

**Vinícius José Martins Duran Porto - 01363106**

## **IAs Utilizadas: PDF Chatbot e ChatGPT**

### **Input: Oque é um analisador sintático ascendente?**

Um analisador sintático ascendente, também conhecido como analisador bottom-up, é um tipo de analisador sintático que constrói a árvore de análise sintática começando pelas folhas (símbolos terminais) e avançando em direção à raiz (símbolo inicial). Ele usa um processo de redução para agrupar os símbolos terminais em símbolos não terminais, conforme as regras da gramática.

Em termos mais simples, um analisador sintático ascendente começa com os símbolos terminais da entrada e tenta construir a estrutura da árvore de análise sintática, combinando os símbolos em regras da gramática até alcançar o símbolo inicial. Esse processo pode ser imaginado como "subindo" na árvore sintática.

Existem várias técnicas e algoritmos para implementar analisadores sintáticos ascendentes, incluindo o algoritmo LR (como LR(0), SLR(1), LALR(1), LR(1), etc.) e o algoritmo CYK. Esses algoritmos analisam a entrada de forma incremental, combinando os símbolos em produções da gramática e, se possível, realizando reduções para substituir grupos de símbolos por um símbolo não terminal.

Em resumo, um analisador sintático ascendente é responsável por determinar se uma sequência de símbolos da linguagem de programação é válida de acordo com as regras da gramática e construir a árvore de análise sintática correspondente. Ele trabalha de baixo para cima, a partir dos símbolos terminais, para identificar as estruturas gramaticais maiores até atingir o símbolo inicial da gramática.

### **Input: Então me construa um na linguagem flex.**

Certamente! Aqui está um exemplo de um analisador sintático ascendente (bottom-up) básico implementado com a ferramenta Flex. Neste exemplo, vamos criar um analisador sintático para uma gramática simples de expressões aritméticas com adição e multiplicação.

## Código do Analisador Sintático Ascendente:

```
%{
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
%}
```

```
%option noyywrap
```

```
%union {
```

```
    char* text;
```

```
}
```

```
%token <text> NUMBER
```

```
%left '+' '-'
```

```
%left '*' '/'
```

```
%type <text> expr term factor
```

```
%{
```

```
void yyerror(const char* msg);
```

```
int yylex();
```

```
int yyparse();
```

```
%}
```

```
%%
```

```
input: expr
```

```
    | /* empty */
```

```
    ;
```

```
expr: expr '+' term { $$ = strcat($1, "+"); }
```

```
    | expr '-' term { $$ = strcat($1, "-"); }
```

```
    | term          { $$ = $1; }
```

```
    ;
```

```
term: term '*' factor { $$ = strcat($1, "*"); }
```

```
    | term '/' factor { $$ = strcat($1, "/"); }
```

```
| factor      { $$ = $1; }
```

```
;
```

```
factor: NUMBER    { $$ = $1; }
```

```
| '(' expr ')' { $$ = $2; }
```

```
;
```

```
%%
```

```
int main() {
```

```
    yyparse();
```

```
    return 0;
```

```
}
```

```
void yyerror(const char* msg) {
```

```
    fprintf(stderr, "Erro de análise sintática: %s\n", msg);
```

```
    exit(1);
```

```
}
```

Neste exemplo, utilizamos o Flex para definir as regras léxicas e sintáticas. A gramática é representada usando as regras `input`, `expr`, `term` e `factor`, que definem as regras de produção e a precedência dos operadores.

Quando ocorre um erro de análise sintática, a função `yyerror` é chamada para exibir uma mensagem de erro.

Para compilar o código acima, você precisará gerar o arquivo C a partir do arquivo Flex. Suponha que o arquivo Flex seja chamado `parser.l`, você pode usar o seguinte comando:

```
flex -o parser.c parser.l
gcc -o parser parser.c -lfl
```

Após a compilação, você pode executar o programa gerado `parser` e inserir expressões aritméticas para que o analisador sintático valide a sintaxe.

Vale ressaltar que este é um exemplo simplificado e que a implementação completa de um analisador sintático ascendente para uma linguagem de programação real seria mais complexa e envolveria a construção de uma árvore sintática, tratamento de erros, entre outros aspectos.

## Input: O que faz o código acima?

O código acima implementa um analisador sintático ascendente básico para uma gramática de expressões aritméticas com adição e multiplicação. Vou explicar o que cada parte do código faz:

1. Definições iniciais:
  - As bibliotecas `stdio.h` e `stdlib.h` são incluídas.
  - `%option noyywrap` é usado para desativar a geração automática de uma função `yywrap()`.
2. Blocos de código:

- `%union` é usado para definir uma união chamada `text`, que será usada para armazenar o texto correspondente aos tokens.
  - `%token` é usado para declarar o tipo dos tokens, neste caso `<text>` é usado para indicar que o tipo dos tokens é `char*`.
  - `%left` é usado para definir a precedência dos operadores. Aqui, temos a precedência esquerda para `+` e `-` e também para `*` e `/`.
  - `%type` é usado para definir o tipo dos não terminais. Neste caso, `<text>` é usado para indicar que o tipo dos não terminais é `char*`.
  - `void yyerror(const char* msg)` é uma função para tratar erros de análise sintática.
3. Regras de produção:
    - `input` é o símbolo inicial que representa a entrada do analisador sintático.
    - As regras `expr`, `term` e `factor` definem as produções da gramática. Cada regra é composta por uma ou mais produções separadas por `|` (pipe). À medida que os tokens são reconhecidos pelo analisador léxico (Flex), essas regras são aplicadas para construir a árvore sintática.
  4. Função `main`:
    - `int main()` é a função principal que chama a função `yyparse()` para iniciar o processo de análise sintática.
  5. Função `yyerror`:
    - `void yyerror(const char* msg)` é chamada quando ocorre um erro de análise sintática. Neste exemplo, a função simplesmente imprime uma mensagem de erro no console.

No geral, o código implementa as regras sintáticas para a gramática de expressões aritméticas e processa a entrada para validar sua sintaxe. As ações semânticas, como a construção de uma árvore sintática ou a avaliação das expressões, não estão implementadas nesse exemplo, mas podem ser adicionadas conforme necessário.

## Input: Adicione as operações de multiplicação e de divisão

Aqui está o código atualizado com as operações de multiplicação (`*`) e divisão (`/`) adicionadas:

## Código do Analisador Sintático Ascendente:

```
%{
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
%}
```

```
%option noyywrap
```

```
%union {
```

```
    char* text;
```

```
}
```

```
%token <text> NUMBER
```

```
%left '+' '-'
```

```
%left '*' '/'
```

```
%type <text> expr term factor
```

```
%{
```

```
void yyerror(const char* msg);
```

```
int yylex();
```

```
int yyparse();
```

```
%}
```

```
%%
```

```
input: expr
```

```
    | /* empty */
```

```
    ;
```

```
expr: expr '+' term { $$ = strcat($1, "+"); }
```

```
    | expr '-' term { $$ = strcat($1, "-"); }
```

```
    | term          { $$ = $1; }
```

```
    ;
```

```
term: term '*' factor { $$ = strcat($1, "*"); }
```

```
    | term '/' factor { $$ = strcat($1, "/"); }
```

```
    | factor         { $$ = $1; }
```



;

factor: NUMBER { \$\$ = \$1; }

| '(' expr ')' { \$\$ = \$2; }

;

%%

int main() {

    yyparse();

    return 0;

}

void yyerror(const char\* msg) {

    fprintf(stderr, "Erro de análise sintática: %s\n", msg);

    exit(1);

}

Agora, as regras de produção `term` foram atualizadas para incluir as operações de multiplicação (`*`) e divisão (`/`). As regras `term: term '*' factor` e `term: term '/' factor` foram adicionadas para lidar com essas operações.

Após adicionar as operações de multiplicação e divisão, você pode recompilar e executar o código para testar expressões que envolvam essas operações. Por exemplo, a expressão `"2 * 3 / 4"` seria interpretada corretamente, construindo a árvore sintática correspondente às operações.