Universidade Estadual do Oeste do Panará – UNIOESTE

Curso de Ciência da Computação

Disciplina de Compiladores

Professora Camile Frazão Bordini

João Pedro Aragão Teixeira

Leticia Zanellatto de Oliveira

Análise Léxica – Jovial C++

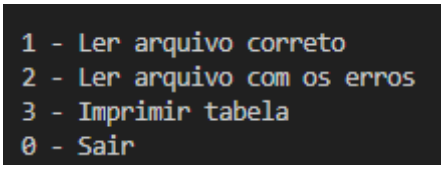
Foz do Iguaçu

2023

Sumário

1. Funcionamento

O *software* inicia com o cabeçario contendo o nome do compilador e analisador léxico (Jovial C++). Em seguida é visto um menu com 4 opções para o usuário escolher:



Na opção 1, o usuário será direcionado para escrever o nome do arquivo fonte sem os erros léxicos (fonte1.txt). Inicialmente o programa irá apenas ler o arquivo e voltar para o menu, para o usuário ver a análise ele terá que selecionar a opção 3 que imprimirá na tela a tabela com a análise.

Caso o usuário queira ver os erros léxicos, ele terá que escolher a opção 2, assim aparecerá para que o usuário escreva o nome do arquivo fonte com alguns erros léxicos (fonte2.txt). E logo em seguida será impresso na tela em qual linha se encontra os erros e onde está o caractere errado. Caso o usuário queira ver a análise léxica também, ao voltar para o menu ele deverá escolher a opção 3.

Se o usuário quiser encerrar o programa, basta selecionar a opção 0, mas se ele quiser fazer outras análises com outros arquivos, é só ele entrar com novos arquivos e assim seguir as instruções com esses novos arquivos.

2. Descrição da Linguagem

A linguagem Jovial C++ tem as seguintes classes de Tokens:

Classes de Token	Expressão Regular
Digito – Representação diretas de valores numéricos na linguagem.	[0-9]+ {Digito}+.{Digito}+
Letra – Palavras que são formadas apenas por uma letra.	[a-zA-Z]
Identificador(Id) – Palavras que são formadas por letras ou por letras concatenadas com dígitos.	[a-zA-Z][a-zA-Z0-9]*
Whitespace – Espaço em branco que a linguagem reconhece.	[\t]
Enter – Quebra linha da linguagem.	[\n]
Aspas – Símbolo para começar uma string.	[“]
String – Conjunto de caracteres que a linguagem reconhece.	\"[^\"]*\"
Condicional – Comandos de condição.	If | else
Repetição – Comando para repetição.	for | while
Tipo de Dado – Representação dos tipos de dado das variáveis.	char|char* int|int* float|float* double|double* {Id}"*"

Funções – Funções da linguagem.	void | main | print | scan
Comando – Comandos da linguagem.	return | import | define
Operador de Contas – Símbolos que fazem operações entre duas variáveis.	"+" | "-" | "/" | "*" | "%" | "!" | "|" |"^" | "&"
Operador Lógicos – Símbolos para realizar operações lógicas.	"&&" | "||" | "<<" | ">>"
Operador de Incremento/Decremento – Símbolos que realiza operação de incremento e decremento.	"++" | "--"
Operadores de Atribuições – Símbolos que fazem operações de atribuições.	"=" | "+= | "-=" | "*=" | "/=" | "%="
Operadores de Comparação – Símbolos que fazem tipos de comparação.	"==" | "!=" | "<" | ">" | ">=" | "<="
Pontuação - Símbolos usados para estruturar o código.	"," | ";" | ":"
Separadores – Símbolos usados para separar elementos em uma expressão ou declaração.	"(" | ")" | "{" | "}" | "["
Encadeamento – Símbolo para encadear variáveis.	"." | "->"
Comentário – Trechos de texto ignorados pelo compilador.	"/*"[^'*/']*"*/" "//"[^\n]*\n
Error – Caracteres não reconhecidos pelo compilador.	[@$¨~çÇ§¹²³£¢¬Ğº`]

3. Tratamento de Erros

O tratamento do erro é feito pela função *"verificaErro"*, na qual recebe a lista, a *string* a ser verificada e a primeira linha do arquivo.

```
int verificaErro(Item *item, char* str, int linha){
    char erro[] = "@$¨~çÇ§¹²³£¢¬Ğº`";
    int pos = strcspn(str, erro);
    if(pos == strlen(str)){
        return 0;
    }else{
        printf("\n\tERRO NA LINHA %d CARACTER %d: %s\n\n", linha, pos+1, str);
        return pos;
    }
}
```

Depois é declarado a *string* erro que é mesma definida como a classe do *Token Erro*, em seguida é usado a função *strcspn* que pega a posição do caractere que se repete. Se a posição desse caractere for igual ao tamanho da *string*, significa que não tem erro, logo a função retorna, caso contrário é feito um print da *string* mostrando em qual linha está o erro e a posição dele na *string*, retornando à posição do erro.

4. Funções do Software e sua construção

No *software*, tem-se um arquivo principal, o *analisador.l*, no qual está todas as regras léxicas e o *main* em *C*. Nele também tem algumas funções do próprio *FLEX*, como o *"yytext"* que guarda a *string* do *token*, a função *"yywrap"*, que fica lendo até o

fim de arquivo, o *"yyin"* que é uma variável do tipo *FILE* do próprio *FLEX* e a chamada da função *"yylex"* que retorna o tipo do *token* que foi obtido a partir das regras que foram estabelecidas antes.

As funções para fazer essa análise léxica estão declaradas e documentadas na biblioteca *"lista.h"* assim como a estrutura usada. No *"lista.c"* tem as funções programadas. Há quatro funções principais: *"inserir"*, *"limpa"*, *"busca"*, *"imprimir" e "verificaErro"*. A construção do código foi feita no *Visual Studio Code*, ultilizando a linguagem C e o gerador de análise léxica *FLEX/Bison*

4.1. Função *inserir*

A função recebe três parâmetros: uma lista, a *string* do *token* e o tipo do *token.* A função começa chamando a *"busca",* verificando se tem alguém na lista que é igual a *string* recebida, se não tive insere esse elemento na cabeça da lista e retorna ele, se tiver não faz nada e retorna a lista.

```
Item *inserir(Item *item, char *nome, char *token){
    if(busca(item, nome)==NULL){
        Item *aux = (Item *)malloc(sizeof(Item));
        strcpy(aux->nome, nome);
        strcpy(aux->token, token);
        aux->prox = item;
        return aux;
    }
    return item;
}
```

4.2. Função *limpa*

A função tem como parâmetro uma lista. Ela começa verificando se a lista não é vazia, se não for, irá chamar recursivamente a função indo para o próximo nó e liberando a memória do anterior, feito isso a função retornará NULL. Já se a lista for vazia a função irá retorna NULL.

```
Item *limpa(Item *item){
    if(!vazia(item)){
        item->prox = limpa(item->prox);
        free(item);
        return NULL;
    }
    return NULL;
}
```

4.3. Função *busca*

A função recebe apenas uma lista e a *string* a ser comparada, assim é criado uma variável do tipo lista auxiliar que recebe a lista do parâmetro,

em seguida percorre a variável auxiliar e compara a *string* dela com a da variável, se for igual retorna a *aux*, se não retorna a lista completa.

```
Item *busca (Item *item, char* str){
    Item *aux = item;
    for(; aux != NULL; aux = aux->prox){
        if(strcmp(aux->nome, str) == 0) return aux;
    }
    return aux;
}
```

4.4. Função *imprimir*

Essa função é básica, ela serve para imprimir a tabela dos dados, ou seja, ela imprime o token e o tipo dele. A função recebe apenas a lista e verifica se essa lista está vazia, se tiver não retorna nada, se não, ela percorre toda a lista e imprime de trás para frente, pois a função inserir fez a inserção na cabeça, ou seja, iria imprimir ao contrário, desse jeito é impresso certo na ordem que as palavras aparecem no arquivo.

```
void imprimir(Item *item){
    if(vazia(item)){
        return;
    }else{
        imprimir(item->prox);
        printf("[%s] --> ", item->nome);
        printf("[%s]\n", item->token);
    }
}
```

5. Referencias

BORDINI, Camile. Compiladores Análise Léxica. UNIOESTE – Foz Ciência da Computação. 2023.

BORDINI, Camile. Compiladores Introdução. UNIOESTE – Foz Ciência da Computação. 2023.