📝 Léxico-Sintático LL(1) (Básico)

🧩 Análise Léxica

🎯 Objetivo

Converter uma entrada bruta (código-fonte ou sequência de caracteres) em uma **lista de tokens** reconhecidos por um **autômato** ou lógica sequencial.

🔶 Etapas

✅ 1. Identifique todos os terminais/tokens da gramática

- `id`
- `+`
- `*`
- `++`
- `(`
- `)`
- `$`

✅ 2. Definir os tokens

Use um `enum` com identificadores únicos para cada tipo de token.

🟦 Versão em **C**:

```c
enum TokenType {
    TOK_ID = 0, TOK_PLUS, TOK_STAR, TOK_PLUSPLUS,
    TOK_LPAREN, TOK_RPAREN, TOK_DOLLAR, TOK_UNKNOWN
};
```

🟪 Versão em **C++**:

```cpp
enum class TokenType {
    TOK_ID = 0, TOK_PLUS, TOK_STAR, TOK_PLUSPLUS,
    TOK_LPAREN, TOK_RPAREN, TOK_DOLLAR, TOK_UNKNOWN
};
```

✅ 3. Implementar o lexer (scanner)

- Percorrer a string caractere por caractere
- Ignorar espaços, caso necessário

- Comparar símbolos com **strings fixas** (modo simplificado, porém não é ideal para gramáticas complexas)
- **A ordem das verificações define a prioridade dos tokens.**
    - **Do topo para baixo: maior prioridade → menor prioridade**
    - Isso é essencial para evitar ambiguidades!

🔁 **Exemplo crítico**:
Para a entrada `++`:

- Se o `+` for testado **antes** do `++`, o lexer reconhecerá `+` e depois `+` (`TOK_PLUS TOK_PLUS`)
- Se o `++` for testado **antes**, o lexer reconhecerá `++` corretamente (`TOK_PLUSPLUS`)

🔽 Portanto, **sempre teste lexemas mais longos primeiro** e fique atento as prioridades.

🟦 Versão em **C**:

```c
int next_token(const char *line, int *pos) {
    // Ignora os espaços
    while (isspace(line[*pos]))
    {
        (*pos)++;
    }

    if (line[*pos] == '\0')
    {
        return TOK_UNKNOWN;
    }

    // Ordem de prioridade: maiores lexemas primeiro!
    if (line[*pos] == '+' && line[*pos + 1] == '+') {
        *pos += 2; return TOK_PLUSPLUS;
    }
    if (line[*pos] == '+' ) {
        *pos += 1; return TOK_PLUS;
    }
    if (line[*pos] == '*') {
        *pos += 1; return TOK_STAR;
    }
    if (line[*pos] == '(') {
        *pos += 1; return TOK_LPAREN;
    }
    if (line[*pos] == ')') {
        *pos += 1; return TOK_RPAREN;
    }
    if (line[*pos] == '$') {
        *pos += 1; return TOK_DOLLAR;
    }
    if (line[*pos] == 'i' && line[*pos + 1] == 'd') {
        *pos += 2; return TOK_ID;
    }
```

```
    (*pos)++;
    return TOK_UNKNOWN;
}
```

🟪 Versão em **C++**:

```
TokenType next_token(const std::string& line, size_t& pos) {
    // Ignora os espaços
    while (isspace(line[pos]))
    {
        ++pos;
    }

    if (pos >= line.size())
    {
        return TokenType::TOK_UNKNOWN;
    }

    // Ordem de prioridade: maiores lexemas primeiro!
    if (line[pos] == '+' && line[pos + 1] == '+') {
        pos += 2; return TokenType::TOK_PLUSPLUS;
    }
    if (line[pos] == '+') {
        ++pos; return TokenType::TOK_PLUS;
    }
    if (line[pos] == '*') {
        ++pos; return TokenType::TOK_STAR;
    }
    if (line[pos] == '(') {
        ++pos; return TokenType::TOK_LPAREN;
    }
    if (line[pos] == ')') {
        ++pos; return TokenType::TOK_RPAREN;
    }
    if (line[pos] == '$') {
        ++pos; return TokenType::TOK_DOLLAR;
    }
    if (line[pos] == 'i' && line[pos + 1] == 'd') {
        pos += 2; return TokenType::TOK_ID;
    }

    ++pos;
    return TokenType::TOK_UNKNOWN;
}
```

✅ **4. Imprimir tokens (debugging)**

🟦 Versão em **C**:

```
printf("%d ", token);  // Ex: 0 1 2 6
```

🟪 Versão em **C++**:

```
std::cout << static_cast<int>(token) << " ";  // Ex: 0 1 2 6
```

📐 Análise Sintática LL(1) (Versão Descendente Recursiva)

🎯 Objetivo

Interpretar uma **sequência de tokens** (gerada pelo lexer) de acordo com uma **gramática livre de contexto**, validando a **estrutura** da entrada e possibilitando a **tradução/execução**.

🔷 LL(1): Características

- **L**eft-to-right: analisa da esquerda para a direita
- **L**eftmost derivation: derivações mais à esquerda
- **1** token de lookahead (pré-visualização)

⚠️ Transformações importantes para LL(1)

Algumas gramáticas precisam ser **reescritas** antes de serem usadas em parsers LL(1), especialmente quando contêm:

1. **Recursão à esquerda**
2. **Produções com prefixos comuns (fatoração)**

✅ Eliminar **Recursão à Esquerda**

📌 Problema:

```
A → A α
A → β
```

Isso é **recursão à esquerda direta**, e causa **loop infinito** no parser descendente.

✅ Solução:

1. Transforme a recursão à esquerda em recursão à direita, criando uma nova produção: `A → A α` em `A' → A' α`
2. Acrescente a novo não terminal no final de todas as produções de `A`: `A → β` em `A' → β A'`
3. Adicione uma produção vazia para o novo não terminal criado `A'`: `A' → ε`

Resultado:

```
A → β A'
A' → α A'
A' → ε
```

🧪 **Exemplo:**

Antes:

```
Expr → Expr + Term
     | Term
```

Depois:

```
Expr → Term Expr'
Expr' → + Term Expr'
      | ε
```

✅ Fatoração (eliminar prefixos comuns)

📌 **Problema:**

```
S → if E then S else S
S → if E then S
```

Os dois ramos começam com o mesmo prefixo: **`if E then S`**.

✅ **Solução:**

Extraia o prefixo comum:

```
S → if E then S S'
S' → else S | ε
```

🔶 Etapas

✅ **1. Defina a gramática**

```
S → E $
E → T R
R → + T R | ε
T → id
```

Essa gramática reconhece expressões como:

- `id $`
- `id + id $`
- `id + id + id $`

✅ 2. Calcule `FIRST`, `FOLLOW`, `NULLABLE`

🟦 Nullable:

Símbolo	Nullable
`S`	❌
`E`	❌
`R`	✅ (tem uma produção ε)
`T`	❌

🟨 FIRST Sets:

Terminais:

- `FIRST(id) = { id }`
- `FIRST(+) = { + }`
- `FIRST($) = { $ }`

Não terminais:

- `FIRST(S) = FIRST(E) = { id }`
- `FIRST(E) = FIRST(T) = { id }`
- `FIRST(R) = { + }`
- `FIRST(T) = { id }`

🟧 FOLLOW Sets:

- `FOLLOW(S) = { }`
- `FOLLOW(E) = FIRST($) = { $ }`
- `FOLLOW(R) = FOLLOW(R) U FOLLOW(E) = { $ }`
- `FOLLOW(T) = FIRST(R) U FOLLOW(E) U FOLLOW(R) = { +, $ }`

🟪 Tabela `NULLABLE`, `FIRST` e `FOLLOW`:

Não terminal	Nullable	First	Follow
`S`	Não	id	
`E`	Não	id	$
`R`	Sim	+	$

Não terminal	Nullable	First	Follow
`T`	Não	id	+, $

✅ 3. Construa a tabela LL(1)

Não-terminal	`id`	`+`	`$`
`S`	`S → E $`		
`E`	`E → T R`		
`R`		`R → + T R`	`R → ε`
`T`	`T → id`		

✅ 4. Implementação do Parser (descendente recursivo)

🧭 Como implementar o parser com a tabela LL(1)

> 💡 **Ideia central**:
>
> A tabela LL(1) funciona como um **roteiro para o parser**. Cada célula da tabela (`Não-terminal × Terminal`) informa **qual produção aplicar** com base no **token atual**.
>
> - O parser é implementado com funções recursivas que representam cada **não-terminal**.
> - Cada função possui uma estrutura `switch-case` ou `if-else` com base no **token atual**, para executar a produção conforme indicada na tabela.
> - Se a produção for do tipo `A → ε`, o bloco correspondente **não faz nada**.
>
> ⚠️ **Erros de sintaxe** são detectados quando o token atual **não corresponde** a nenhuma entrada válida na tabela para aquele não-terminal.

Exemplo visual da ideia:

Não-terminal	`+`	`id`	`$`
`R`	`R → + T R`		`R → ε`

```
void R() {
    if (token == TOK_PLUS) {
        eat(TOK_PLUS);
        T();
        R();
    }
    else if (token == TOK_DOLLAR) {
        // Produção R → ε
        // Nada é feito
    }
    else {
        error("Token inesperado em R()");
```

```
    }
}
```

🟦 **Em C:**

```c
int token; // Token atual

void advance() {
    // Avança para o próximo token da entrada (via scanner/lexer)
    token = next_token();  // Função fornecida pelo analisador léxico
}

void eat(int expected) {
    // Verifica se o token atual é o esperado
    // Se for, consome ele e avança
    // Senão, lança erro de sintaxe
    if (token == expected)
    {
        advance();
    }
    else
    {
        error("Token inesperado");
    }
}

void S() {
    // Produção S → E $
    E();
    eat(TOK_DOLLAR);
}

void E() {
    // Produção E → T R
    T();
    R();
}

void R() {
    // Possui mais de uma produção então usamos a tabela LL(1)
    if (token == TOK_PLUS) {
        // Se for + ou TOK_PLUS (olhar coluna)
        // Produção R → + T R
        eat(TOK_PLUS);
        T();
        R();
    }
    if (token == TOK_DOLLAR) {
        // Se for $ ou TOK_DOLLAR (olhar coluna)
        // Produção R → ε
        // Não faz nada
```

```
    }
}

void T() {
    // Produção T → id
    eat(TOK_ID);
}
```

🟪 **Em C++:**

```
TokenType token; // Token atual

void advance() {
    // Avança para o próximo token analisado pelo lexer
    token = next_token();
}

void eat(TokenType expected) {
    // Verifica se o token atual é o esperado
    // Se sim, consome o token e avança
    // Caso contrário, gera erro de sintaxe
    if (token == expected)
    {
        advance();
    }
    else
    {
        throw std::runtime_error("Unexpected token");
    }
}

void S() {
    // Produção S → E $
    E();
    eat(TokenType::TOK_DOLLAR);
}

void E() {
    // Produção E → T R
    T();
    R();
}

void R() {
    // Possui mais de uma produção então usamos a tabela LL(1)
    if (token == TokenType::TOK_PLUS) {
        // Se for + ou TOK_PLUS (olhar coluna)
        // Produção R → + T R
        eat(TokenType::TOK_PLUS);
        T();
        R();
```

```
        }
        if (token == TokenType::TOK_DOLLAR) {
            // Se for $ ou TOK_DOLLAR (olhar coluna)
            // Produção R → ε
            // Nada é feito
        }
    }

    void T() {
        // Produção T → id
        eat(TokenType::TOK_ID);
    }
```

✅ 5. Funções auxiliares

Função	Papel
`advance()`	Consulta o próximo token da entrada (pelo analisador léxico)
`eat(x)`	**Verifica se o token atual é o esperado (`x`)**; se for, consome-o; senão, gera erro de sintaxe
`token`	Armazena o token atual sendo analisado pelo parser

✅ 6. Exemplo de uso

Entrada: `id + id + id $`
Tokens: `TOK_ID TOK_PLUS TOK_ID TOK_PLUS TOK_ID TOK_DOLLAR`

Chamada de `S()`:

```
S()
└── E()
    ├── T() → eat(id)
    └── R()
        ├── eat(+)
        ├── T() → eat(id)
        └── R()
            ├── eat(+)
            ├── T() → eat(id)
            └── R() → ε
```