Universidade Estadual do Oeste do Paraná - Unioeste
Ciência da Computação
Disciplina de Compiladores

Trabalho 2

Análise Léxica e Sintática - Jovial C++

João Pedro Aragão Teixeira

Leticia Zanellatto de Oliveira

Foz do Iguaçu

2024

Sumário

1 Funcionamento do Software

Para compilar o código fonte usando Yacc (Bison) em conjunto com o analisador léxico usando Flex, o usuário precisa executar os seguintes comandos na prompt:

- chcp 65001 // comando para que a impressão da tabela saia corretamente
- yacc -d .\sintatico.y
- flex .\lexico.l
- gcc *.c
- .\a.exe

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

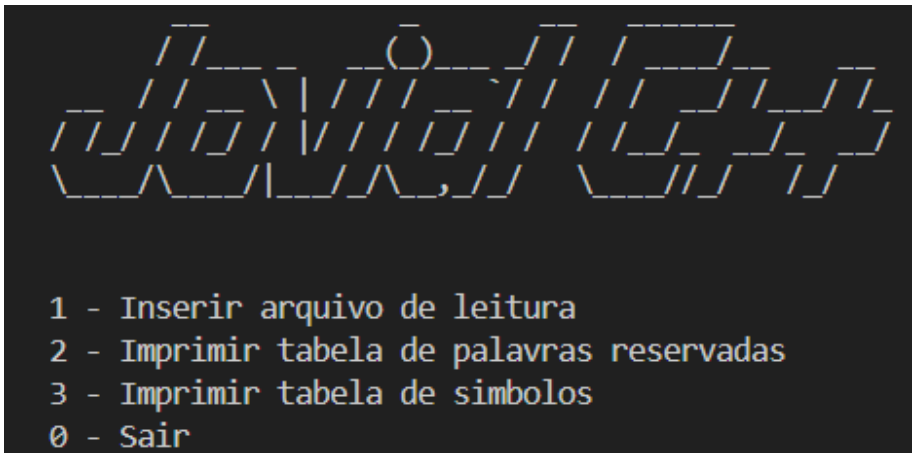

Figura 1: Menu

Na opção 1, o usuário será solicitado a inserir o nome do arquivo que deseja analisar. Após a leitura do arquivo, o programa irá gerar um arquivo chamado «arvore.txt» com a impressão da árvore de derivação do código. Além disso, serão exibidas na tela as derivações feitas pelo parser, indicando se o conteúdo do arquivo está correto e a quantidade de linhas analisadas.

O usuário poderá selecionar a opção 2 para visualizar a tabela de palavras reservadas gerada durante a análise do arquivo de código. Caso ele deseje visualizar a tabela símbolos gerada durante a análise do arquivo de código, basta selecionar a opção 3.

Se o usuário desejar encerrar o programa, basta selecionar a opção 0. No entanto, se ele quiser realizar outras análises com novos arquivos, poderá inseri-los e seguir as instruções fornecidas para análise.

2 Descrição da Linguagem

A linguagem desenvolvida tem as seguintes classe de Tokens

Classes de Token	**Expressão Regular**
Digito – Representação direta de valores numéricos na linguagem.	[0-9]+
Letra – Palavras formadas apenas por uma letra.	[a-zA-Z]
Identificador(Id) – Palavras formadas por letras ou letras concatenadas com dígitos.	[a-zA-Z][a-zA-Z0-9]*
Whitespace – Espaço em branco que a linguagem reconhece.	[\t]
Enter – Quebra de 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.	“+” | “-” | “/” | “*” | “%” | “!” | “|” | “^” | “&”
Operadores Lógicos – Símbolos para realizar operações lógicas.	“&&” | “||” | “«” | “»”
Operador de Incremento/Decremento – Símbolos que realizam 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£¢¬°‘]
Números inteiros – Conjunto de dígitos numéricos.	[0-9]+
Número real – Número com parte decimal.	Digito+.Digito+

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.

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.

```c
// Descrição: Verifica se uma string contém caracteres inválidos.
// Pré-Condição: str deve ser uma string válida, linha deve ser um número válido
// Pós-Condição: retorna a posição do primeiro caractere inválido ou 0 se não há erro
int verificaErro(char* str, int linha){
   char erro[] = "@$¨~çÇ§¹²³£¢¬ɢº`";
   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;
   }
}
```

Figura 2: Funçao verificaErro

4 Funções Léxicas

No código-fonte, há um arquivo principal chamado lexico.l, que contém todas as regras léxicas e o main em C. Nele também estão presentes algumas funções do próprio FLEX, como “yytext”, que armazena a string do token, a função “yywrap”, que continua lendo até o final do 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 obtido a partir das regras estabelecidas anteriormente.

Há duas estruturas usadas na implementações de tabelas de hash. A estrutura Item representa um elemento que será armazenado na tabela hash, contendo um nome, um token associado a esse nome, o número da linha onde foi encontrado e um ponteiro para o próximo item na mesma posição da tabela. Já a estrutura Hash é a própria tabela hash, que armazena um tamanho (tam) e um vetor de ponteiros para Item, representando os elementos armazenados na tabela.

As funções para realizar essa análise léxica estão declaradas e documentadas na biblioteca "hash.h", assim como a estrutura utilizada. No arquivo "hash.c", as funções estão programadas. Existem quatro funções principais: "criar", "inserir", "limpa", "busca", "imprimir"e "verificaErro".

4.1 Função Hashing e criaHash

A primeira função calcula o valor hash de uma string. Para isso, percorre cada caractere da string nome e soma o valor ASCII de cada caractere. Em seguida, utiliza o operador de módulo (%) para reduzir esse valor para que ele caiba dentro do tamanho da tabela. O resultado desse cálculo é o valor hash da string, que é retornado pela função.

```c
// Descrição: Calcula o valor hash de uma string.
// Pré-Condição: h deve ser um ponteiro válido, nome deve ser uma string válida
// Pós-Condição: retorna o valor hash calculado
int hashing(Hash *h, char *nome){
    int t = 0;
    for(int i = 0; nome[i]; i++){
        t += (int) nome[i];
    }
    return t%h->tam;
}

// Descrição: Cria uma nova tabela hash.
// Pré-Condição: tam deve ser um valor válido
// Pós-Condição: retorna a nova tabela hash criada
Hash *criarHash(int tam){
    Hash *h = (Hash*) malloc (sizeof(Hash));
    h->tam = tam;
    h->vet = (Item**) malloc (sizeof(Item*)*tam);
    for(int i = 0; i < tam; i++){
        h->vet[i] = NULL;
    }
    return h;
}
```

Figura 3: Funções para criação da Tabela Hash

A segunda função cria uma nova tabela hash com o tamanho especificado por tam. Para isso, aloca memória para a estrutura Hash e para o vetor de itens vet dentro dela. Em seguida, inicializa cada posição do vetor com NULL, indicando que inicialmente não há itens na tabela. Por fim, retorna um ponteiro para a nova tabela hash criada.

4.2 Função InserirHash e inserir

A função inserirHash recebe como parâmetros um ponteiro para a tabela hash h, uma string nome e uma string token. Ela calcula a chave (key) para a string nome usando a função hashing. Em seguida, ela insere o novo item na posição key do vetor de itens da tabela hash (h->vet[key]) utilizando a função inserir.

Por outro lado, a função inserir recebe como parâmetros um ponteiro para um item da lista encadeada item, uma string nome e uma string token. Primeiramente, ela verifica se já existe um

item com o mesmo nome na lista. Se não existir, ela cria um novo item (aux), copia as strings nome e token para esse novo item e o insere no início da lista, retornando o ponteiro para o novo item inserido.

Dessa forma, a função inserirHash é responsável por inserir um novo item na tabela hash, enquanto a função inserir é responsável por inserir um novo item na lista encadeada. Juntas, essas funções permitem a inserção de itens em uma estrutura de dados de tabela hash com lista encadeada.

```c
// Descrição: Insere um novo item na tabela hash.
// Pré-Condição: h deve ser um ponteiro válido, nome e token devem ser strings válidas
// Pós-Condição: insere o novo item na tabela hash
void inserirHash(Hash *h, char *nome, char *token){
   int key = hashing(h, nome);
   h->vet[key] = inserir(h->vet[key], nome, token);
}

// Descrição: Insere um novo item na lista encadeada.
// Pré-Condição: item deve ser um ponteiro válido, nome e token devem ser strings válidas
// Pós-Condição: insere o novo item na lista encadeada
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;
}
```

Figura 4: Funções para inserção na tabela Hash

4.3 Funcão limpa

A função limpa é responsável por liberar a memória ocupada por uma lista encadeada de itens. Ela recebe como parâmetro um ponteiro para o primeiro item da lista (Item *item). A função verifica se a lista não está vazia (!vazia(item)), ou seja, se o ponteiro item não é nulo. Se a lista não estiver vazia, a função chama recursivamente a si mesma passando como parâmetro o próximo item da lista (item->prox). Isso é feito para percorrer toda a lista até o último item. Após chegar ao último item, a função libera a memória ocupada por esse item (free(item)) e retorna NULL, indicando que a lista está vazia.

Já a função limpaHash é responsável por liberar a memória ocupada por uma tabela hash. Ela recebe como parâmetro um ponteiro para a tabela hash (Hash *h). Primeiramente, a função verifica se a tabela hash não é nula (h != NULL). Em seguida, ela percorre cada posição do vetor de itens da tabela hash (h->vet) usando um loop for. Para cada posição do vetor, a função chama a função limpa para liberar a memória ocupada pela lista encadeada de itens nessa posição. Após percorrer todas as posições do vetor, a função finalmente libera a memória ocupada pela tabela hash (free(h)) e retorna NULL, indicando que a tabela foi liberada da memória.

Essas funções são importantes para garantir a correta liberação de memória alocada dinamicamente, evitando vazamentos de memória e mantendo o programa mais eficiente em termos de uso de recursos.

```c
// Descrição: Libera a memória ocupada pela lista encadeada.
// Pré-Condição: item deve ser um ponteiro válido
// Pós-Condição: libera a memória ocupada pela lista encadeada
Item *limpa(Item *item){
    if(!vazia(item)){
        item->prox = limpa(item->prox);
        free(item);
    }
    return NULL;
}

// Descrição: Libera a memória ocupada pela tabela hash.
// Pré-Condição: h deve ser um ponteiro válido
// Pós-Condição: libera a memória ocupada pela tabela hash
Hash *limpaHash(Hash *h){
    if(h != NULL){
        for(int i = 0; i < h->tam; i++){
            h->vet[i] = limpa(h->vet[i]);
        }
        free(h);
    }
    return NULL;
}
```

Figura 5: Funções para liberar a memoria

4.4 Função busca

A função busca é responsável por procurar um item na lista encadeada a partir de um nome específico. Ela recebe como parâmetros um ponteiro para o primeiro item da lista (Item *item) e uma string representando o nome a ser buscado (char* str).

A função começa criando um ponteiro auxiliar (Item *aux) e o inicializa com o ponteiro para o primeiro item da lista (item). Em seguida, entra em um loop for que percorre a lista encadeada enquanto o ponteiro aux não for nulo. A cada iteração, o ponteiro aux é movido para o próximo item da lista (aux = aux->prox).

```c
// Descrição: Busca um item na lista encadeada.
// Pré-Condição: item deve ser um ponteiro válido, str deve ser uma string válida
// Pós-Condição: retorna o item encontrado ou NULL se não encontrado
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;
}
```

Figura 6: Funções para busca na lista encadeada

Dentro do loop, a função verifica se o nome do item atual (aux->nome) é igual à string buscada (str) usando a função strcmp. Se os nomes forem iguais, a função retorna o ponteiro para esse item (return aux), indicando que o item foi encontrado.

Caso o loop termine sem encontrar o item (ou seja, o ponteiro aux se torna nulo), a função retorna NULL, indicando que o item não foi encontrado na lista.

4.5 Função imprimir e imprimiHash

A função imprimir(Item *item) é responsável por imprimir os elementos de uma lista encadeada de forma recursiva. Ela recebe como parâmetro um ponteiro para o primeiro elemento da lista. Primeiro ela verifica se a lista está vazia. Se estiver vazia, a função simplesmente retorna, pois não há nada para imprimir. Se a lista não estiver vazia, a função chama a si mesma recursivamente, passando como parâmetro o próximo elemento da lista (item->prox). Isso garante que todos os elementos da lista sejam impressos. Após a chamada recursiva, a função imprime na tela o token e o nome do elemento atual, formatando-os para ocupar 20 caracteres à esquerda e à direita, respectivamente. Isso garante que a saída seja alinhada e fácil de ler.

A função imprimirHash(Hash *h) é responsável por imprimir os itens de uma tabela hash, utilizando a função imprimir para imprimir cada lista encadeada. Ela recebe como parâmetro um ponteiro para a tabela hash. Para cada posição da tabela hash (h->vet[i]), a função chama a função imprimir, passando o ponteiro para a lista encadeada dessa posição. Isso resulta na impressão de todos os elementos da tabela hash. Cada posição da tabela hash é representada como uma lista encadeada, e a função imprimir é responsável por imprimir os elementos dessa lista de forma formatada.

```c
// Descrição: Imprime os itens da lista encadeada.
// Pré-Condição: item deve ser um ponteiro válido
// Pós-Condição: imprime os itens da lista encadeada
void imprimir(Item *item){
    if(vazia(item)){
        return;
    }else{
        imprimir(item->prox);
        printf("| %-20s |", item->token);
        printf("%20s |\n", item->nome);
    }
}

// Descrição: Imprime os itens da tabela hash.
// Pré-Condição: h deve ser um ponteiro válido
// Pós-Condição: imprime os itens da tabela hash
void imprimirHash(Hash *h){
    for(int i = 0; i<h->tam; i++){
        imprimir(h->vet[i]);
    }
}
```

Figura 7: Funções para impressão

5 Parte Sintática

A análise sintática é responsável por verificar se o código fonte está de acordo com a gramática da linguagem de programação. Essa análise foi realizada por meio do Bison, um gerador de

analisadores sintáticos que utiliza a notação BNF (Backus-Naur Form) para descrever a gramática da linguagem e sendo capaz de reconhecer a estrutura do código fonte e gerar uma árvore de derivação que representa essa estrutura. A descrição da gramática da linguagem JovialC++ utilizando a notação BNF ficou da seguinte forma:

```
<inicio> ::= <definicoes> | ;
<definicoes> ::= <dec_imp> <definicoes> | <dec_def> <definicoes> | <seq>;
<seq> ::= <dec> <pvirg> | <dec> <pvirg> <seq>;
<dec> ::= main | <dec_cond> | <dec_rep> | <dec_atr> | <dec_lei> | <dec_esc> | <exp> | <func> | <dec_ret> | <comm> | <string>;
<dec_cond> ::= cond_if <abre_p> <exp> <fecha_p> <abre_c> <seq> <fecha_c> | cond_if <abre_p> <exp> <fecha_p> <abre_c> <seq> <fecha_c> cond_else <abre_c> <seq> <fecha_c>;
<dec_rep> ::= rep <abre_p> <dec_atr> <pvirg> <exp> <pvirg> <dec_atr> <fecha_p> <abre_c> <seq> <fecha_c> | rep <abre_p> <exp> <fecha_p> <abre_c> <seq> <fecha_c>;
<dec_atr> ::= id <sinal_atr> <exp> | <tipo> id <sinal_atr> <exp> | id <sinal_atr>;
<dec_lei> ::= scan <abre_p> id <fecha_p>;
<dec_esc> ::= print <abre_p> <exp> <fecha_p>;
<dec_ret> ::= ret <exp>;
<dec_imp> ::= imp <comp> bib <comp>;
<dec_def> ::= def id <exp>;
<exp> ::= <exp_simples> <comp> <exp_simples> | <exp_simples> | <exp_simples> <logic> <exp_simples>;
<func> ::= <tipo> id <abre_p> <lista_parametros> <fecha_p> <abre_c> <seq> <fecha_c> | <tipo> id <abre_p> <fecha_p> <abre_c> <seq> <fecha_c> | <tipo> main <abre_p> <fecha_p> <abre_c> <seq> <fecha_c>;
<lista_parametros> ::= <parametro> | <parametro> <virgula> <lista_parametros>;
<parametro> ::= <tipo> id;
<exp_simples> ::= <exp_simples> <soma> <termo> | <termo>;
<termo> ::= <termo> <mult> <termo> | <fator>;
<fator> ::= <abre_p> <exp> <fecha_p> | <numero> | id;
<rep> ::= FOR_TOK | WHILE_TOK;
<sinal_atr> ::= ATR_TOK | INC_TOK | DEC_TOK | ATR_SM_TOK | ATR_DC_TOK | ATR_MT_TOK | ATR_DV_TOK | ATR_MD_TOK;
<comp> ::= MEN_TOK | IG_TOK | MAI_TOK | DIF_TOK | MAI_IG_TOK | MEN_IG_TOK;
<mult> ::= MULT_TOK | DIV_TOK;
<soma> ::= MAIS_TOK | MENOS_TOK;
<numero> ::= INTEIRO | REAL;
<tipo> ::= INT_TOK | DOUBLE_TOK | CHAR_TOK | FLOAT_TOK | VOID_TOK;
<logic> ::= L_AND_TOK | L_OR_TOK | L_SHL_TOK | L_SHR_TOK;
<comm> ::= COMENTARIO;
<string> ::= STRING;
<id> ::= ID;
<abre_p> ::= A_PAR_TOK;
<fecha_p> ::= F_PAR_TOK;
<abre_c> ::= A_CHA_TOK;
<fecha_c> ::= F_CHA_TOK;
<pvirg> ::= PVIRG_TOK;
<main> ::= MAIN_TOK;
<cond_if> ::= IF_TOK;
<cond_else> ::= ELSE_TOK;
<imp> ::= IMPORT_TOK;
<bib> ::= BIB;
<ret> ::= RETURN_TOK;
<def> ::= DEFINE_TOK;
<virgula> ::= VIRG_TOK;
<scan> ::= SCAN_TOK;
<print> ::= PRINT_TOK;
```

Figura 8: Gramática BNF

6 Arvore Sintática e Funções

A analise sintática é feita pelo parser e a partir dela foi feito a construção da arvore sintática com o auxilio de funções em C para ser feita a impressão da mesma em outro arquivo.txt.

```
1   Nivel 00-> Inicio
2   Nivel 01-> Definicoes
3   Nivel 02-> Sequencia
4   Nivel 03-> Declaracao   Pvirg
5   Nivel 04-> Atribuicao   ;
6   Nivel 05-> Tipo         Id          Sinal       Exp
7   Nivel 06-> int          x           =           Exp_S
8   Nivel 07-> Termo
9   Nivel 08-> Fator
10  Nivel 09-> Num
11  Nivel 10-> 10
```

Figura 9: Arvore Sintática

Foi criado duas estruturas para representar árvores e pilhas na implementações. A estrutura Arv representa um nó em uma árvore, contendo um token (representando um elemento da árvore), o número de filhos desse nó, um vetor de ponteiros para os filhos e o nível do nó na árvore. Já a estrutura Pilha é utilizada para implementar uma pilha de nós da árvore, onde cada elemento da pilha é um ponteiro para um nó da árvore (Arv) e um ponteiro para o próximo elemento da pilha.

6.1 Funções da Pilha

As duas funções estão relacionadas à manipulação de uma estrutura de dados do tipo pilha. A função push recebe uma pilha p e um elemento a para ser inserido no topo da pilha. Ela cria um novo nó no com o elemento a e aponta o próximo nó para a pilha p, retornando o novo topo da pilha. Já a função pop remove o elemento do topo da pilha p e retorna o novo topo da pilha. Ela simplesmente move o ponteiro para o próximo nó e libera a memória do nó removido, garantindo que a pilha continue consistente.

```c
// Descrição: Insere um novo elemento no topo da pilha.
// Pré-Condição: p deve ser uma pilha válida
// Pós-Condição: o elemento a é inserido no topo da pilha
Pilha *push(Pilha* p, Arv *a){
    Pilha *no = (Pilha*)malloc(sizeof(Pilha));
    no->token = a;
    no->prox = p;
    return no;
}

// Descrição: Remove o elemento do topo da pilha.
// Pré-Condição: p deve ser uma pilha válida
// Pós-Condição: o elemento do topo da pilha é removido
Pilha *pop(Pilha* p){
    Pilha *no = p->prox;
    free(p);
    return no;
}
```

Figura 10: Função Push e Pop

6.2 Função criarNo

Essa função criaNo é responsável por criar um novo nó de uma árvore genérica. Ela recebe como parâmetros uma string token, que representa o valor do nó, e um inteiro numFilhos, que indica o número de filhos que o nó terá. O primeiro passo da função é alocar memória para o novo nó no. Em seguida, ela duplica a string token usando a função strdup e atribui o resultado ao campo token do nó. O campo numFilhos do nó é então atribuído com o valor recebido como parâmetro.

Depois, a função verifica se o número de filhos é zero. Se for, ela define o campo filhos como NULL, indicando que o nó não possui filhos. Caso contrário, ela aloca memória para um array de ponteiros para nós (Arv** filhos) com o tamanho especificado por numFilhos. Por fim, a função retorna o novo nó criado, que pode ser utilizado para construir uma árvore genérica.

```c
// Descrição: Cria um novo nó para a árvore.
// Pré-Condição: nenhum
// Pós-Condição: um novo nó é criado com o token e número de filhos especificados
Arv *criaNo(char *token, int numFilhos){
    Arv *no = (Arv*)malloc(sizeof(Arv));
    no->token = strdup(token);
    no->numFilhos = numFilhos;
    if(numFilhos == 0){ no->filhos = NULL; }else{
        no->filhos = (Arv**) malloc(numFilhos*sizeof(Arv*));
    }
    return no;
}
```

Figura 11: Função criaNo

6.3 Funções de inserção

A função insere recebe uma pilha (Pilha *p), um token (representando um elemento da árvore) e o número de filhos que o nó correspondente na árvore terá. Ela cria um novo nó (Arv *no) com o token e o número de filhos recebidos. Em seguida, um loop é executado para atribuir os filhos do nó, que são retirados da pilha p. Por fim, o novo nó é inserido na pilha p e ela é retornada atualizada.

```c
// Descrição: Insere um nó na pilha.
// Pré-Condição: p deve ser uma pilha válida
// Pós-Condição: o nó é inserido na pilha
Pilha *insere(Pilha *p, char *token, int numFilhos){
    Arv *no = criaNo(token, numFilhos);
    for(int i = 0; i < numFilhos; i++){
        (no->filhos)[i] = p->token;
        p = pop(p);
    }
    p = push(p, no);
    return p;
}
```

Figura 12: Função insere

A função insereInt recebe uma pilha (Pilha *p), um número inteiro (int n) e o número de filhos (int numFilhos) que o nó correspondente na árvore terá. Ela converte o número inteiro para uma string usando sprintf e, em seguida, chama a função insere para inserir essa string na pilha, juntamente com o número de filhos. Por fim, retorna a pilha atualizada. A função insereDouble é semelhante, mas recebe um número de ponto flutuante (double n) em vez de um número inteiro. Ela converte o número de ponto flutuante para uma string usando sprintf e, em seguida, chama a função insere para inserir essa string na pilha, juntamente com o número de filhos. Por fim, retorna a pilha atualizada.

```c
// Descrição: Insere um número inteiro na pilha.
// Pré-Condição: p deve ser uma pilha válida
// Pós-Condição: o número inteiro é inserido na pilha
Pilha *insereInt(Pilha *p, int n, int numFilhos){
    char token[100];
    sprintf(token, "%d", n);
    p = insere(p, token, numFilhos);
    return p;
}

// Descrição: Insere um número real na pilha.
// Pré-Condição: p deve ser uma pilha válida
// Pós-Condição: o número real é inserido na pilha
Pilha *insereDouble(Pilha *p, double n, int numFilhos){
    char token[100];
    sprintf(token, "%lf", n);
    p = insere(p, token, numFilhos);
    return p;
}
```

Figura 13: Função Insere Int e Double

A função insereLista insere um nó de árvore Arv* a em uma lista encadeada de nós de árvore Pilha* l, mantendo a lista ordenada em ordem decrescente com base no nível dos nós da árvore. A função verifica se a lista está vazia ou se o nível do nó a ser inserido é maior ou igual ao nível do primeiro nó da lista. Se uma dessas condições for verdadeira, um novo nó é criado com o nó a ser inserido e inserido no início da lista. Caso contrário, a função é chamada recursivamente para inserir o nó na lista restante.

```c
// Descrição: Insere um nó em uma lista de acordo com o nível.
// Pré-Condição: nenhum
// Pós-Condição: o nó a é inserido na lista de forma ordenada por nível
Pilha *insereLista(Pilha* l, Arv* a){
    if(l==NULL || l->token->nivel>=a->nivel){
        Pilha* aux = (Pilha*) malloc(sizeof(Pilha));
        aux->token = a;
        aux->prox = l;
        return aux;
    }
    l->prox = insereLista(l->prox, a);
    return l;
}
```

Figura 14: Função InsereLista

6.4 Função de Ordenação

A função ordenaPorNivel recebe uma lista encadeada de nós de árvore Pilha** l, um nó de árvore Arv* a e um nível nivel. Ela atribui o nível nivel ao nó a e o insere na lista l mantendo a ordem decrescente com base no nível. Em seguida, para cada filho de a, a função é chamada recursivamente para ordenar os filhos de a por nível, incrementando o nível em 1 a cada chamada recursiva.

```
// Descrição: Ordena os nós por nível na lista.
// Pré-Condição: l deve ser uma lista válida, a deve ser um nó válido, nivel deve ser o nível inicial
// Pós-Condição: os nós são ordenados por nível na lista
void ordenaPorNivel(Pilha** l, Arv *a, int nivel){
    a->nivel = nivel;
    *l = insereLista(*l, a);
    for(int i = 0; i < a->numFilhos; i++){
        ordenaPorNivel(l, a->filhos[i], nivel+1);
    }
}
```

Figura 15: Função de ordenação

6.5 Funções de Impressão

A função imprimirPilha é responsável por imprimir o conteúdo da pilha de forma recursiva. Ela recebe como parâmetro um ponteiro para o topo da pilha Pilha *p. A função verifica se a pilha não está vazia (ou seja, p não é NULL). Se não estiver vazia, ela imprime o token do nó atual da pilha (p->token->token) e então chama recursivamente a função imprimirPilha passando o próximo nó da pilha (p->prox). Esse processo se repete até que todos os nós da pilha tenham sido impressos. Ao final da execução, todos os tokens dos nós da pilha terão sido impressos em ordem, do topo ao fim da pilha.

```
// Descrição: Imprime os elementos da pilha.
// Pré-Condição: p deve ser uma pilha válida
// Pós-Condição: os elementos da pilha são impressos na tela
void imprimirPilha(Pilha *p){
    if(p!=NULL){
        printf("%s\n", p->token->token);
        imprimirPilha(p->prox);
    }
}
```

Figura 16: Função Imprimir Pilha

A função imprimirPorNivel recebe como parâmetro o nome de um arquivo (nome_arq) e uma árvore (Arv* a) e imprime os tokens da árvore em ordem de nível no arquivo especificado. Primeiramente, ela abre o arquivo em modo de escrita (”w”), verificando se a abertura foi bem-sucedida. Em seguida, ela inicializa uma pilha l como NULL e uma variável nivel como 0.

A função ordenaPorNivel é chamada para ordenar os nós da árvore por nível, preenchendo a pilha l com os nós ordenados. Em seguida, um loop while é usado para percorrer a pilha l.

Dentro do loop, a função verifica se o nível do nó atual (l->token->nivel) é maior que o nível atual (nivel). Se for, ela imprime o número do novo nível e atualiza a variável nivel. Em seguida, o token do nó atual é impresso no arquivo, seguido por um caractere de tabulação.

Após imprimir o token, a função libera a memória alocada para o nó atual (l) e avança para o próximo nó da pilha. Esse processo se repete até que todos os nós da pilha tenham sido impressos e a pilha esteja vazia. Ao final, o arquivo contém os tokens da árvore impressos em ordem de nível, com cada nível separado por uma quebra de linha.

```c
// Descrição: Imprime os nós da árvore por nível em um arquivo.
// Pré-Condição: a deve ser uma árvore válida
// Pós-Condição: os nós da árvore são impressos por nível no arquivo
void imprimirPorNivel(char* nome_arq, Arv* a){
    FILE* arq = fopen(nome_arq, "w");
    if(arq==NULL) return;
    Pilha* l = NULL, *aux;
    int nivel = 0;
    ordenaPorNivel(&l, a, 1);
    aux = l;
    while(l!=NULL){
        if(l->token->nivel>nivel){
            if(nivel)fprintf(arq, "\n");
            fprintf(arq, "Nivel %02d-> ", nivel);
            nivel=l->token->nivel;
        }
        aux = l->prox;
        fprintf(arq, "%-10s\t", l->token->token);
        free(l);
        l = aux;
    }
    fclose(arq);
}
```

Figura 17: Função para criar e imprimir no arquivo

7 Processo de montagem

A primeira etapa foi realizada a especificação da linguagem a ser compilada, ou seja, um analisador léxico (foi desenvolvido para converter o código-fonte em uma sequência de tokens. Cada token representa um elemento da linguagem, como identificadores, números e palavras-chave. O lexico também remove comentários e espaços em branco desnecessários.

Em seguida, um analisador sintático (parser) é implementado para analisar a estrutura gramatical do código-fonte. O parser verifica se o código-fonte está de acordo com as regras sintáticas da linguagem e constrói uma árvore sintática.

Foram utilizadas as ferramentas Bison e FLEX, juntamente com a IDE Visual Studio Code, para o desenvolvimento do código, o qual foi implementado na linguagem C.

8 Referências

BORDINI, Camile. Compiladores Análise Léxica. UNIOESTE – Foz Ciência da Computação. 2023. [Notas de aula]

BORDINI, Camile. Compiladores Introdução. UNIOESTE – Foz Ciência da Computação. 2023. [Notas de aula]

BORDINI, Camile. Exemplo de Gramatica Simples. UNIOESTE – Foz Ciência da Computação. 2024. [Notas de aula]