1.5. Elementos Sintáticos e Semânticos

Como já comentado (seção 1.2) os componentes gerais de uma linguagem são a sua sintaxe e a sua semântica. A sintaxe de uma linguagem influencia na maneira como os programas são escritos pelo programador, lidos por outros programadores e reconhecidos pelo computador. A semântica de uma linguagem determina como os programas são resolvidos pelos programadores, entendidos por outros programadores e interpretados pelo computador [WAT 90].

1.5.1 Sintaxe

Como já comentado, a sintaxe consiste num conjunto de regras que definem a forma da linguagem , isto é, como as sentenças podem ser formadas como seqüências de componentes básicos, chamados **palavras**. Usando estas regras, pode-se identificar quando uma sentença está correta ou não. A sintaxe não revela nada sobre o conteúdo (ou significado) da sentença. Por exemplo, na linguagem C, palavras chave (tais como *while*, *do*, *if* e *else*), identificadores, números, operadores, etc. são palavras da linguagem. A sintaxe desta linguagem também diz como combinar tais palavras para construir comandos e programas corretamente.

Entretanto, palavras não são elementares, elas são construídas com caracteres que pertencem a um **alfabeto**. Assim, a sintaxe de uma linguagem é definida por dois conjuntos de regras: regras léxicas e regras sintáticas. As **regras léxicas** especificam o conjunto de caracteres que constituem o alfabeto da linguagem e a maneira como os caracteres podem ser combinados para formar palavras válidas.

Por exemplo, no *Pascal* letras maiúsculas e minúsculas são idênticas, mas em *C* e *Ada* elas são diferenciadas. Desta forma, de acordo com as regras léxicas, “Memória” e “memória” referem-se à mesma variável em *Pascal*, mas a variáveis diferentes em *C* e *Ada*. As regras léxicas também identificam que “<>” (ou ≠) é um operador válido em *Pascal* mas não em *C*, onde o mesmo operador é representado por “!=”. *Ada* difere dos dois, uma vez que “não igual” é representado por “/=:”.

As **regras sintáticas**, por sua vez, especificam as seqüências de símbolos que constituem estruturas sintáticas válidas, e são verificadas através de uma varredura (*parsing*) da representação interna do programa fonte. Estas regras permitem, por exemplo, o reconhecimento de expressões e comandos. Para ilustrar, a sintaxe do *Pascal* determina que o comando de atribuição entre duas variáveis é “a := b;”, e do *C* é “a = b;”. Na verdade, a distinção entre regras sintáticas e léxicas é um tanto arbitrária, uma vez que ambas contribuem para a aparência “externa” da linguagem. Aqui, muitas vezes, os termos sintaxe e regras sintáticas são usados num sentido amplo que inclui os componentes léxicos.

Como existe um grande número programas sintaticamente corretos ou não em várias linguagens, não é possível enumerar a sintaxe de todos eles. É necessário apenas uma maneira de definir um conjunto infinito usando uma descrição finita [GHE 97]. Para alcançar este objetivo, a sintaxe de uma linguagem é definida através de uma Gramática, que é um conjunto de regras que definem todos os construtores que podem ser aceitos na linguagem [DER 90].

Para exemplificar, *Fortran* foi definido simplesmente através da especificação de algumas regras em inglês. *Algol 60* foi definido através de uma gramática livre de contexto desenvolvida por John Backus, que ficou conhecida como BNF (*Backus-Naur Form*). BNF, que foi usada posteriormente na definição de várias linguagens de programação, incluindo Pascal, C e Ada, fornece uma definição compacta e clara para a sintaxe de linguagens de programação. Todo programador deve saber como ler, interpretar e aplicar descrições BNF da sintaxe das linguagens. BNF ocorre com variações textuais menores em três formas básicas: BNF original, BNF estendida e diagrama de sintaxe, que fornecem outras maneiras de definir a sintaxe de LP. Os diagramas são conceitualmente equivalentes a BNF, mas sua notação é mais intuitiva [GHE 97, LOU 93].

BNF é uma “metalinguagem”, pois consiste numa linguagem para descrição de outras linguagens, mais especificamente de gramáticas. Inicialmente BNF será descrita através de um exemplo. Em inglês, sentenças simples consistem de uma *noun phrase* e de uma *verb phrase* seguida de um ponto. Isto pode ser expresso da seguinte maneira:

1. <*sentence*> ::= <*noun-phrase*> <*verb-phrase*>.

Por outro lado deve-se saber descrever a estrutura de uma frase nominal e de uma frase verbal:

2. <*noun-phrase*> ::= <*article*> <*noun*>
3. <*article*> ::= *a* | *the*
4. <*noun*> ::= *girl* | *dog*
5. <*verb-phrase*> ::= <*verb*> <*noun-phrase*>
6. <*verb*> ::= *sees* | *pets*

Cada uma das regras gramaticais apresentadas consiste de um *string* colocado entre "<" e ">" (nome da estrutura que está sendo descrita), seguida pelo símbolo "::=" que pode ser lido como "consiste de" ou "é o mesmo que", e uma seqüência de outros nomes e símbolos. Os sinais maior e menor servem para distinguir os nomes das estruturas das palavras que podem aparecer na linguagem. Por exemplo, em Pascal, "<*program*>" irá representar toda estrutura do programa, enquanto a palavra "*program*" é o primeiro símbolo da linguagem: <*program*> ::= *program* .....

O símbolo "::=" é um metasímbolo que serve para separar o lado esquerdo do lado direito de uma regra. Os sinais ">", "<" e "|", que significa "ou", também são metasímbolos. Assim, no exemplo anterior, a regra 6 significa que *verb* pode ser tanto a palavra "*sees*" como a palavra "*pets*". Algumas vezes um metasímbolo é também um símbolo atual em uma linguagem. Neste caso, o símbolo pode ser colocado entre aspas para ser distinguido do metasímbolo. Freqüentemente isto é feito para símbolos especiais, tais como pontos, mesmo quando eles não são metasímbolos. No exemplo anterior, na regra 1, poderia-se ter: <*sentence*> ::= <*noun-phrase*> <*verb-phrase*> "." (é claro que neste caso as aspas tornam-se metasímbolos).

Cada sentença legal, de acordo com a gramática anterior, pode ser construída da seguinte maneira: inicia-se com o símbolo <*sentence*> e prossegue-se trocando o lado esquerdo por alternativas do lado direito nas regras. Este processo cria uma derivação na linguagem. Assim, pode-se construir a sentença "*the girl sees a dog*." da seguinte maneira:

<*sentence*> → <*noun-phrase*> <*verb-phrase*>.	(regra 1)
→ <*article*> <*noun*> <*verb-phrase*>.	(regra 2)
→ *the* <*noun*> <*verb-phrase*>.	(regra 3)
→ *the girl* <*verb-phrase*>.	(regra 4)
→ *the girl* <*verb*> <*noun-phrase*>.	(regra 5)
→ *the girl sees* <*noun-phrase*>.	(regra 6)
→ *the girl sees* <*article*> <*noun*>.	(regra 2)
→ *the girl sees a* <*noun*>.	(regra 3)
→ *the girl sees a dog*.	(regra 4)

De maneira inversa, pode-se começar com a sentença "*the girl sees a dog*." e "voltar" até <*sentence*> para provar que é uma sentença válida na linguagem.

De acordo com a descrição anterior, conclui-se que para descrever uma linguagem é necessário uma série de regras gramaticais. Tais regras são formadas por uma única estrutura do lado esquerdo, seguida do metasímbolo "::=", e por uma seqüência de itens do lado direito que podem ser símbolos ou outras estruturas. As estruturas que aparecem entre "<>" são chamadas não terminais. As palavras e símbolos, tais como *girl* no exemplo anterior, são os terminais, e as regras gramaticais são as produções.

Logo a seguir é apresentado um exemplo simples de uma gramática para expressões aritméticas de adição e multiplicação. Nesta gramática os símbolos não-terminais são <*exp*>, <*number*> e <*digit*>, e os terminais são 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (e) [LOU 93].

<*exp*> ::= <*exp*> + <*exp*> | <*exp*> * <*exp*> | (<*exp*>) | <number>
<*number*> ::= <*number*> <*digit*> | <*digit*>
<*digit*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A partir de agora será apresentada uma extensão da BNF, chamada EBNF (*Extended BNF*), e como ela pode ser usada para descrever a sintaxe de uma linguagem de programação simples exibida na figura 1.2a. Os símbolos ::=, <, >, |, e * e + sobrescritos, que aparecem na figura 1.2, são símbolos da metalinguagem, e, como já visto, são chamados "metasímbolos".

A linguagem é descrita em EBNF através de um conjunto de regras. Por exemplo, "<*program*>*::={*<*statement*>**}*" é uma regra. O símbolo "*::=*" significa "é definido por". O símbolo "*" significa "zero ou mais ocorrências do elemento precedente". Assim, a regra estabelece que um "<*program*>" é definido como uma seqüência arbitrária de "<*statement*>" entre chaves, "*{*" e "*}*". As entidades que aparecem entre "<" e ">" são chamadas não terminais; e entidades como "*{*" são chamadas terminais. Terminais são as palavras da linguagem que está sendo definida, enquanto não terminais são entidades que são definidas por outras regras EBNF. O metasímbolo "$^{+}$" denota "uma ou mais ocorrências do elemento precedente", e "|" denota uma escolha. Já as regras léxicas que fornecem a descrição EBNF de identificadores, números e operadores são mostradas na figura 1.2b. Neste caso, "<*operator*>", "<*identifier*>" e "<*number*>", que são palavras da linguagem que está sendo definida, são detalhados em termos de símbolos elementares do alfabeto [GHE 97]. Para ilustrar os conceitos apresentados, a figura 1.3 mostra a definição de uma calculadora em EBNF [DER 90].

A figura 1.4 mostra o diagrama de sintaxe equivalente à linguagem de programação simples cujas regras sintáticas em EBNF são apresentadas na figura 1.2. Agora, não terminais são representados por retângulos e terminais por círculos. O símbolo não terminal é definido como um diagrama de transição tendo uma aresta de entrada e uma de saída. Um *string* de palavras é um programa válido se ele pode ser gerado ao se percorrer o diagrama de sintaxe a partir da aresta de entrada até a aresta de saída. Neste "percurso", se um terminal (círculo) é encontrado, esta palavra deve estar no *string* que está sendo reconhecido; se um não terminal (retângulo) é encontrado, então o não terminal deve ser reconhecido ao se percorrer o diagrama de transição para aquele não terminal. Quando uma "ramificação" é encontrada, qualquer aresta pode ser percorrida, isto é, os diferentes "caminhos" representam as possíveis seqüências de símbolos.

(a) Regras Sintáticas

<program> ::= { <statement>* }
<statement> ::= <assignment> | <conditional> | <loop>
<assignment> ::= <identifier>=<expr>;
<conditional> ::= ***if*** *<expr> { <statement>$^{+}$ } |*
*****if*** *<expr> { <statement>$^{+}$ }* ***else*** *{ <statement>$^{+}$ }*
<loop> ::= ***while*** *<expr> { <statement>$^{+}$ }*
<expr> ::= <identifier> | <number> | (<expr>) |
<expr> <operator> <expr>

(b) Regras Léxicas

<operator> ::= **+** | **-** | ***** | **/** | **=** | **≠** | **<** | **>** | **≤** | **≥**
<identifier> ::= <letter> <ld>*
<ld> ::= <letter> | <digit>
<number> ::= <digit>$^{+}$
<letter> ::= a | b | c | ... | z
<digit> ::= 0 | 1 | ... | 9

Figura 1.2 – Definição EBNF para uma linguagem de programação simples [GHE 97]

<calculation> ::= <expression> =
<expression> ::= <value> | <value> <operator> <expression>
<value> ::= <unsigned> | <sign> <unsigned>
<unsigned> ::= <digit>$^{+}$ *| <digit>*$^{+}$ *. <digit>*$^{+}$
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<sign> ::= + | -
*<operator> ::= + | - | * | /*

Figura 1.3 – Definição EBNF para uma calculadora

As vantagens do diagrama de sintaxe são a utilização de duas dimensões para facilitar o entendimento, e a similaridade com a EBNF para permitir o entendimento das regras. Sua desvantagem é a dificuldade em gerar os diagramas usando um dispositivo de entrada linear, tal como o teclado. As gramáticas das linguagens que podem ser descritas por BNF ou EBNF são conhecidas como gramáticas livres de contexto (seção 1.5.5) [GHE 97, DER 90].

Resumindo, a descrição sintática de uma linguagem tem duas aplicações principais:

- Ajuda o programador a saber como escrever um programa sintaticamente correto;
- Pode ser usada para determinar se um programa está sintaticamente correto, que é exatamente o que o compilador faz.

É interessante atentar para o fato de que algumas construções em diferentes linguagens de programação possuem a mesma estrutura conceitual, mas diferem na aparência a nível léxico. Por exemplo, analisado os seguintes pedaços de código:

Em *C:*	Em *Pascal:*
while (*x != y* ***) {***	***while*** *x <> y* ***do***
...	***begin***
}	*...*
	end

Ambos podem ser descritos através de simples variantes léxicas de acordo com as regras EBNF. Eles diferem apenas na maneira como os comandos são agrupados (*begin...end* X *{...}*), no operador "não igual" (<> X *!=*) e no fato de que a condição do laço em *C* deve estar entre parênteses.. Quando duas construções diferem apenas no nível léxico, se diz que elas seguem a mesma **sintaxe abstrata**, mas diferem na **sintaxe concreta**. Em outras palavras, elas possuem a mesma estrutura abstrata mas diferem somente em detalhes de baixo nível. Embora conceitualmente a sintaxe concreta possa ser irrelevante, pragmaticamente ela pode afetar o uso da linguagem e a legibilidade dos programas [GHE 97].

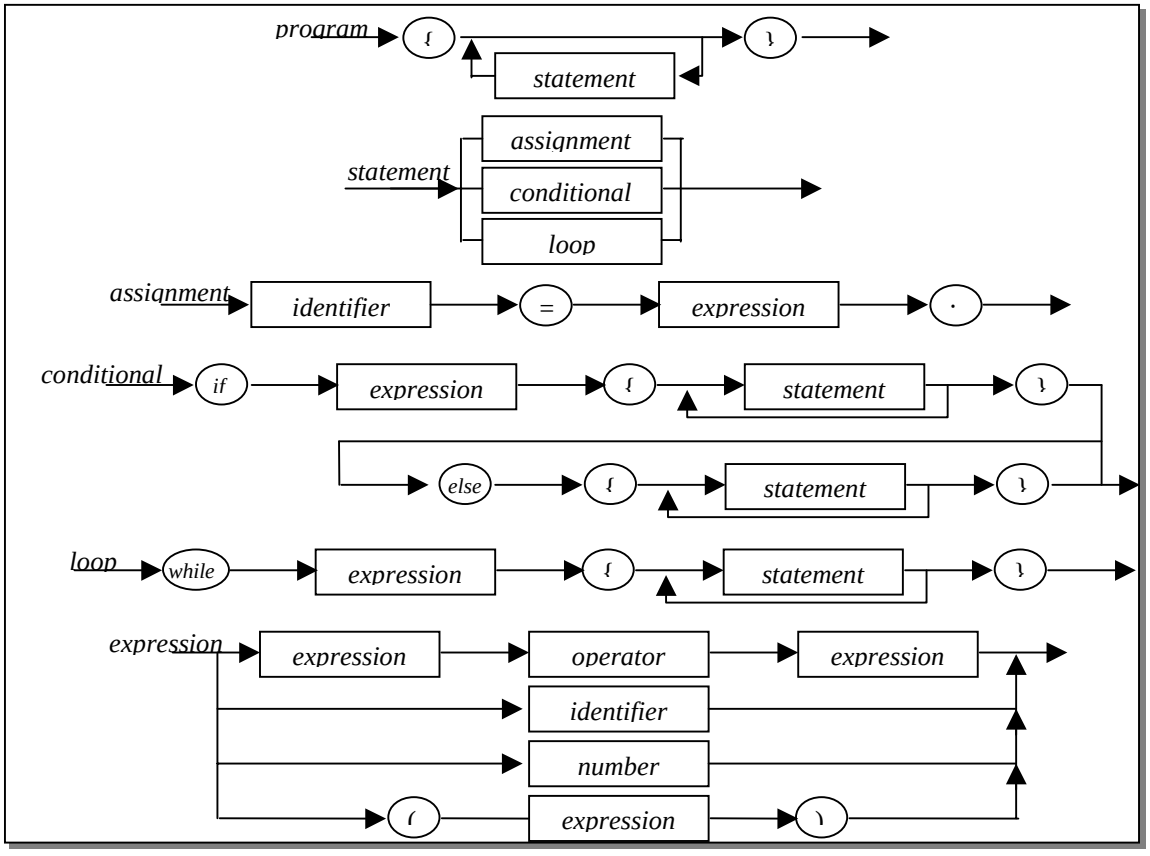

Figura 1.4 – Diagrama de sintaxe para a linguagem descrita na figura 1.2 [GHE 97]