícita**, isto é, o programador deve usar operadores de conversão.

Em geral, o tipo de coerção que pode ocorrer em um determinado ponto depende do contexto. Por exemplo, na seguinte instrução em *C*:

```
x = x + z;
```

onde "*z*" é um *float* e "*x*" é um *int*, há uma coerção de "*x*" para *float* para avaliar a aritmética do operador de soma, e há uma coerção do resultado para *int* para fazer a atribuição. Pode-se observar, então, que o *C* fornece um sistema simples de coerção. Além disso, conversões explícitas podem ser aplicadas em *C* usando o construtor *cast*. Por exemplo, um *cast* pode ser usado para sobrescrever a coerção não desejada que seria aplicada. Assim, o comando acima ficaria:

```
x = x + (int) z;
```

*Ada* não fornece coerção. Sempre que uma conversão é permitida através da linguagem, ela deve ser invocada explicitamente. Por exemplo, se "*X*" é declarado como uma variável *FLOAT* e "*I*" como um *INTEGER*, a atribuição de "*X*" para "*I*" pode ser realizada pela seguinte instrução:

```
I := INTEGER(X);
```

A função de conversão *INTEGER* fornecida por *Ada* processa um inteiro a partir de um valor de ponto flutuante arredondando para o inteiro mais próximo [GHE 87, GHE 97].