**UNIVERSIDADE CATÓLICA DE PERNAMBUCO**

**CIÊNCIAS DA COMPUTAÇÃO**

**CONSTRUÇÃO DE COMPILADORES**

**MARCOS VINICIUS MEDEIROS FERREIRA**

**NICOLAS FERREIRA BICUDO DUARTE**

**NOEMI SOARES GONÇALVES DA SILVA**

**NUNNO WAKIYAMA DINIZ CARVALHO**

# RELATÓRIO:

# DESENVOLVIMENTO DO ANALISADOR LÉXICO

**RECIFE**

**2024**

**SUMÁRIO**

## RESUMO

Neste projeto, nosso objetivo foi desenvolver um analisador léxico em C, focando na análise de expressões regulares e na estruturação de tokens. Iniciamos com uma pesquisa aprofundada, onde analisamos documentos como uma tabela de transição fornecidos pelo avaliador. Esses materiais foram essenciais para compreender as regras gramaticais necessárias à implementação do analisador.

A implementação do analisador envolveu a manipulação de arquivos em C, permitindo uma leitura dinâmica de um arquivo fonte `.c`. Para armazenar os tokens extraídos, decidimos desenvolver duas versões do analisador: uma utilizando vetores e outra utilizando listas encadeadas. Essa abordagem nos possibilitou comparar a eficiência de cada implementação em termos de uso de memória e flexibilidade. Além disso, criamos uma tabela de símbolos para registrar informações detalhadas sobre identificadores, facilitando a verificação de erros.

Durante o desenvolvimento, enfrentamos diversas dificuldades, que foram superadas através de colaboração e pesquisa. Este documento também inclui um passo a passo de ambos os códigos, detalhando as abordagens e as lições aprendidas ao longo do processo.

## 1. ANÁLISE DE PROJETO

Antes de iniciar o projeto, realizamos uma reunião para discutir as opções de linguagem de programação que poderíamos utilizar. Após uma análise dos pontos fortes e das limitações de cada linguagem que poderíamos usar, decidimos pela linguagem C. Essa escolha se deve à familiaridade da maioria dos integrantes do grupo com suas funcionalidades, especialmente em relação à manipulação de arquivos, uso de structs e a criação de Tipos Abstratos de Dados (TADs). Além disso, a linguagem C é amplamente utilizada no desenvolvimento de compiladores, o que se alinha perfeitamente com nosso objetivo.

## 2. PESQUISA

No primeiro estágio da nossa pesquisa, focamos na análise das Expressões Regulares presentes em um arquivo Excel e em um documento PDF intitulado "Tabela de Transição - Aux". Esses documentos foram essenciais para nos ajudar a compreender a estrutura dos tokens e as regras gramaticais que precisaríamos implementar no nosso analisador léxico. A partir dessa análise, conseguimos delinear um plano claro para o desenvolvimento do código em C.

Aprofundamo-nos em fóruns online e assistimos a vídeos em inglês, que abordavam técnicas e melhores práticas para a construção de analisadores léxicos. Essa pesquisa foi crucial para que pudéssemos identificar não apenas as estratégias mais eficazes, mas também as armadilhas comuns a evitar. Em particular, estudamos como usar expressões regulares para tokenização e as estruturas de dados que seriam mais apropriadas para armazenar e classificar os tokens gerados.

## 3. CONSIDERAÇÕES SOBRE IMPLEMENTAÇÃO

Para a construção do analisador léxico, concluímos que a leitura de um arquivo fonte .c seria mais bem gerenciada por meio de manipulação de arquivos em C. Isso nos permitiu processar entradas de forma dinâmica e flexível. Optamos por implementar uma abordagem baseada em listas para armazenar os tokens extraídos durante a análise. As listas são ideais para essa tarefa, pois oferecem fácil inserção e remoção de elementos, além de permitir a expansão dinâmica conforme necessário. Além disso, planejamos criar uma tabela de símbolos que manterá informações detalhadas sobre os identificadores

encontrados no código-fonte, como seus tipos e escopos. Isso não apenas facilitou a verificação de erros, como também forneceu informações valiosas para as fases subsequentes do processo de compilação.

Decidimos, ainda, produzir dois códigos diferentes para o analisador: um utilizando vetores e outro utilizando listas. Essa abordagem permitiu comparar a eficiência de cada uma das implementações, levando em consideração o uso de memória e a flexibilidade para diferentes cenários, além de garantir maior versatilidade na escolha da estrutura de dados mais adequada para o projeto.

## 4. DIFICULDADES PASSADAS DURANTE O DESENVOLVIMENTO

Durante o desenvolvimento do nosso analisador léxico, enfrentamos diversas dificuldades. Um dos primeiros desafios foi a identificação correta dos diferentes tipos de tokens. Precisamos fazer com que o analisador diferenciásse entre números inteiros, decimais, identificadores, palavras reservadas, operadores e outros símbolos da linguagem. Cada token possui expressões regulares próprias, o que nos auxiliou, mas, em casos mais complexos, acabamos enfrentando erros na detecção, como a confusão entre números inteiros e decimais, ou a interpretação incorreta de identificadores que não foram bem delimitados.

Outra dificuldade significativa foi lidar com os operadores e símbolos especiais. Operadores compostos, como `<=`, `>=`, `==` e `!=`, exigiram atenção extra para garantir que não fossem tratados como operadores simples separados. Além disso, garantir que símbolos especiais, como delimitadores e sinais de pontuação, fossem corretamente processados de modo a não interferir no reconhecimento dos outros tokens foi uma questão desafiadora, especialmente para caracteres especiais como `'.'`, `'#'`, `'@'`, e `'$'`. Tais símbolos precisavam ser corretamente classificados e adicionados à tabela de tokens.

Além disso, o tratamento de operadores literais que também apresentou complicações. Precisamos garantir que esses operadores compostos fossem detectados corretamente, evitando interpretações incorretas caso aparecessem de maneira separada ou malformada no código. O reconhecimento dos operadores como um todo foi importante para manter a coerência na análise.

O tratamento de comentários e espaços em branco foi outra etapa desafiadora.

Comentários, tanto os de linha única (como `//`), precissavam ser corretamente identificados pelo analisador. No entanto, erros ao ignorar esses elementos poderiam fazer com que partes do código fossem interpretadas erroneamente como tokens inválidos. Da mesma forma, lidar com espaços em branco, tabulações e quebras de linha foi um ponto delicado, já que, embora muitas vezes devam ser ignorados, eles podem influenciar o significado do código.

Também enfrentamos complicações com o tratamento de literais e constantes de texto. Strings delimitadas por aspas, por exemplo, precisavam ser reconhecidas como um único token, independentemente do que houvesse dentro delas. Isso exigiu uma leitura própria para que o analisador ignorasse os tokens internos e tratasse a string como uma unidade. Problemas surgiram principalmente ao lidar com caracteres especiais ou casos de escape, como aspas duplas dentro de uma string.

Outro desafio foi a separação correta entre números inteiros e decimais. Implementamos uma função `is_number()` para diferenciar entre os dois, mas houve dificuldades iniciais em garantir que números com ponto decimal fossem classificados como decimais e não como inteiros. Precisávamos garantir que esses números fossem corretamente classificados e que seus literais fossem armazenados de maneira adequada nas listas de símbolos e tokens.

O gerenciamento de memória e o uso de listas encadeadas para armazenar os tokens também foi uma etapa complexa. Precisávamos garantir que as estruturas de dados utilizadas fossem eficientes e que o uso de memória fosse bem controlado para que nada se perdesse. Erros na alocação ou na manipulação de ponteiros poderiam comprometer a execução do analisador e levar ao armazenamento incorreto dos tokens.

Tivemos que lidar com a ambiguidade na interpretação de alguns tokens, como aqueles que poderiam ser tanto palavras reservadas quanto identificadores. Para resolver essa questão, foi necessário criar uma lógica precisa baseada em uma lista de vetor em uma função de processamento que permitisse distinguir corretamente entre esses dois casos, garantindo que o programa seguisse o fluxo correto de análise e evitasse confusões.

Uma dificuldade adicional envolveu a classificação de caracteres que não se enquadravam em nenhuma das categorias tradicionais. Implementamos uma verificação

adicional para lidar com esses casos especiais, garantindo que fossem devidamente categorizados como tokens especiais ou operadores.

Por fim, enfrentamos a etapa mais desafiadora: garantir que os identificadores fossem inseridos na tabela de símbolos na mesma ordem em que apareciam na lista de tokens, preservando sua numeração original. Qualquer erro na ordem de inserção poderia comprometer a correspondência entre o código-fonte e a tabela de símbolos.

## 5. PASSO A PASSO: CONSTRUÇÃO DO ANALISADOR COM VETORES

Para desenvolver esta etapa do processo de compilação, utilizei como referência uma abordagem semelhante à dos separadores de moedas, que são organizados de acordo com o tamanho. Nos separadores, as moedas deslizam por uma rampa repleta de furos que aumentam gradualmente em diâmetro, permitindo que cada moeda permaneça em seu caminho até que reste apenas uma opção. Nessa analogia, os lexemas e caracteres correspondem às moedas, enquanto os furos na rampa representam as condicionais do código.

Minha ideia consiste em que os caracteres especiais ou palavras sejam lidos de um determinado arquivo usado como entrada no código. Em seguida, passarão por uma sequência de condicionais capazes de distinguir e classificar adequadamente cada um deles. Com base nessa ideia principal, nas pesquisas que realizei e nas contribuições dos meus colegas de grupo, procurei desenvolver um código capaz de separar o conteúdo do arquivo de entrada em duas listas distintas. A primeira lista, denominada tabela de tokens, seria responsável por armazenar os tokens e identificadores, enquanto a segunda, denominada lista de símbolos, ficaria responsável pela coleta dos símbolos.

- **ETAPA 1 (função de buscar caracteres numa lista)**

Implementei uma função que retorna um inteiro, seu nome foi “buscaLista” (posteriormente se tornou “buscaCaractere”). Essa função recebe como parâmetro um caractere, a lista a ser percorrida e o tamanho da lista. Após sua implementação, realizei um teste definindo um caractere específico e alguns caracteres especiais na lista de tokens. Como planejado, deu certo.

```c
#include <stdio.h>

// Função para verificar se o caractere está presente na lista de caracteres especiais
int buscaLista(char caractere, char lista[], int tamanho_lista) {
    for (int i = 0; i < tamanho_lista; i++) {
        if (caractere == lista[i]) {
            return 1;  // O caractere está na lista
        }
    }
    return 0;  // O caractere não está na lista
}

int main() {
    char caractere = ')';   // Exemplo de caractere
    char lista_de_tokens[] = {'(', ')', ';', '.', ',', '=', '*', '<', '>', '#', '{', '}'}; // Exemplo de lista
    int tamanho_lista_tokens = sizeof(lista_de_tokens) / sizeof(lista_de_tokens[0]); // Tamanho da lista

    if (caractere != ' ' && buscaLista(caractere, lista_de_tokens, tamanho_lista_tokens) == 1) {
        printf("Caractere presente na lista");
    }
    else {
        printf("Caractere nao esta na lista");
    }

    return 0;
}
```

**Figura 1: Código da etapa 1.**

```
PS D:\unicap\Assis (Compiladores)\Analisador lexico\output> & .\'teste.exe'
Caractere presente na lista
PS D:\unicap\Assis (Compiladores)\Analisador lexico\output>
```

**Figura 2: Saída da etapa 1.**

- **ETAPA 2 (ler os caracteres de um arquivo)**

Após concluir essa etapa, meu próximo objetivo foi permitir que o programa recebesse como entrada um arquivo, verificando cada caractere em relação à lista de tokens previamente estabelecida. O arquivo selecionado foi um código padrão em C que imprime “Hello World!”. Por mais que a implementação tenha funcionado, notei que as quebras de linha “\n” não estavam sendo reconhecidas como parte da lista, enquanto os espaços em

```c
#include <stdio.h>

// Função para verificar se o caractere está presente na lista de caracteres especiais
int buscaLista(char caractere, char lista[], int tamanho_lista) {
    for (int i = 0; i < tamanho_lista; i++) {
        if (caractere == lista[i]) {
            return 1;  // O caractere está na lista
        }
    }
    return 0;  // O caractere não está na lista
}

int main() {
    FILE* arquivo;
    char caractere = ')';   // Exemplo de caractere
    char lista_tokens[] = {'(', ')', ';', '.', ',', '=', '*', '<', '>', '#', '{', '}'}; // Exemplo de lista
    int tamanho_lista_tokens = sizeof(lista_tokens) / sizeof(lista_tokens[0]); // Tamanho da lista

    // Abrindo o arquivo em modo de leitura
    arquivo = fopen("index.c", "r");

    // Verificando se o arquivo foi aberto corretamente
    if (arquivo == NULL) {
    // Lendo os caracteres do arquivo um por um
    while ((caractere = fgetc(arquivo)) != EOF) {
        // Verifica se o caractere não é um espaço e não está na lista
        if (caractere != ' ' && buscaLista(caractere, lista_tokens, tamanho_lista_tokens) == 0) {
            printf("Caractere '%c' nao esta na lista.\n", caractere);
        } else {
            printf("Caractere '%c' esta na lista.\n", caractere);
        }
    }

    // Fechando o arquivo
    fclose(arquivo);

    return 0;
}
```

branco ‘ ‘ estavam sendo considerados como elementos da minha lista de tokens.

**Figura 3 e 4: Código da etapa 2.**

```
PS D:\unicap\Assis (Compiladores)\Analisador lexico\output> & .\'teste.exe'
Caractere '#' esta na lista.
Caractere 'i' nao esta na lista.
Caractere 'n' nao esta na lista.
Caractere 'c' nao esta na lista.
Caractere 'l' nao esta na lista.
Caractere 'u' nao esta na lista.
Caractere 'd' nao esta na lista.
Caractere 'e' nao esta na lista.
Caractere ' ' esta na lista.
Caractere '<' esta na lista.
Caractere 's' nao esta na lista.
Caractere 't' nao esta na lista.
Caractere 'd' nao esta na lista.
Caractere 'i' nao esta na lista.
Caractere 'o' nao esta na lista.
Caractere '.' esta na lista.
Caractere 'h' nao esta na lista.
Caractere '>' esta na lista.
Caractere '
' nao esta na lista.
Caractere '
' nao esta na lista.
```

**Figura 5: Saída da etapa 2.**

- **ETAPA 3 (criar tabela de simbolos e a lista de tokens)**

Para a próxima etapa, decidi renomear a “lista_tokens” para “char_especiais”, uma vez que ela não representa a lista de tokens propriamente dita, mas sim uma lista auxiliar para identificar tokens. Além disso, defini o valor 100 como “TAMANHO_MAXIMO” e adicionei duas novas listas, também com tamanho máximo de 100. A primeira, denominada “symbol_table”, funcionará como a futura tabela de símbolos, enquanto a segunda, intitulada “lista_tokens”, será responsável por armazenar os tokens.

- **ETAPA 4 (alocar os vetores dinamicamente)**

Na etapa 4, percebi que seria uma péssima ideia definir um tamanho máximo para os vetores, uma vez que o tamanho varia conforme o arquivo de entrada. Portanto, essa etapa foi dedicada à correção desse erro. Para isso, utilizei variáveis ponteiro para alocar valores dinamicamente. Sempre que um novo valor precisava ser adicionado, empreguei a função “realloc” da biblioteca “stdlib.h”, Sempre verificando se os vetores estavam vazios (== NULL) para retornar um erro de alocação, quando necessário.

- **ETAPA 5 (ignorar comentários comuns e blocos de comentários)**

Nesta etapa, busquei corrigir o erro identificado na etapa 2, no qual o código considerava as quebras de linha “\n” ao formar a lista de tokens e a tabela de símbolos. Para resolver isso, adicionei uma condição que garante que apenas (caracteres != “\n”) sejam incluídos nas duas listas.

Além disso, o próximo passo foi fazer com que o código ignorasse os comentários durante a leitura do arquivo. Para implementar essa funcionalidade, foi necessário adicionar uma condição que, ao encontrar uma barra (‘/’), verificasse se o próximo caractere era outra barra ‘/’ (comentário comum) ou um asterisco ‘*’ (comentário em bloco), uma vez que os comentários são iniciados por “//” ou “/*”. Se um comentário comum for identificado, o programa ignorará todos os caracteres subsequentes até encontrar uma quebra de linha (‘\n’), que indicaria o fim do comentário. Se um comentário de bloco for identificado, o programa ignorará todos os caracteres até que um “*/” seja encontrado, sinalizando o fim do comentário em bloco.

- **ETAPA 6 (definir um buffer para armazenar símbolos)**

Até o momento, a tabela de símbolos e a lista de tokens estavam recebendo apenas caracteres que não eram espaços em branco, quebras de linha (‘\n’) ou caracteres pertencentes a comentários. Para melhorar esse processo, criei uma variável denominada palavra_buffer, com capacidade para armazenar 256 caracteres (palavra_buffer[256]). Essa variável tem a finalidade de acumular os caracteres lidos até que um char_escpecial, uma quebra de linha, um espaço em branco ou o início de um comentário seja detectado.

Assim que um desses caracteres for identificado, o conteúdo do buffer é transferido para a tabela de símbolos. Caso o caractere responsável por interromper o preenchimento do buffer seja um char_especial, ele é adicionado à lista de tokens. No entanto, se for uma quebra de linha, um espaço em branco ou o início de um comentário, esses caracteres são simplesmente desconsiderados.

- **ETAPA 7 (função buscar palavras reservadas através do buffer)**

Na etapa 7, adicionei um novo array denominado palavras_reservadas, que contém todas as palavras reservadas listadas no arquivo Excel disponibilizado: {"int", "float", "char",

"boolean", "void", "if", "else", "for", "while", "scanf", "printf", "main", "return"}.

Além disso, implementei uma função no código chamada “buscaPalavra”, seu objetivo é verificar se a palavra lida do arquivo é uma das palavras_reservadas. Essa função recebe como parâmetro a string contida no buffer, a lista a ser percorrida e o tamanho dessa lista. Para facilitar a comparação de palavras (strings), adicionei a biblioteca string.h, que permite o uso da função strcpy(). (Então invés do buffer ir direto para lista de símbolos, a string dentro do buffer será comparada com o array palavras_reservadas, se for compatível vai para lista de tokens, se não vai para tabela de simbolos).

- **ETAPA 8 (printar lista de símbolos e tabela de tokens)**

Etapa 8 foi apenas para gerar os prints no fim do código tanto da tabela de tokens quanto da lista de símbolos. Ambos printavam com o auxílio de um for que percorria todos os elementos do array e ia printando. Depois dos prints, também coloquei mais 2 fors para desalocar corretamente todos os elementos de cada array, garantindo a liberação adequada da memória utilizada.

- **ETAPA 9 (dividir os símbolos em strings e números is_number() e isdigit())**

Nesta etapa, o objetivo foi verificar os símbolos lidos do buffer para determinar se eles são números ou strings. Dependendo de como cada símbolo for classificado, ele será inserido na lista de tokens como identificadores: (NUM, contador) para números e (ID, contador) para strings.

Para pôr isto em prática, importei a biblioteca ctype.h, que permite identificar o tipo primitivo dos caracteres. Criei uma função chamada is_number(), que recebe como parâmetro a palavra do buffer e verifica (através de um for) se a string é inteira composta por numeros. Para essa verificação, utilizei a função isdigit() (função da biblioteca ctype.h que retorna 1/true se o caractere for um digito), caso a string seja composta apenas por números, a função is_number() retorna 1, se não, a função retorna 0.

No código, se a função is_number() retornar 1, o número será adicionado à tabela de símbolos, e o seu token será registrado na lista de tokens como (NUM, contador). Por outro lado, se a função retornar 0, a string será inserida na tabela de símbolos, e seu token será registrado na lista de tokens como (ID, contador).

- **ETAPA 10 (função find_symbol() evitar valores iguais na lista de simbolos)**

Etapa 10 foi para que o código verificasse se um novo token do tipo NUM ou ID já possui um equivalente na lista de tokens, ou seja, caso o código leia do arquivo um símbolo que já foi lido anteriormente, não será adicionado novamente na tabela de símbolos.

Para tal, uma função find_symbol() foi adicionada, essa função recebe como parâmetro a própria lista de símbolos, a quantidade de símbolos adicionados e o novo símbolo que acabou de ser lido. A função possui um for que passa por todos os termos da tabela de símbolos verificando com o strcmp() se o buffer em questão já existe na lista de simbolos.

- **ETAPA 11 (identificar comparadores e nomes de bibliotecas)**

Essa etapa serviu para classificar os nomes de biblioteca como (ID, count), adicionar um novo vetor chamado vetor_comp[ ] e adicionar uma nova categoria de token, os (COMP, count), neles são inclusos {">=", "<=", "==", "!=", ">", "<"}.

Já na parte de implementação. Verifiquei se após um ‘<’ ser lido o próximo caractere é diferente de um número ou um espaço em branco, se for diferente, tudo que estiver entre ‘<’ e ‘>’ será considerado um (ID, count). Entretanto, caso o próximo caractere for um número ou espaço em branco, ele seguirá normalmente o código sem passar (continue).

Criei um vetor possivel_comp[ ] juntamente a um if para que caso o caractere lido do arquivo for um ‘>’, ‘<’, ‘=’ ou ‘!’, o código verifique se o proximo_caractere é um ‘=’, para formar ‘>=’, ‘<=’, ’==’ ou ‘!=’, se não for um ‘=’ ele devolve o próximo caractere ficando apenas com ‘>’, ‘<’, ‘=’ ou ‘!’ e o adiciona no possivel_comp, mas se o proximo_caractere for sim um ‘=’, ambos serão adicionados no possivel_comp.

Após definir quem seria o possível comparador, basta jogar o possível comparador na função buscaComparador(), se achado, adiciona o seu token (COMP, count) na lista de tokens e na tabela de símbolos o seu símbolo literal.

*(buscaComparador() recebe como parâmetro um possível comparador, o vetor_comp[] e o tamanho do vetor).*

- **ETAPA 12 (classificar caracteres nao char_especiais e &, && ||)**

Classificar alguns caracteres especiais como (ID, count) ex: '.', '#', '@' ou '$'.

Essa etapa se deve ao fato de que no Excel disponibilizado alguns caracteres ficaram fora da classificação de tokens e não poderiam ser ignorados na leitura do arquivo (Eles precisam ser classificados de alguma forma).

Nesta etapa, também fiz com que o código reconhecesse && e || como tokens literais.

Dada a ordem que as coisas acontecem no código, primeiro verifica-se se o caractere é ‘/‘ ou ‘*’ para identificar um possível comentário, se não for, verifica se é ‘<’ para identificar uma possível biblioteca, se não, verifica se é uma aspa ‘ “ ‘ para identificar um bloco de texto, se não for, é um número, e se não for um número, é um comparador. Caso não seja nenhuma dessas opções, o caractere só pode ser um char_especial, então basta apenas descobrir qual é.

Adicionam-se condicionais logo após a verificação de comparadores, se o caractere lido pertencer ao vetor char_especiais[ ] e for um desses: '.', '#', '@' ou '$' o seu (ID, count) entrará na tabela de tokens e seu literal na tabela de símbolos, se o caractere nao for nenhum desses 4, seu token literal será adicionado a tabela de tokens.

Se por acaso o caractere não pertencer ao vetor char_especias[], só resta ser ‘&’ ou ‘|’. Então o proximo_caractere é lido e uma condicional é feita, caso o caractere seja & e o próximo & ou caractere seja | e o próximo |, ‘&&’ ou ‘||’ será adicionado diretamente na tabela de tokens. Mas se o caractere for diferente do proximo_caractere, seu token (ID, count) é adicionado na tabela de tokens, juntamente ao seu literal na lista de símbolos.

- **ETAPA 13 (separar números em inteiros e decimais)**

Separar números inteiros dos decimais na lista de tokens alterei a função is_number() para que recebe como parâmetro o buffer de palavra lida e uma variável recém-criada chamada decimal, essa função é capaz de identificar se uma determinada string é um número decimal ou não, se for decimal, adiciona o token (NUM_DEC, count) na lista de tokens e o número literal na tabela de símbolos, mas se o número for inteiro, adiciona o token (NUM_INT, count) na lista de tokens e o número literal na lista de símbolos. Caso a string

não seja um número, adiciona o símbolo e o token do ID em suas devidas listas.

- **ETAPA 14 (arrumar o contador dos números)**

Essa etapa foi para corrigir como os números eram vistos no código, antes eles recebiam o token (NUM_DEC ou NUM_INT, count) e o seu símbolo era o número literal. Porém na análise léxica os números não vão pra tabela de símbolos e seu token é no formato (NUM_DEC ou NUM_INT, número literal). Para mudar esse funcionamento, bastou remover a parte do código em que os números eram inseridos na tabela de símbolos e mudar o count pelo buffer de palavra (que representa o número).

## 6. PASSO A PASSO: CONSTRUÇÃO DO ANALISADOR COM LISTA ENCADEADA

Nos propusemos a desenvolver um analisador utilizando listas encadeadas. O objetivo é identificar e classificar os diferentes componentes de um código fonte, como tokens e símbolos, de forma eficiente e organizada. Para isso, construimos uma estrutura que nos permita armazenar essas informações de maneira dinâmica, facilitando a manipulação e o acesso durante o processo de análise.

- **ETAPA 1**

Nesta etapa, nós começamos a implementação do nosso analisador importando as bibliotecas necessárias e criando as estruturas que serão utilizadas para armazenar os tokens e símbolos. Utilizamos as bibliotecas stdio.h, stdlib.h, e string.h para permitir operações de entrada e saída, alocação de memória e manipulação de strings. Definimos a estrutura Token para representar cada lexema identificado, incluindo um tipo, um valor e um ponteiro para o próximo token na lista encadeada. Em seguida, criamos a estrutura Simbolo, que manterá informações sobre identificadores e suas posições no código.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct token {
    char tipo[20];
    char valor[100];
    struct token *prox;
} Token;

typedef struct simbolo {
    char id[100];
    int posicao;
    struct simbolo *next;
} Simbolo;
```

**Figura 6: Definição das Estruturas de Dados.**

- **ETAPA 2**

Na segunda etapa do nosso projeto, nós implementamos um conjunto de funções que desempenham papéis fundamentais na análise léxica do código-fonte. Essas funções são essenciais para identificar e classificar diferentes componentes da linguagem, garantindo que o nosso analisador funcione de maneira eficiente.

A primeira função que criamos foi a *biblioteca*, que verifica se um lexema corresponde a uma das bibliotecas padrão, como *stdio.h*, *stdlib.h* e *string.h*. Essa validação é importante, pois assegura que as bibliotecas necessárias sejam corretamente reconhecidas no código. Em seguida, implementamos a função *numero_inteiro*, que nos ajuda a determinar se um lexema representa um número inteiro. Através dela, nós validamos que todos os caracteres de um lexema sejam dígitos, assegurando que apenas valores inteiros sejam aceitos.

```c
int biblioteca(char *lexema) {
    char *bibliotecas[] = {"stdio.h", "stdlib.h", "string.h"};
    for (int i = 0; i < 3; i++) {
        if (strcmp(lexema, bibliotecas[i]) == 0) {
            return 1;
        }
    }
    return 0;
}
```

**Figura 7: Função Biblioteca.**

Nós também desenvolvemos a função *numero_decimal,* que é semelhante à anterior, mas especificamente para verificar se um lexema representa um número decimal. Com essa função, nós garantimos que um ponto decimal possa estar presente, mas somente uma vez no lexema, evitando assim formatações erradas. A função *palavra_reservada* foi outra adição significativa, pois ela verifica se um lexema é uma palavra reservada da linguagem, como *int, float* e *if*. Ao identificá-las, nós conseguimos garantir que esses termos sejam tratados corretamente e não sejam utilizados como identificadores.

**Figura 8: Funções de Número Inteiro e Decimal.**

```
int numero_inteiro(char *lexema) {
    for (int i = 0; lexema[i] != '\0'; i++) {
        if (lexema[i] < '0' || lexema[i] > '9') {
            return 0;
        }
    }
    return 1;
}

int numero_decimal(char *lexema) {
    int ponto_encontrado = 0;

    for (int i = 0; lexema[i] != '\0'; i++) {
        if (lexema[i] == '.') {
            if (ponto_encontrado) {
                return 0;
            }
            ponto_encontrado = 1;
        } else if (lexema[i] < '0' || lexema[i] > '9') {
            return 0;
        }
    }
    return ponto_encontrado;
}
```

A função *identificador* foi crucial para validar se um lexema segue as regras para identificadores. Com ela, nós asseguramos que os identificadores comecem com uma letra e que os caracteres subsequentes possam ser letras ou dígitos, evitando assim identificadores inválidos. Em seguida, implementamos a função *constante_texto*, que verifica se um lexema é uma constante de texto. Essa função valida que o lexema comece e termine com aspas duplas, importante para a identificação de strings no código-fonte.

```c
int palavra_reservada(char *lexema) {
    char *palavras_reservadas[] = {
        "int", "float", "char", "boolean", "void",
        "if", "else", "for", "while", "scanf",
        "println", "printf", "main", "return", "include"
    };
    for (int i = 0; i < 15; i++) {
        if (strcmp(lexema, palavras_reservadas[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int identificador(char *lexema) {
    if (!((lexema[0] >= 'A' && lexema[0] <= 'Z') ||
          (lexema[0] >= 'a' && lexema[0] <= 'z'))) {
        return 0;
    }
    for (int i = 1; lexema[i] != '\0'; i++) {
        if (!(((lexema[i] >= 'A' && lexema[i] <= 'Z') ||
               (lexema[i] >= 'a' && lexema[i] <= 'z')) ||
               (lexema[i] >= '0' && lexema[i] <= '9'))) {
            return 0;
        }
    }
    return 1;
}
```

**Figura 9: Funções de Palavras Reservadas e Identificador.**

Além disso, nós criamos a função *comentario*, responsável por identificar comentários de linha única, que começam com //. Isso nos permite ignorar o conteúdo de comentários

durante a análise. A função *simbolo_especial* foi implementada para verificar a presença de símbolos especiais, como parênteses, chaves e ponto e vírgula. Esses símbolos são essenciais para a estruturação do código e precisam ser reconhecidos corretamente.

```c
int constante_texto(char *lexema) {
    return lexema[0] == '"' && lexema[strlen(lexema) - 1] == '"';
}

int comentario(char *lexema) {
    return (lexema[0] == '/' && lexema[1] == '/');
}

int simbolo_especial(char lexema) {
    char simbolos[] = {'(', ')', '{', '}', '[', ']', ';', ',', '.', '#'};
    for (int i = 0; i < 10; i++) {
        if (lexema == simbolos[i]) {
            return 1;
        }
    }
    return 0;
}

int operador_comparacao(char *lexema) {
    char *simbolos[] = {"==", ">=", "<=", ">", "<", "!="};
    for (int i = 0; i < 6; i++) {
        if (strcmp(lexema, simbolos[i]) == 0) {
            return 1;
        }
    }
    return 0;
}
```

**Figura 10: Funções de Constantes de Texto, Comentário, Simbolos Especiais e Operadores de Comparacao.**

Para lidar com os operadores, nós desenvolvemos várias funções: a *operador_comparacao*, que identifica operadores de comparação como ==, >= e <; a *operador_logico*, que reconhece operadores lógicos como *&&,* || e *!;* e a operador_aritmetico, que permite a identificação de operadores aritméticos como +, -, *, /, e %. A correta identificação desses operadores é crucial para a execução de operações matemáticas e

lógicas no código. Por fim, nós implementamos a função operador_atribuicao, que verifica se um lexema representa um operador de atribuição, que é o sinal =. Esse operador é essencial para a atribuição de valores a variáveis.

**Figura 11: Funções dos Operadores Lógicos, Aritimeticos e Atribuição.**

```c
int operador_logico(char *lexema) {
    char *simbolos[] = {"&&", "||", "!"};
    for (int i = 0; i < 3; i++) {
        if (strcmp(lexema, simbolos[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int operador_aritmetico(char lexema) {
    char simbolos[] = {'+', '-', '*', '/', '%'};
    for (int i = 0; i < 5; i++) {
        if (lexema == simbolos[i]) {
            return 1;
        }
    }
    return 0;
}

int operador_atribuicao(char lexema) {
    return lexema == '=';
}
```

- **ETAPA 3**

Na terceira etapa, nós implementamos funções essenciais para gerenciar tokens e símbolos na lista encadeada. A função *add_token* é responsável por adicionar novos tokens à lista. Quando chamamos essa função, alocamos memória para um novo token e copiamos o tipo e o valor fornecidos. Se a lista de tokens estiver vazia, o novo token se torna o primeiro elemento. Caso contrário, percorremos a lista até encontrar o final e, então, anexamos o novo token. Essa abordagem nos permite armazenar sequencialmente os tokens à medida que os encontramos.

```c
void add_token(Token **head, char *tipo, char *valor) {
    Token *novo_token = (Token *)malloc(sizeof(Token));
    strcpy(novo_token->tipo, tipo);
    strcpy(novo_token->valor, valor);
    novo_token->prox = NULL;

    if (*head == NULL) {
        *head = novo_token;
    } else {
        Token *temp = *head;
        while (temp->prox != NULL) {
            temp = temp->prox;
        }
        temp->prox = novo_token;
    }
}
```

**Figura 12: Função para Adicionar Tokens na Lista de Tokens.**

A função *add_simbolo* tem a responsabilidade de adicionar novos símbolos à lista. Primeiro, verificamos se o símbolo já existe na lista; se sim, a função retorna sem fazer nada. Se não existir, alocamos memória para um novo símbolo, copiamos o identificador e determinamos sua posição na lista. O novo símbolo é então inserido no final da lista ou se torna o primeiro elemento se a lista estiver vazia. Esse gerenciamento eficiente de símbolos é crucial para a construção do nosso analisador.

```c
void add_simbolo(Simbolo **head, char *id) {
    Simbolo *temp = *head;

    while (temp != NULL) {
        if (strcmp(temp->id, id) == 0) {
            return;
        }
        temp = temp->next;
    }

    Simbolo *novo_simbolo = (Simbolo *)malloc(sizeof(Simbolo));
    strcpy(novo_simbolo->id, id);
```

```
    int posicao = 1;
    temp = *head;
    while (temp != NULL) {
        posicao++;
        temp = temp->next;
    }
    novo_simbolo->posicao = posicao;
    novo_simbolo->next = NULL;

    if (*head == NULL) {
        *head = novo_simbolo;
    } else {
        Simbolo *current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = novo_simbolo;
    }
}
```

**Figura 13 e 14: Função para Adicionar Simbolos na Lista "Tabela" de Simbolos.**

- **ETAPA 4**

Na quarta etapa a função *processar_lexema* recebe um lexema e determina seu tipo, chamando a função apropriada para adicioná-lo à lista de tokens. Essa função verifica se o lexema é uma palavra reservada, identificador, número inteiro, número decimal, literal, comentário, operador de comparação, operador lógico, operador aritmético, operador de atribuição, símbolo especial ou uma biblioteca. Dependendo do resultado, o lexema é adicionado à lista de tokens e, no caso de identificadores, também à lista de símbolos.

```
void processar_lexema(char *lexema, Token **tokens, Simbolo **simbolos) {
    if (palavra_reservada(lexema)) {
        add_token(tokens, "Reservado", lexema);
    } else if (identificador(lexema)) {
        add_token(tokens, "Identificador", lexema);
        add_simbolo(simbolos, lexema);
    } else if (numero_inteiro(lexema)) {
        add_token(tokens, "Inteiro", lexema);
    } else if (numero_decimal(lexema)) {
        add_token(tokens, "Decimal", lexema);
```

```
    } else if (constante_texto(lexema)) {
        add_token(tokens, "Literal", lexema);
    } else if (comentario(lexema)) {
        add_token(tokens, "Comentario", lexema);
    } else if (operador_comparacao(lexema)) {
        add_token(tokens, "Comparacao", lexema);
    } else if (operador_logico(lexema)) {
        add_token(tokens, "Logico", lexema);
    } else if (operador_aritmetico(lexema[0])) {
        add_token(tokens, "Aritmetico", lexema);
    } else if (operador_atribuicao(lexema[0])) {
        add_token(tokens, "Atribuicao", lexema);
    } else if (simbolo_especial(lexema[0])) {
        add_token(tokens, "Especial", lexema);
    } else if(biblioteca(lexema)){
        add_token(tokens, "Biblioteca", lexema);
    }
}
```

**Figura 15 e 16: Função para Identificar corretamente o Lexema.**

A função *analisar_linha* processa uma linha de código, extraindo lexemas e enviando-os para a função *processar_lexema*. Ela percorre cada caractere da linha, reconhecendo espaços, comentários e diferentes tipos de símbolos, e acumulando caracteres em um buffer (lexema) até que um delimitador seja encontrado. Quando um delimitador é encontrado, o lexema é processado e adicionado à lista de tokens conforme necessário. A função também lida com casos especiais, como strings e comentários, garantindo que sejam corretamente identificados e processados.

```
void analisar_linha(char *linha, Token **tokens, Simbolo **simbolos) {
    char lexema[100];
    int i = 0, j = 0;

    while (linha[i] != '\0') {
        char c = linha[i];

        if (c == ' ' || c == '\t') {
            if (j > 0) {
                lexema[j] = '\0';
                processar_lexema(lexema, tokens, simbolos);
                j = 0;
            }
        }
```

```
    else if (c == '/' && linha[i + 1] == '/') {
        if (j > 0) {
            lexema[j] = '\0';
            processar_lexema(lexema, tokens, simbolos);
            j = 0;
        }
        i += 2;
        while (linha[i] != '\n' && linha[i] != '\0') {
            lexema[j++] = linha[i++];
        }
        lexema[j] = '\0';
        add_token(tokens, "Comentario", lexema);
        j = 0;
        continue;
    }
    else if (c == '"' || c == '\'') {
        if (j > 0) {
            lexema[j] = '\0';
            processar_lexema(lexema, tokens, simbolos);
            j = 0;
        }
        lexema[j++] = c;
        i++;
        while (linha[i] != c && linha[i] != '\0') {
            lexema[j++] = linha[i++];
        }
        if (linha[i] == c) {
            lexema[j++] = linha[i];
            lexema[j] = '\0';
            add_token(tokens, "Literal", lexema);
            j = 0;
        }
    }
```

**Figura 17 e 18: Uma parte Inicial da Função Analisar do Código.**

- **ETAPA 5**

Nesta etapa, implementamos funções para imprimir a lista de tokens e a tabela de símbolos. Essas funções são fundamentais para visualizar os resultados da análise lexical e entender a estrutura dos dados processados.

A função *print_tokens* imprime uma tabela que lista todos os tokens identificados, categorizando-os por tipo e valor. Ela verifica se o token é um identificador, um número, um literal, uma comparação ou um comentário, e imprime informações adicionais, como a

posição do identificador na tabela de símbolos.

```c
void print_tokens(Token *head, Simbolo *simbolos) {
    Token *temp = head;

    printf("Lista de Tokens:\n");
    printf("%-20s %-30s\n", "Tipo", "Token");
    printf("------------------------------\n");

    while (temp != NULL) {
        if (strcmp(temp->tipo, "Identificador") == 0) {
            Simbolo *s = simbolos;
            while (s != NULL) {
                if (strcmp(s->id, temp->valor) == 0) {
                    printf("%-20s (ID, %d)\n", temp->tipo, s->posicao);
                    break;
                }
                s = s->next;
            }
        } else if (strcmp(temp->tipo, "Inteiro") == 0 || strcmp(temp->tipo, "Decimal")
            printf("%-20s (NUM, %s)\n", temp->tipo, temp->valor);
        } else if (strcmp(temp->tipo, "Literal") == 0) {
            printf("%-20s (Literal, %s)\n", temp->tipo, temp->valor);
        } else if (strcmp(temp->tipo, "Comparacao") == 0) {
            printf("%-20s (Comp, %s)\n", temp->tipo, temp->valor);
        } else if (strcmp(temp->tipo, "Comentario") == 0) {
            printf("%-20s // %s\n", temp->tipo, temp->valor);
        } else {
            printf("%-20s %-30s\n", temp->tipo, temp->valor);
        }
        temp = temp->prox;
    }
    printf("------------------------------\n");
}
```

**Figura 19: Função para Realizar a Impressão da Lista de Tokens.**

A função *print_simbolos* exibe a tabela de símbolos, mostrando a posição e o ID de cada símbolo. Isso permite que o usuário visualize todos os identificadores que foram registrados durante a análise.

```c
void print_simbolos(Simbolo *head) {
    Simbolo *temp = head;
    printf("Tabela de Simbolos:\n");
    printf("%-5s %-20s\n", "Posicao", "ID");
    printf("------------------------------\n");
    while (temp != NULL) {
        printf("%-5d %-20s\n", temp->posicao, temp->id);
        temp = temp->next;
    }
    printf("------------------------------\n");
}
```

**Figura 20: Função para Realizar a Impressão da Tabela de Simbolos.**

- **ETAPA 6**

Nesta etapa final implementamos a função principal analisador, que é responsável por ler um arquivo de código fonte e processar cada linha para enviar para o analisador linha a linha para gerar a lista de tokens e a tabela de símbolos. Também incluímos a função main, que abre o arquivo e inicia o processo de análise.

A função *analisador* lê o arquivo linha por linha, chamando a função *analisar_linha* para processar cada linha e atualizar a lista de tokens e a tabela de símbolos. Após a leitura completa do arquivo, as funções *print_tokens* e *print_simbolos* são chamadas para exibir os resultados.

```
void analisador(FILE *arquivo) {
    char linha[1024];
    Token *lista_token = NULL;
    Simbolo *tabela_simbolo = NULL;

    while (fgets(linha, sizeof(linha), arquivo) != NULL) {
        analisar_linha(linha, &lista_token, &tabela_simbolo);
    }

    print_tokens(lista_token, tabela_simbolo);
    print_simbolos(tabela_simbolo);
}
```

**Figura 21: Função para Realizar a Impressão da Lista de Tokens.**

A função *main* é a entrada do programa. Ela tenta abrir um arquivo chamado index.c para leitura. Se o arquivo for aberto com sucesso, a função analisador é chamada. Caso contrário, uma mensagem de erro é exibida. O arquivo é fechado e o programa termina.

```
int main() {
    FILE *arquivo = fopen("index.c", "r");

    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    analisador(arquivo);

    fclose(arquivo);

    return 0;
}
```

**Figura 22: Main para Inicializar o Analisador Léxico.**