



**PUC  
GOIÁS**

Escola  
**Politécnica  
e de Artes**

**CMP1076 – COMPILADORES**

**ELISEU RODRIGUES GUIMARÃES**

**GOIÂNIA  
2024**

## Uso da BNF e da EBNF para Descrever Gramáticas Livres de Contexto

As Técnicas de Descrição Formal (TDF), como a Backus-Naur Form (BNF) e a Extended Backus-Naur Form (EBNF), são essenciais no desenvolvimento de compiladores e interpretadores, onde uma linguagem precisa ser descrita de forma precisa para que um programa consiga processar comandos corretamente. A obra de Aho et al. (2008) explica em detalhes como a BNF e a EBNF contribuem para a construção de linguagens de programação e compiladores.

### 1. Conceito de BNF

A BNF usa uma notação formal que define a gramática de uma linguagem através de regras de produção. Cada regra de produção define como uma sequência de símbolos pode ser substituída por outros símbolos terminais e não-terminais.

### Exemplo em C de Definição de Expressões Aritméticas em BNF:

Consideremos uma linguagem de expressões aritméticas simples. A BNF pode ser representada de forma prática em C como:



```
bnf_extendida.c  bnf.c  x
FASE03 > C bnf.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int parse_expression();
5  int parse_term();
6  int parse_factor();
7
8  int parse_expression() {
9      int value = parse_term();
10     char operator;
11     while ((operator = getchar()) == '+' || operator == '-') {
12         int term = parse_term();
13         if (operator == '+')
14             value += term;
15         else
16             value -= term;
17     }
18     ungetc(operator, stdin); // retorna o operador não usado
19     return value;
20 }
21
22 int parse_term() {
23     int value = parse_factor();
24     char operator;
25     while ((operator = getchar()) == '*' || operator == '/') {
26         int factor = parse_factor();
27         if (operator == '*')
28             value *= factor;
29         else
30             value /= factor;
31     }
32     ungetc(operator, stdin);
33     return value;
34 }
35
36 int parse_factor() {
37     int value = 0;
38     char ch = getchar();
39     if (ch == '(') {
40         value = parse_expression();
41         getchar(); // consome o ')'
42     } else {
43         ungetc(ch, stdin);
44         scanf("%d", &value);
45     }
46     return value;
47 }
```

Neste exemplo, temos:

**parse\_expression:** Lida com operadores + e -.

**parse\_term:** Lida com operadores \* e /.

**parse\_factor:** Avalia números ou expressões entre parênteses.

Essas funções juntas seguem uma gramática BNF para expressões aritméticas.

## 2. Extensão para EBNF (BNF Extendida)

A EBNF é uma extensão da BNF que facilita a descrição de gramáticas com notações adicionais, como { } para repetição, [ ] para opcionalidade e | para alternativas.

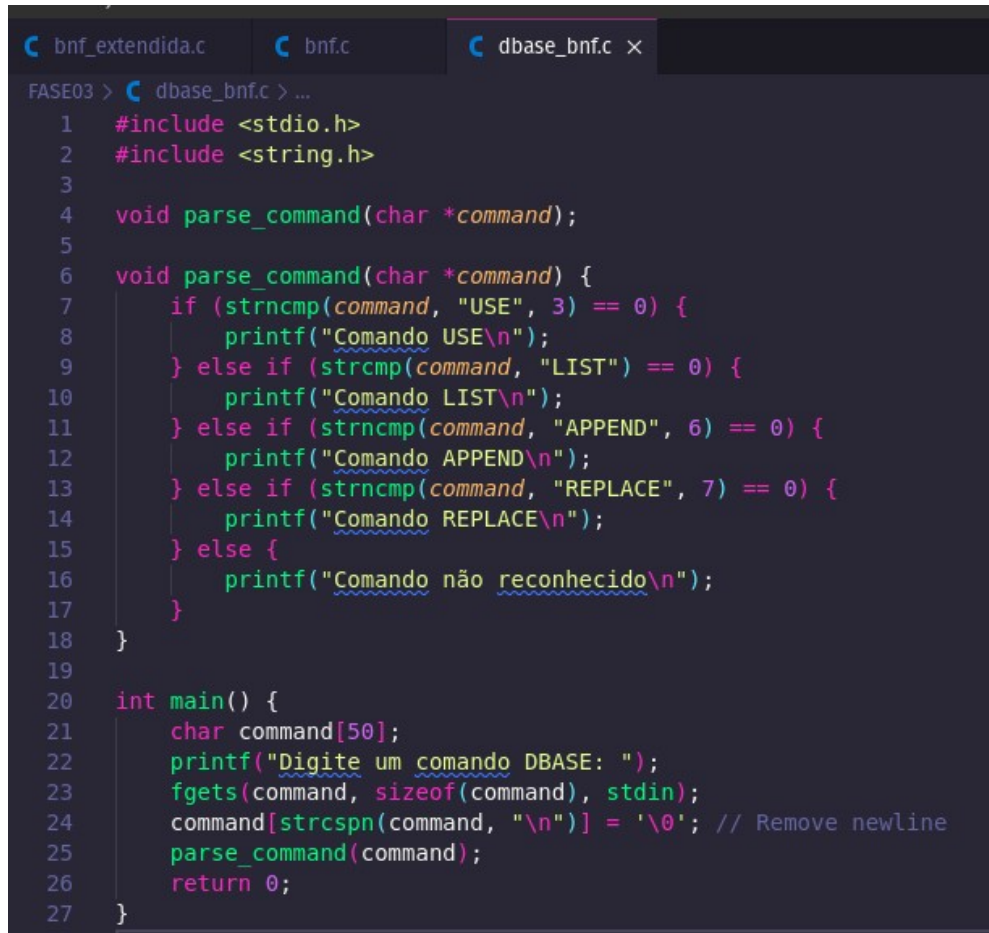
### Exemplo em C de uma Gramática EBNF

```
bnf_extendida.c x  bnf.c
FASE03 > C bnf_extendida.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4
5  int parse_expression();
6  int parse_term();
7  int parse_factor();
8
9  int parse_expression() {
10     int value = parse_term();
11     char operator;
12     while ((operator = getchar()) == '+' || operator == '-') {
13         int term = parse_term();
14         if (operator == '+')
15             value += term;
16         else
17             value -= term;
18     }
19     ungetc(operator, stdin);
20     return value;
21 }
22
23 int parse_term() {
24     int value = parse_factor();
25     char operator;
26     while ((operator = getchar()) == '*' || operator == '/') {
27         int factor = parse_factor();
28         if (operator == '*')
29             value *= factor;
30         else
31             value /= factor;
32     }
33     ungetc(operator, stdin);
34     return value;
35 }
36
37 int parse_factor() {
38     int value = 0;
39     char ch = getchar();
40     if (isdigit(ch)) { // valor é um dígito
41         ungetc(ch, stdin);
42         scanf("%d", &value);
43     } else if (ch == '(') {
44         value = parse_expression();
45         getchar(); // Consome ')'
46     }
47     return value;
48 }
```

Para interpretar expressões com elementos opcionais e repetidos, podemos usar EBNF. Vamos expandir a gramática de expressões para incluir números e parênteses usando EBNF.

### 3. Representação da Linguagem DBASE em BNF

Para descrever a gramática de comandos DBASE (USE, LIST, APPEND, REPLACE) em C usando BNF, definimos as regras de forma que cada comando possua uma função correspondente.



```
bnf_extendida.c  bnf.c  dbase_bnf.c x
FASE03 > C dbase_bnf.c > ...
1  #include <stdio.h>
2  #include <string.h>
3
4  void parse_command(char *command);
5
6  void parse_command(char *command) {
7      if (strncmp(command, "USE", 3) == 0) {
8          printf("Comando USE\n");
9      } else if (strcmp(command, "LIST") == 0) {
10         printf("Comando LIST\n");
11     } else if (strncmp(command, "APPEND", 6) == 0) {
12         printf("Comando APPEND\n");
13     } else if (strncmp(command, "REPLACE", 7) == 0) {
14         printf("Comando REPLACE\n");
15     } else {
16         printf("Comando não reconhecido\n");
17     }
18 }
19
20 int main() {
21     char command[50];
22     printf("Digite um comando DBASE: ");
23     fgets(command, sizeof(command), stdin);
24     command[strcspn(command, "\n")] = '\0'; // Remove newline
25     parse_command(command);
26     return 0;
27 }
```

Neste exemplo:

**USE:** Seleciona o banco de dados.

**LIST:** Exibe registros.

**APPEND:** Adiciona novo registro.

**REPLACE:** Substitui registros específicos.

#### **4. Aplicação e Benefícios**

A aplicação de BNF e EBNF facilita a implementação de interpretadores e compiladores que podem entender comandos e expressões com precisão. Como discutido por Aho et al., ao definir formalmente a gramática de uma linguagem, evitamos ambiguidades e facilitamos o desenvolvimento de sistemas de análise e execução (Aho, 2008).

## **Referências**

AHO, Alfred V. et al. Compiladores: Princípios, Técnicas e Ferramentas. 2ª ed. São Paulo: Pearson Addison-Wesley, 2008.