1.5.2. Semântica

Enquanto a sintaxe define se os programas estão corretamente escritos numa linguagem, a semântica define o significado dos programas sintaticamente corretos. Por exemplo, a semântica do *C* ajuda a determinar que a declaração

int *vetor[10]**;***

faz com que sejam reservados dez espaços para elementos inteiros, para uma variável chamada vetor. Os elementos do vetor podem ser referenciados por um índice i, $0 \leq i \leq 9$, sendo o primeiro deles vetor[0]. Outro exemplo, a semântica do *C* define que a instrução

if (*a > b**) max = a**;*** ***else*** *max = b**;***

significa que a expressão "a>b" deve ser avaliada e, dependendo do seu valor, um dos dois comandos de atribuição é executado. Assim, é possível observar que as regras sintáticas mostram como formar o comando e as regras semânticas mostram qual é o efeito do comando.

Na verdade, nem todos os programas sintaticamente corretos possuem um significado. Sendo assim, a semântica também separa os programas com significado daqueles que são apenas sintaticamente corretos. Alguns exemplos de erros semânticos: acesso a uma posição inválida de um vetor, atribuição de uma variável do tipo inteira para uma variável do tipo *string*, e utilização de variável fora do escopo. Por exemplo, de acordo com a EBNF do exemplo descrito na figura 1.2, é possível escrever qualquer expressão como uma condição de comandos *if* e *while*. A semântica da linguagem deve exigir que tais expressões retornem um valor, verdadeiro ou não. Em muitos casos, tais regras que restringem programas sintaticamente corretos podem ser verificadas antes da execução do programa; estas são chamadas de **semântica estática**, ao contrário da **semântica dinâmica**, que descreve o efeito da execução das diferentes construções da linguagem. Em tais casos, programas podem ser executados somente se eles estão corretos em relação à sintaxe e à semântica estática.

Alguns conceitos semânticos básicos nas LP são:

- Variáveis: uma variável, que é referenciada por um nome, corresponde a uma região da memória usada para guardar valores que são manipulados pelo programa. Questões semânticas associadas com variáveis envolvem sua declaração. Algumas propriedades semânticas da declaração são: escopo (em qual(is) parte(s) do programa a variável pode ser acessada), tipo (que tipos podem ser armazenados na variável e quais operações podem ser feitas com ela), e o tempo de vida da variável (quando a variável foi criada, isto é, se há uma área de memória reservada, ou alocada, para variável).
- Valores e referências: envolve a definição de que valor está associado com a variável, se é um valor que denota a localização da memória ou o conteúdo da localização da memória.
- Expressões: possuem algumas regras semânticas para serem escritas envolvendo, principalmente, os tipos de expressões permitidas.

Para finalizar, torna-se interessante comentar que enquanto diagramas de sintaxe e EBNF tornaram-se ferramentas padrão para descrição sintática, não há uma ferramenta que tenha se tornado padrão para descrição semântica. Em outras palavras, existem diferentes abordagens formais para definição semântica, mas nenhuma é inteiramente satisfatória. Uma metalinguagem para semântica formal deve se basear em conceitos matemáticos bem definidos, para que a definição resultante seja rigorosa e não ambígua. A habilidade de fornecer semântica formal torna as definições da linguagem independentes da implementação. Neste caso, a descrição especifica o que a linguagem faz sem qualquer referência a como isto pode ser alcançado através da implementação [GHE 97].

1.5.3 Abstração e Amarração

Abstração consiste no processo de identificar apenas as qualidades ou propriedades relevantes do fenômeno que se quer modelar. Usando o modelo abstrato, pode-se concentrar unicamente nas qualidades ou propriedades relevantes e ignorar as irrelevantes. Porém, a determinação do que é relevante depende da finalidade para a qual a abstração está sendo projetada. A abstração permeia toda a programação. Em particular, apresenta uma dupla relação com as linguagens de programação. Por um lado, as linguagens de programação são as

ferramentas com as quais os programadores podem implementar os modelos abstratos. Por outro lado, as próprias linguagens de programação são abstrações do processador subjacente, onde o modelo é implementado [GHE 87].

Na programação, a abstração sugere a distinção que deve ser feita entre “o que” o programa faz e “como” ele é implementado. Por exemplo, quando um procedimento é chamado, pode-se concentrar somente no que ele faz; apenas quando se está escrevendo o procedimento é que deve-se concentrar em como implementá-lo. Através da implementação de procedimentos de alto nível, que podem chamar procedimentos de nível inferior, os programadores podem introduzir tantos níveis de abstração quanto desejado. Este tipo de hierarquia é uma ferramenta essencial na construção de grandes sistemas [WAT 90].

A primeira abstração introduzida, nos primórdios da computação, foi o uso de mnemônicos em *assembly* tanto para as operações do computador como para localizações de memória. Com o surgimento das linguagens de alto nível, a primeira abstração apresentada foi o conceito de variáveis como uma representação de localizações de memória. Associado ao conceito de variáveis, existe o conceito de tipo que representa a classe de valores que essas variáveis podem assumir e um conjunto de operações para manipular cada tipo de variável. Nessa primeira fase de evolução da abstração de dados, as linguagens possuíam alguns tipos elementares, ou embutidos, nos quais todos os dados da aplicação e suas operações tinham de ser mapeados. Uma primeira tentativa de se aumentar a capacidade de abstração das linguagens, foi aumentar o número de tipos embutidos na linguagem [SIL 88].

Assim, a abstração pode ser definida como uma entidade que engloba o processamento. Por exemplo, uma abstração de função contém uma expressão para ser avaliada, que quando chamada produzirá um valor resultante; e uma abstração de procedimento contém um comando a ser executado, que quando chamado irá atualizar as variáveis. Em outras palavras, o princípio da abstração diz que é possível construir abstrações sobre qualquer classe sintática [WAT 90].

A habilidade de uma LP expressar e usar a abstração é muito importante, tanto em relação a dados como em procedimentos. Com dados, o programador está apto a trabalhar mais efetivamente usando simples abstrações que não incluem muitos detalhes irrelevantes dos objetos de dados. Com procedimentos, as abstrações facilitam práticas adequadas de projeto e modularidade. O nível no qual a linguagem oculta os detalhes irrelevantes na criação e uso de abstrações é importante para a sua habilidade de efetivamente dar suporte ao projeto, implementação e modificação de programas [DER 90].

É importante ressaltar que os programas trabalham com entidades, tais como variáveis, rotinas e comandos. As entidades dos programas, por sua vez, possuem certas propriedades chamadas atributos. Por exemplo: uma variável tem um nome, um tipo e um valor; e uma rotina tem um nome, parâmetros formais de um determinado tipo e certas convenções de passagem de parâmetros. Os valores dos atributos devem ser definidos antes que eles possam ser usados. A definição do valor de um atributo é conhecida como **amarração**, ou *binding* (figura 1.5). Para cada entidade, a informação do atributo está contida em um repositório chamado descritor.

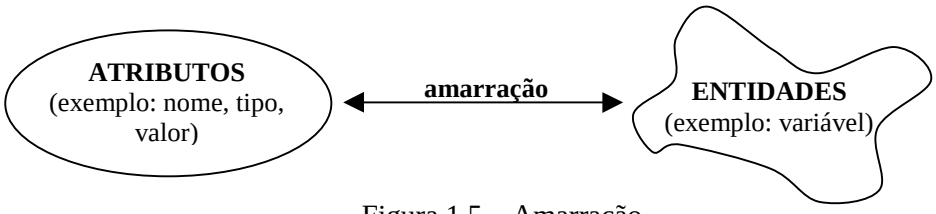

Figura 1.5 – Amarração

A amarração consiste em um conceito central na definição da semântica da linguagem de programação. LP diferem no número de entidades com as quais elas podem lidar, no número de atributos a serem amarrados às entidades, no tempo no qual as amarrações ocorrem (*binding time*), e na estabilidade da amarração, isto é, quando a determinação de uma amarração é fixa ou pode ser alterada. Alguns atributos podem ser amarrados no tempo de definição da linguagem, outros no tempo de implementação da linguagem, outros no tempo de tradução do programa (ou tempo de compilação), e outros ainda no tempo de execução do programa (ou *run time*). A seguir é apresentada uma lista com exemplos de amarrações:

- Amarração no tempo de definição da linguagem: na maioria das linguagens, incluindo *Fortran*, *Ada*, *C* e *C++*, o tipo inteiro é amarrado no tempo de definição da linguagem ao conjunto de operações algébricas que produzem e manipulam inteiros.
- Amarração no tempo de implementação da linguagem: na maioria das linguagens, incluindo *Fortran*, *Ada*, *C* e *C++*, um conjunto de valores é amarrado ao tipo inteiro no tempo de implementação da linguagem. Em outras palavras, a definição da linguagem especifica que o tipo inteiro deve ser suportado e a implementação da linguagem amarra-o à representação da memória, que por sua vez determina o conjunto de valores que estão contidos no tipo.
- Amarração no tempo de compilação: *Pascal* fornece uma pré-definição do tipo inteiro, mas permite que o programador redefina-a. Assim, o tipo inteiro é amarrado à representação em tempo de implementação, mas a amarração pode ser alterada no tempo de compilação.
- Amarração no tempo de execução: Na maioria das LP as variáveis são amarradas a um valor em tempo de execução, e a amarração pode ser alterada repetidamente durante a execução.

Nos três primeiros exemplos, a amarração é estabelecida antes de tempo de execução e não pode ser alterada depois. Este tipo de amarração é geralmente chamada de **amarração estática**. O termo estática denota tanto o tempo de amarração (que ocorre antes do programa ser executado), como a estabilidade (a amarração não pode ser alterada). Em oposição, uma amarração estabelecida em tempo de execução é geralmente alterável durante a execução, o que é ilustrado no quarto exemplo. Este tipo de amarração é geralmente chamada de **amarração dinâmica**. Existem casos, entretanto, onde a amarração é estabelecida em tempo de execução e não pode ser alterada depois de ter sido estabelecida. Um exemplo, consiste nas linguagens que fornecem variáveis constantes (somente de leitura) que são inicializadas com uma expressão a ser avaliada em tempo de execução [GHE 97].

Os conceitos de amarração, tempo de amarração e estabilidade ajudam a clarear muitos aspectos semânticos das LP. Isto ocorre porque as linguagens, basicamente, diferenciam-se pelo número de entidades que manipulam, pelo número de atributos que são amarrados a essas entidades e pelo momento em que essa amarração ocorre [GHE 97, SIL 88].

Computadores convencionais se baseiam no conceito de uma memória principal que consiste em células elementares, cada qual identificada por um endereço. O conteúdo de uma célula contém seu valor, que pode ser lido e alterado pela substituição de um valor por outro. Além disso, circuitos internos permitem o acesso a uma célula de cada vez. Com poucas exceções, LP podem ser consideradas como abstrações, em níveis diferentes, do comportamento destes computadores convencionais. Em particular, o conceito de **variável** é introduzido como uma abstração de células de memória. Em outras palavras, uma variável corresponde a região da memória que é usada para armazenar valores que são manipulados pelo programa [GHE 87].

Como já citado, por exemplo, uma variável é caracterizada por um nome e quatro atributos básicos: escopo, tempo de vida, valor (ou conteúdo) e tipo. O nome é usado para identificar e fazer referência à variável. O escopo de uma variável é o trecho de programa onde a variável é conhecida e, portanto, pode ser usada. A variável é dita visível dentro do seu escopo e invisível fora dele. A amarração de variáveis a escopos pode ser feita estática ou dinamicamente:

- Amarração estática: escopo é definido por regras da linguagem, em geral, função da estrutura sintática do programa, sendo determinado em tempo de compilação; em outras palavras, cada referência a uma variável é estaticamente amarrada a uma declaração (implícita ou explícita) desta variável.
- Amarração dinâmica: escopo é definido em tempo de execução; as declarações são válidas até que surja um novo contexto para o mesmo nome da variável; neste caso a desvantagem é que variáveis com o mesmo nome podem ter tipos distintos durante a execução do programa, o que dificulta a legibilidade do programa.

O tempo de vida de uma variável significa o intervalo de tempo durante o qual uma área da memória, usada para guardar o valor da variável, está amarrada a uma variável. A alocação da área de memória pode ser estática, quando realizada antes da execução do programa, ou dinâmica, quando realizada durante a execução.

O valor de uma variável é a interpretação do conteúdo armazenado na área alocada à variável. Esse conteúdo é armazenado de maneira codificada e depende do tipo da variável. Além disso, em algumas LP, o valor de uma variável pode ser uma referência (ponteiro) para um objeto. Normalmente, a amarração entre uma variável e o valor armazenado na área de memória correspondente é dinâmica, sendo feita em tempo de execução por um comando de atribuição. Algumas linguagens, entretanto, permitem o "congelamento" da amarração entre uma variável e seu valor, quando a amarração é estabelecida. A entidade resultante é uma constante definida pelo programador.

Finalmente, o tipo de uma variável é caracterizado pela especificação de uma classe de valores que podem ser associados a uma variável, juntamente com as operações que podem ser usadas para criar, acessar e modificar tais valores. A amarração do tipo à variável pode ser feita estática ou dinamicamente. Na maioria das linguagens, como *Pascal*, *Algol*, *C*, *Ada*, *Fortran*, *Cobol* e *PL/I*, a amarração é estática. Nestas linguagens, a amarração é feita estaticamente mediante um comando de declaração. As declarações do tipo de uma variável podem ser explícitas (como por exemplo em *Pascal*) ou implícita (como por exemplo em *Fortran*, onde a primeira ocorrência da variável pode determinar seu tipo). As declarações explícitas permitem maior clareza e confiabilidade dos programas.

No caso de amarração dinâmica, o tipo é implicitamente determinado pelo valor atribuído à variável. Por exemplo, em *APL*, uma variável pode receber valores diferentes, de diferentes tipos, numa mesma execução. A amarração dinâmica de tipos permite maior flexibilidade para a criação e manipulação de estruturas de dados, porém apresenta desvantagens na legibilidade de programas. São necessários descritores de tipo em tempo de execução, os quais são modificados sempre que uma nova amarração é estabelecida. No caso de amarração estática, existem descritores somente em tempo de tradução.

Um programa é composto por **unidades de programa** que podem ser desenvolvidas até certo ponto independentemente, e podem algumas vezes ser traduzidas em separado e combinadas depois da tradução. As variáveis declaradas dentro de uma unidade são locais à unidade, que pode ser ativada durante a execução do programa. Sub-rotinas de *Fortran* e funções do *C* são alguns exemplos conhecidos de unidades de programa. A cada unidade de programa é associada uma unidade de ativação, que são as representações de uma unidade em tempo de execução. As unidades de ativação são compostas por duas partes:

- Registro de ativação: é a parte da unidade de ativação que contém as informações necessárias à execução da unidade de programa. Entre essas informações, pode-se citar a área de armazenamento das variáveis locais e o endereço de retorno após a desativação da unidade.
- Segmento de código: é a parte da unidade de ativação que contém as instruções da unidade; possui, portanto, um conteúdo fixo.

Uma unidade de programa pode utilizar objetos definidos internamente (locais) ou definidos externamente (globais), desde que estes sejam visíveis dentro da unidade. Objetos locais são definidos na unidade e, portanto, correspondem a objetos armazenados no registro de ativação da própria unidade (ambiente local). Objetos globais são definidos externamente à unidade, mas visíveis dentro da unidade. Correspondem a objetos armazenados em registros de ativação de outras unidades que formam o seu ambiente global. Para acessar um objeto durante a execução de um programa, o processador pode usar o endereço inicial do registro de ativação que contém o objeto e o deslocamento do objeto (*offset*), ou seja, qualquer objeto é identificado por um registro de ativação e pelo seu deslocamento dentro do registro de ativação.

Unidades podem muitas vezes ser ativadas recursivamente, isto é, uma unidade pode chamar a ela própria diretamente, ou através de uma outra unidade Em outras palavras, a recursividade, consiste em uma nova ativação de uma unidade antes do término de uma ativação anterior da mesma unidade. Todas as ativações de uma mesma unidade são compostas do mesmo segmento de código e de registros de ativação diferentes. Assim, quando existe recursão, a amarração entre o registro de ativação e o segmento de código correspondente é necessariamente dinâmico. A cada vez que uma unidade é ativada, uma amarração deve ser estabelecida entre um registro de ativação e seu segmento de código, formando desta maneira uma nova ativação da unidade [GHE 87, SIL 88].