

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	2
2	Descrição da Linguagem	2
3	Tratamento de Erros	4
4	Funções Léxicas	4
4.1	Função Hashing e criaHash	5
4.2	Função InserirHash e inserir	5
4.3	Função limpa	6
4.4	Função busca	7
4.5	Função imprimir e imprimiHash	8
5	Parte Sintática	8
6	Arvore Sintática e Funções	9
6.1	Funções da Pilha	10
6.2	Função criarNo	10
6.3	Funções de inserção	11
6.4	Função de Ordenação	13
6.5	Funções de Impressão	13
7	Análise Semântica e Funções	14
8	Alterações Realizadas	15
9	Funções	16
9.1	Função atribuiValorId	16
9.2	Função atribuiSomaId	17
9.3	Função setValorId	18
9.4	Função incDec	19
9.5	Função setTipoId	20
9.6	Função setCategoriaId	20
9.7	Função inicializaId	21
9.8	Função inicializaNum	21
9.9	Função soma	22
9.10	Função mult	24
9.11	Função comparacao	25
9.12	Função logica	27
10	Processo de montagem	28
11	Referências	28

Classes de Token	Expressão Regular
Dígito – 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.

```
// 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£¢¬°“";
    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: Função 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 nas 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 “verifica-Erro”.

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.

```
// 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 *criaHash(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.

```
// 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 Função 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.

```

// 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 memória

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).

```

// 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.

```
// 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 <senal_atr> <exp> | <tipo> id <senal_atr> <exp> | id <senal_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_simple> <comp> <exp_simple> | <exp_simple> | <exp_simple> <logic> <exp_simple>;
<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_simple> ::= <exp_simple> <soma> <termo> | <termo>;
<termo> ::= <termo> <mult> <termo> | <fator>;
<fator> ::= <abre_p> <exp> <fecha_p> | <numero> | id;
<exp> ::= FOR_TOK | WHILE_TOK;
<senal_atr> ::= ATR_TOK | INC_TOK | DEC_TOK | ATR_SM_TOK | ATR_DC_TOK | ATR_MT_TOK | ATR_OV_TOK | ATR_MD_TOK;
<comp> ::= MEN_TOK | IG_TOK | MAI_TOK | DIF_TOK | MAI_IG_TOK | MEN_IG_TOK;
<mult> ::= MULTI_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 análise sintática é feita pelo parser e a partir dela foi feito a construção da árvore sintática com o auxílio 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 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.

```
// 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 `criarNo` 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.

```

// 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.

```

// 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.

```

// 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.

```

// 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.

```
// 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, "Nível %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 Análise Semântica e Funções

A análise semântica de um código é a etapa de compilação ou interpretação em que o significado do código fonte é verificado em relação às regras da linguagem de programação. Ela vai além da análise sintática, que verifica a estrutura gramatical do código, e se concentra em garantir que o código tenha significado lógico e consistência de uso de variáveis, tipos de dados, operações e outras construções da linguagem. Durante a análise semântica, o compilador ou interpretador verifica se as variáveis são usadas corretamente de acordo com seu escopo e tipo, se as operações são aplicadas a tipos compatíveis, se as funções são chamadas corretamente, entre outros aspectos.

Nesse contexto, o software desenvolvido não apenas exibe a Tabela de Símbolos e de Tokens, mas também apresenta na tela uma tabela contendo informações sobre Nome, Categoria, Tipo e Valor.

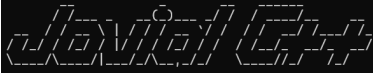


TABELA DE TOKENS

Token	Nome	Tipo	Categoria	Valor
CONDICIONAL	if	0	0	0000000000000000
ABERTURA PARENTeses	(0	0	0000000000000000
FECHAMENTO PARENTese)	0	0	0000000000000000
OPERADOR DIVISAO	/	0	0	0000000000000000
PONTO E VIRGULA	;	0	0	0000000000000000
OPERADOR ATRIBUICAO	=	0	0	0000000000000000
COMANDO DE RETORNO	return	0	0	0000000000000000
AND LOGICO	&&	0	0	0000000000000000
OPERADOR INCREMENTO	++	0	0	0000000000000000
OPERADOR MENOR IGUAL	<=	0	0	0000000000000000
ABERTURA CHAVES	{	0	0	0000000000000000
FECHAMENTO CHAVES	}	0	0	0000000000000000
REPETICAO	for	0	0	0000000000000000
TIPO DE DADO	int	0	0	0000000000000000

Figura 18: Tabelas

8 Alterações Realizadas

Para fazer a parte semântica foram feitas algumas alterações dentro do código, como também a estrutura da Struct Item e as funções de inserção.

```
typedef struct no {
    char nome[100];
    char token [100];
    int tipo; // 0 = eh um token, 1 = char, 2 = int, 3 = float, 4 = double, 5 = void;
    int categoria; // 0 = eh um token, 1 = variavel, 2 = procedimento, 3 = numero;
    void *valor; // null = eh um token, ponteiro generico para valor (pode ser qualquer um dos tipos acima)
    int linha;
    struct no *prox;
}Item;

typedef struct no {
    char nome[100];
    char token [100];
    int linha;
    struct no *prox;
}Item;
```

Figura 19: Nova Estrutura para Item

A alteração nas regras foi de declaração de variável (decVar) e a regra do Identificador e foi criada uma regra para função. As duas funções são quase idênticas, exceto pela função setCategoriaId(), onde a função setCategoriaId define o valor 2 (procedimento), enquanto na função decVar o valor atribuído é 1 (variável).

<i>dec_var:</i>	<i>id:</i>	<i>func:</i>
<i>tipo id</i>	<i>ID</i>	<i>tipo id</i>
<i>;</i>	<i>;</i>	<i>;</i>

Figura 20: Regras Alteradas

A modificação em ambas as funções de inserção foi os parâmetros e o código, mas a lógica permaneceu a mesma. Como visto na figura a seguir, a entrada ficou com um ponteiro para o primeiro item da lista (Item *item), nome do novo item (char *nome), token do novo item (char *token), valor do novo item (void* valor), tipo do novo item (int tipo), categoria do novo item (int categoria).


```

// 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, void* valor, int tipo, int categoria){
    int key = hashing(h, nome);
    h->vet[key] = inserir(h->vet[key], nome, token, valor, tipo, categoria);
}

// 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, void* valor, int tipo, int categoria){
    if(busca(item, nome)==NULL){
        Item *aux = (Item *)malloc(sizeof(Item));
        strcpy(aux->nome, nome);
        strcpy(aux->token, token);
        aux->tipo = tipo;
        aux->categoria = categoria;
        aux->valor = valor;
        aux->prox = item;
        return aux;
    }
    return item;
}

```

Figura 21: Novas funções de inserção

9 Funções

No código-fonte as funções para realizar a análise semântica estão declaradas e documentadas na biblioteca `semantica.h`, assim como a estrutura utilizada. No arquivo “`hash.c`”, as funções estão programadas. A implementação está no `semantico.c`.

9.1 Função `atribuiValorId`

A função `atribuiValorId` realiza a atribuição de valores entre dois itens, `item1` e `item2`, com base no tipo de dado de `item1`. A função verifica o tipo de `item1` utilizando um comando `switch` e, dependendo do tipo, aloca a memória necessária e copia o valor de `item2` para `item1`. Caso o tipo de `item1` seja 0, a função sinaliza um erro de não declaração e encerra a execução. Para os tipos `char`, `int`, `float` e `double`, a função aloca memória suficiente para o tipo específico e copia o valor correspondente de `item2` para `item1`. Dessa forma, a função assegura a correta atribuição de valores entre itens, respeitando os tipos de dados envolvidos e prevenindo erros de segmentação através da alocação dinâmica de memória.

```

// Descrição: Atribui o valor de item2 para item1 conforme o tipo de item1.
// Entrada: Dois ponteiros para estruturas do tipo Item, item1 e item2.
// Retorno: Nenhum.
// Pré-Condição: item1 e item2 não devem ser NULL e devem ter um tipo válido.
// Pós-Condição: O valor de item1 é alocado e atribuído com base no valor de item2.
void atribuiValorId(Item *item1, Item *item2) {
    // Verifica o tipo de item1 para realizar a atribuição correspondente
    switch (item1->tipo) {
        case 0:
            // Caso o tipo seja 0, exibe um erro e finaliza a execução
            printf("Error: '%s' nao declarada na linha %d\n", item1->nome, linha);
            exit(EXIT_FAILURE);
            break;
        case 1:
            // Caso o tipo seja char, aloca memória e copia o valor de item2 para item1
            item1->valor = malloc(sizeof(char));
            *(char*)item1->valor = *(char*)item2->valor;
            break;
        case 2:
            // Caso o tipo seja int, aloca memória e copia o valor de item2 para item1
            item1->valor = malloc(sizeof(int));
            *(int*)item1->valor = *(int*)item2->valor;
            break;
        case 3:
            // Caso o tipo seja float, aloca memória e copia o valor de item2 para item1
            item1->valor = malloc(sizeof(float));
            *(float*)item1->valor = *(float*)item2->valor;
            break;
        case 4:
            // Caso o tipo seja double, aloca memória e copia o valor de item2 para item1
            item1->valor = malloc(sizeof(double));
            *(double*)item1->valor = *(double*)item2->valor;
            break;
    }
}

```

Figura 22: Função atribuiValorId

9.2 Função atribuiSomaId

A função atribuiSomaId é responsável por atribuir um valor a item1 conforme seu tipo, convertendo o valor a partir de uma string valor. Esta função recebe dois parâmetros: um ponteiro para uma estrutura do tipo Item, denominada item1, e uma string valor. A função não retorna nenhum valor.

A função começa verificando o tipo de item1 utilizando um comando switch e, para cada caso, realiza a alocação de memória necessária e a conversão do valor da string valor para o tipo correspondente. Especificamente, a função comporta-se da seguinte forma para cada tipo:

- Caso o tipo de item1 seja 0, a função exibe uma mensagem de erro indicando que o item não foi declarado e encerra a execução do programa utilizando exit(EXIT_FAILURE).
- Caso o tipo de item1 seja char, a função aloca memória para um char e atribui o valor da string valor a item1.
- Caso o tipo de item1 seja int, a função aloca memória para um int e converte a string valor para um int utilizando atoi, atribuindo este valor a item1.
- Caso o tipo de item1 seja float, a função aloca memória para um float e converte a string valor para um float utilizando atof, atribuindo este valor a item1.
- Caso o tipo de item1 seja double, a função aloca memória para um double e converte a string valor para um double utilizando atof, atribuindo este valor a item1.

As pré-condições para a função incluem que item1 não deve ser NULL e deve ter um tipo válido, e que valor deve ser uma string válida que possa ser convertida para o tipo de item1.

A pós-condição é que o valor de `item1` será alocado e atribuído com base no valor convertido da string `valor`.

```
void atribuiSomaId(item *item1, char *valor) {
    // Verifica o tipo de item1 para realizar a atribuição correspondente
    switch (item1->tipo) {
        case 0:
            // Caso o tipo seja 0, exibe um erro e finaliza a execução
            printf("Error: '%s' nao declarada na linha %d\n", item1->nome, linha);
            exit(EXIT_FAILURE);
            break;
        case 1:
            // Caso o tipo seja char, aloca memória e atribui o valor da string 'valor' a item1
            item1->valor = malloc(sizeof(char));
            *(char*)item1->valor = *valor;
            break;
        case 2:
            // Caso o tipo seja int, aloca memória e converte a string 'valor' para int, atribuindo a item1
            item1->valor = malloc(sizeof(int));
            *(int*)item1->valor = atoi(valor);
            break;
        case 3:
            // Caso o tipo seja float, aloca memória e converte a string 'valor' para float, atribuindo a item1
            item1->valor = malloc(sizeof(float));
            *(float*)item1->valor = atof(valor);
            break;
        case 4:
            // Caso o tipo seja double, aloca memória e converte a string 'valor' para double, atribuindo a item1
            item1->valor = malloc(sizeof(double));
            *(double*)item1->valor = atof(valor);
            break;
    }
}
```

Figura 23: Função `atribuiSomaId`

9.3 Função `setValorId`

A função `setValorId` é utilizada para atribuir valores entre dois itens identificados por `id1` e `id2` em uma tabela hash. A função segue os seguintes passos:

- Calcula a chave hash para `id1` utilizando a função `hashing`, e busca o item correspondente na tabela hash, armazenando-o em `item1`.
- Calcula a chave hash para `id2` utilizando a função `hashing`, e busca o item correspondente na tabela hash, armazenando-o em `item2`.
- Verifica se `item2` não é nulo, o que indica que `id2` existe na tabela hash.
 - Se `item2` não for nulo:
 - * Verifica se os tipos dos itens `item1` e `item2` são iguais.
 - Se os tipos forem iguais e o sinal for de atribuição ("="), chama a função `atribuiValorId` para atribuir o valor de `item2` a `item1`.
 - Se os tipos forem diferentes, exibe uma mensagem de erro indicando a incompatibilidade de tipos e finaliza a execução com `exit(EXIT_FAILURE)`.
 - Se `item2` for nulo:
 - * Verifica se o sinal é de atribuição ("=").
 - Se o sinal for de atribuição, chama a função `atribuiSomaId` para atribuir o valor da string `id2` a `item1`.

```

void setValorId(Hash *h, char *id1, char *sinal, char *id2) {
    // Calcula a chave hash para id1 e busca o item correspondente na tabela hash
    int key1 = hashing(h, id1);
    Item *item1 = busca(h->vet[key1], id1);

    // Calcula a chave hash para id2 e busca o item correspondente na tabela hash
    int key2 = hashing(h, id2);
    Item *item2 = busca(h->vet[key2], id2);

    // Verifica se item2 não é nulo (existe na tabela hash)
    if (item2 != NULL) {
        // Verifica se os tipos dos itens são iguais
        if (item1->tipo == item2->tipo) {
            // Se o sinal é de atribuição '=', chama a função para atribuir o valor de item2 a item1
            if (strcmp(sinal, "=") == 0) {
                atribuiValorId(item1, item2);
            }
        } else {
            // Se os tipos são diferentes, exibe uma mensagem de erro e finaliza a execução
            printf("Error: tipo de '%s' incompatível com '%s' na linha %d\n", item1->nome, item2->nome, linha);
            exit(EXIT_FAILURE);
        }
    } else {
        // Se item2 não existe na tabela hash e o sinal é de atribuição '=', chama a função para atribuir o valor da string id2 a item1
        if (strcmp(sinal, "=") == 0) {
            atribuiSomaid(item1, id2);
        }
    }
}

```

Figura 24: Função setValorId

9.4 Função incDec

A função `incDec` é utilizada para incrementar ou decrementar o valor de um item na tabela hash, identificado por `id`, de acordo com o sinal passado em `sinal`. Os passos da função são os seguintes:

- Calcula a chave hash para `id` utilizando a função `hashing`, e busca o item correspondente na tabela hash, armazenando-o em `item`.
- Verifica se `sinal` é igual a `++` (incremento).
 - Se for igual a `++`, incrementa o valor do item em 1.
- Caso contrário, verifica se `sinal` é igual a `--` (decremento).
 - Se for igual a `--`, decrementa o valor do item em 1.

```

void incDec(Hash* h, char *id, char *sinal){
    // Calcula a chave hash para id e busca o item correspondente na tabela hash
    int key = hashing(h, id);
    Item *item = busca(h->vet[key], id);

    // Verifica se o sinal é de incremento '++'
    if(strcmp(sinal, "++") == 0){
        // Incrementa o valor do item em 1
        *(int*)item->valor += 1;
    } else if(strcmp(sinal, "--") == 0){
        // Verifica se o sinal é de decremento '--'
        // Decrementa o valor do item em 1
        *(int*)item->valor -= 1;
    }
}

```

Figura 25: Função incDec

Essa função permite a manipulação dos valores dos itens na tabela hash, facilitando operações de incremento e decremento de forma eficiente e segura. Ela assegura que a atribuição de valores entre itens seja realizada corretamente, respeitando os tipos de dados e prevenindo erros de execução através de verificações e tratamento de erros apropriados.

9.5 Função setTipoId

A função `setTipoId` é responsável por atribuir um tipo específico a um item na tabela hash identificado por `id`. Os passos da função são os seguintes:

- Calcula a chave hash para `id` utilizando a função `hashing`, e busca o item correspondente na tabela hash, armazenando-o em `item`.
- Verifica se `item` não é nulo (ou seja, se o item existe na tabela hash).
 - Se `item` não for nulo, verifica se o tipo atual do item é 0.
 - * Se o tipo atual for 0, atualiza o tipo do item para o valor especificado em `tipo`.
 - * Caso contrário, exibe uma mensagem de erro informando que há tipos conflitantes para `id` na linha atual do programa e encerra a execução.

Essa função é útil para garantir a consistência dos tipos dos itens na tabela hash, evitando conflitos e erros de interpretação durante a execução do programa.

```
void setTipoId(Hash *h, int tipo, char *id){
    // Calcula a chave hash para id e busca o item correspondente na tabela hash
    int key = hashing(h, id);
    Item *item = busca(h->vet[key], id);

    // Verifica se o item existe
    if(item != NULL){
        // Verifica se o tipo atual do item é 0
        if (item->tipo == 0){
            // Atualiza o tipo do item
            item->tipo = tipo;
        }else{
            // Exibe uma mensagem de erro e encerra o programa se os tipos forem conflitantes
            printf("Error: tipos conflitantes para '%s' na linha %d\n", id, linha);
            exit(EXIT_FAILURE);
        }
    }
}
```

Figura 26: Função `setTipoId`

9.6 Função setCategoriaId

A função `setCategoriaId` tem como objetivo atribuir uma categoria específica a um item na tabela hash identificado por `id`. Os passos da função são os seguintes:

- Calcula a chave hash para `id` utilizando a função `hashing`, e busca o item correspondente na tabela hash, armazenando-o em `item`.
- Atualiza a categoria do item para o valor especificado em `categoria`.

Essa função é útil para classificar os itens da tabela hash em categorias específicas, o que pode ser relevante em determinadas aplicações para organização e identificação dos itens.

```
void setCategoriaId(Hash *h, int categoria, char *id){
    // Calcula a chave hash para id e busca o item correspondente na tabela hash
    int key = hashing(h, id);
    Item *item = busca(h->vet[key], id);
    // Atualiza a categoria do item
    item->categoria = categoria;
}
```

Figura 27: Função setCategoriaId

9.7 Função inicializaId

A função inicializaId tem como objetivo inicializar o valor de um item na tabela hash identificado por id. Os passos da função são os seguintes:

- Calcula a chave hash para id utilizando a função hashing, e busca o item correspondente na tabela hash, armazenando-o em item.
- Utiliza um comando switch para verificar o tipo do item e, com base nesse tipo, inicializa o valor do item:
 - Se o tipo for 2 (int), a função aloca memória para um inteiro e inicializa o valor com zero.
 - Se o tipo for 4 (double), a função aloca memória para um double e inicializa o valor com zero.

Essa função é útil para garantir que o valor de um item na tabela hash seja inicializado corretamente, evitando valores indesejados ou não inicializados em operações subsequentes.

```
void inicializaId(Hash *h, char* id){
    // Calcula a chave hash para id e busca o item correspondente na tabela hash
    int key = hashing(h, id);
    Item *item = busca(h->vet[key], id);

    // Inicializa o valor do item baseado no seu tipo
    switch (item->tipo){
        case 2:
            // Aloca memória para um inteiro e inicializa com zero
            item->valor = malloc(sizeof(int));
            *(int*)item->valor = 0;
            break;
        case 4:
            // Aloca memória para um double e inicializa com zero
            item->valor = malloc(sizeof(double));
            *(double*)item->valor = 0.0;
            break;
    }
}
```

Figura 28: Função inicializaId

9.8 Função inicializaNum

A função inicializaNum tem como objetivo inicializar o valor de um item na tabela hash identificado por id, convertendo o valor da string id para o tipo apropriado (inteiro ou double) e atribuindo-o ao item. Os passos da função são os seguintes:

- Calcula a chave hash para id utilizando a função `hashing`, e busca o item correspondente na tabela hash, armazenando-o em `item`.
- Utiliza um comando `switch` para verificar o tipo do item e, com base nesse tipo, inicializa o valor do item:
 - Se o tipo for 2 (int), a função aloca memória para um inteiro e inicializa o valor com o resultado da conversão da string `id` para inteiro usando a função `atoi`.
 - Se o tipo for 4 (double), a função aloca memória para um double e inicializa o valor com o resultado da conversão da string `id` para double usando a função `atof`.

```
void inicializaNum(Hash *h, char* id){
    // Calcula a chave hash para id e busca o item correspondente na tabela hash
    int key = hashing(h, id);
    Item *item = busca(h->vet[key], id);

    // Inicializa o valor do item baseado no seu tipo
    switch (item->tipo){
        case 2:
            // Aloca memória para um inteiro e inicializa com o valor convertido da string id
            item->valor = malloc(sizeof(int));
            *(int*)item->valor = atoi(id);
            break;
        case 4:
            // Aloca memória para um double e inicializa com o valor convertido da string id
            item->valor = malloc(sizeof(double));
            *(double*)item->valor = atof(id);
            break;
    }
}
```

Figura 29: Função `atribuiValorId`

Essa função é útil para garantir que o valor de um item na tabela hash seja inicializado corretamente com um valor numérico, convertendo-o da string `id` para o tipo apropriado.

9.9 Função soma

A função `soma` realiza operações de soma ou subtração entre dois valores, identificados pelos IDs `id1` e `id2`, com base no sinal especificado (+ para soma e - para subtração). Ela retorna o resultado da operação como uma string em `str`. Os passos da função são os seguintes:

```

char *soma(hash *h, char *id1, char *sinal, char *id2, char *str){
    // Calcula a chave hash para id1 e busca o item correspondente na tabela hash
    int key1 = hashing(h, id1);
    Item *item1 = busca(h->vet[key1], id1);

    // Calcula a chave hash para id2 e busca o item correspondente na tabela hash
    int key2 = hashing(h, id2);
    Item *item2 = busca(h->vet[key2], id2);

    int i; double d;
    // Verifica se id2 não existe na tabela hash
    if(item2 == NULL){
        switch (item1->tipo){
            case 2:
                // Realiza a soma ou subtração com base no sinal e armazena o resultado em i
                if(*sinal == '+'){
                    i = *(int*)item1->valor + atoi(id2);
                }else if(*sinal == '-'){
                    i = *(int*)item1->valor - atoi(id2);
                }
                // Converte o resultado para string e armazena em str
                sprintf(str, "%d", i);
                break;
            case 4:
                // Realiza a soma ou subtração com base no sinal e armazena o resultado em d
                if(*sinal == '+'){
                    d = *(double*)item1->valor + atof(id2);
                }else if(*sinal == '-'){
                    d = *(double*)item1->valor - atof(id2);
                }
                // Converte o resultado para string e armazena em str
                sprintf(str, "%lf", d);
                break;
        }
    }
}

```

Figura 30: Função soma parte 1

- Calcula a chave hash para id1 e busca o item correspondente na tabela hash, armazenando-o em item1.
- Calcula a chave hash para id2 e busca o item correspondente na tabela hash, armazenando-o em item2.
- Verifica se id2 não existe na tabela hash. Em caso afirmativo, realiza a operação com base no tipo de item1 e armazena o resultado em i (para inteiros) ou d (para doubles). Converte o resultado para string e armazena em str.
- Se id2 existe na tabela hash e os tipos de item1 e item2 são iguais, realiza a operação com base no tipo de item1 e armazena o resultado em i ou d, dependendo do tipo. Converte o resultado para string e armazena em str.
- Se id2 não está declarado (tipo igual a 0), exibe uma mensagem de erro e encerra o programa.

Essa função é útil para realizar operações aritméticas simples entre valores, considerando os tipos de dados e lidando adequadamente com erros de declaração de variáveis.


```

} else if (item1->tipo == item2->tipo) {
    // Realiza a soma ou subtração com base no sinal e armazena o resultado em i ou d, dependendo do tipo
    switch (item1->tipo) {
        case 2:
            if (*sinal == '+') {
                i = *(int*)item1->valor + *(int*)item2->valor;
            } else if (*sinal == '-') {
                i = *(int*)item1->valor - *(int*)item2->valor;
            }
            // Converte o resultado para string e armazena em str
            sprintf(str, "%d", i);
            break;
        case 4:
            if (*sinal == '+') {
                d = *(double*)item1->valor + *(double*)item2->valor;
            } else if (*sinal == '-') {
                d = *(double*)item1->valor - *(double*)item2->valor;
            }
            // Converte o resultado para string e armazena em str
            sprintf(str, "%lf", d);
            break;
    }
} else if (item2->tipo == 0) {
    // Exibe uma mensagem de erro e encerra o programa se id2 não estiver declarado
    printf("Error: '%s' não declarada na linha %d\n", item2->nome, linha);
    exit(EXIT_FAILURE);
}
return str;
}

```

Figura 31: Função soma parte 2

9.10 Função mult

A função `mult` realiza operações de multiplicação ou divisão entre dois valores, identificados pelos IDs `id1` e `id2`, com base no sinal especificado (`*` para multiplicação e `/` para divisão). Ela retorna o resultado da operação como uma string em `str`. Os passos da função são os seguintes:

```

char *mult(Hash *h, char *id1, char *sinal, char *id2, char *str) {
    // Verifica se id2 não existe na tabela hash
    if (item2 == NULL) {
        switch (item1->tipo) {
            case 2:
                // Realiza a multiplicação ou divisão com base no sinal e armazena o resultado em i
                if (*sinal == '*') {
                    i = *(int*)item1->valor * atoi(id2);
                } else if (*sinal == '/') {
                    // Verifica se o divisor é diferente de zero antes de realizar a divisão
                    if (*id2 != '0') {
                        i = *(int*)item1->valor / atoi(id2);
                    } else {
                        printf("Error: divisao por 0(zero) na linha %d\n", linha);
                        exit(EXIT_FAILURE);
                    }
                }
                sprintf(str, "%d", i); // Converte o resultado para string e armazena em str
                break;
            case 4:
                // Realiza a multiplicação ou divisão com base no sinal e armazena o resultado em d
                if (*sinal == '*') {
                    d = *(double*)item1->valor * atof(id2);
                } else if (*sinal == '/') {
                    // Verifica se o divisor é diferente de zero antes de realizar a divisão
                    if (atof(id2) != 0) {
                        d = *(double*)item1->valor / atof(id2);
                    } else {
                        printf("Error: divisao por 0(zero) na linha %d\n", linha);
                        exit(EXIT_FAILURE);
                    }
                }
                // Converte o resultado para string e armazena em str
                sprintf(str, "%lf", d);
                break;
        }
    }
}

```

Figura 32: Função Mult parte 1

- Calcula a chave hash para `id1` e busca o item correspondente na tabela hash, armazenando-o em `item1`.

- Calcula a chave hash para id2 e busca o item correspondente na tabela hash, armazenando-o em item2.
- Verifica se id2 não existe na tabela hash. Em caso afirmativo, realiza a operação com base no tipo de item1 e armazena o resultado em i (para inteiros) ou d (para doubles). Converte o resultado para string e armazena em str.
- Se id2 existe na tabela hash e os tipos de item1 e item2 são iguais, realiza a operação com base no tipo de item1 e armazena o resultado em i ou d, dependendo do tipo. Converte o resultado para string e armazena em str.
- Em ambos os casos, verifica se o divisor é diferente de zero antes de realizar a divisão. Se for zero, exibe uma mensagem de erro e encerra o programa.

Essa função é útil para realizar operações aritméticas simples entre valores, considerando os tipos de dados e lidando adequadamente com erros de divisão por zero.

```
char *mult(Hash *h, char *id1, char *sinal, char *id2, char *str){
    }else if(item1->tipo == item2->tipo){
        // Realiza a multiplicação ou divisão com base no sinal e armazena o resultado em i ou d, dependendo do tipo
        switch (item1->tipo){
            case 2:
                if(*sinal == '*'){
                    i = *(int*)item1->valor * *(int*)item2->valor;
                }else if(*sinal == '/'){
                    // Verifica se o divisor é diferente de zero antes de realizar a divisão
                    if(*(int*)item2->valor != 0){
                        i = *(int*)item1->valor / *(int*)item2->valor;
                    }else{
                        printf("Error: divisao por 0(zero) na linha %d\n", linha);
                        exit(EXIT_FAILURE);
                    }
                }
                sprintf(str, "%d", i); // Converte o resultado para string e armazena em str
                break;
            case 4:
                if(*sinal == '*'){
                    d = *(double*)item1->valor * *(double*)item2->valor;
                }else if(*sinal == '/'){
                    // Verifica se o divisor é diferente de zero antes de realizar a divisão
                    if(*(double*)item2->valor != 0){
                        d = *(double*)item1->valor / *(double*)item2->valor;
                    }else{
                        printf("Error: divisao por 0(zero) na linha %d\n", linha);
                        exit(EXIT_FAILURE);
                    }
                }
                sprintf(str, "%lf", d); // Converte o resultado para string e armazena em str
                break;
        }
    }
    return str;
}
```

Figura 33: Função Mult parte 2

9.11 Função comparacao

A função comparacao compara os valores de dois itens identificados pelos IDs id1 e id2, utilizando o operador de comparação especificado em sinal. Ela retorna "1" se a comparação for verdadeira e "0" caso contrário. Os passos da função são os seguintes:

```

char *comparacao(Hash *h, char *id1, char *sinal, char *id2){
    int key1 = hashing(h, id1);
    Item *item1 = busca(h->vet[key1], id1);

    int key2 = hashing(h, id2);
    Item *item2 = busca(h->vet[key2], id2);

    if(item2 != NULL){
        if(item1->tipo == item2->tipo){
            switch (item1->tipo){
                case 2:
                    if(strcmp(sinal, "<") == 0){
                        return (*(int*)item1->valor < *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, "==") == 0){
                        return (*(int*)item1->valor == *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, ">") == 0){
                        return (*(int*)item1->valor > *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, "<=") == 0){
                        return (*(int*)item1->valor <= *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, ">=") == 0){
                        return (*(int*)item1->valor >= *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, "!=") == 0){
                        return (*(int*)item1->valor != *(int*)item2->valor) ? "1" : "0";
                    }
                }
            break;
        }
    }
}

```

Figura 34: Função Comparacao parte 1

- Calcula a chave hash para id1 e busca o item correspondente na tabela hash, armazenando-o em item1.
- Calcula a chave hash para id2 e busca o item correspondente na tabela hash, armazenando-o em item2.
- Verifica se id2 não existe na tabela hash. Em caso afirmativo, a função não realiza a comparação e retorna "0".
- Verifica se os tipos de item1 e item2 são iguais. Se forem, a função utiliza um switch para comparar os valores de acordo com o tipo de dado (inteiro ou double) e o operador de comparação especificado em sinal.
- Para tipos inteiros (tipo = 2), a função realiza as comparações de menor que (<), igual a (==), maior que (>), menor ou igual a (<=), maior ou igual a (>=) e diferente de (!=).
- Para tipos double (tipo = 4), as mesmas comparações são realizadas, porém com valores do tipo double.
- Se a comparação for verdadeira, a função retorna "1". Caso contrário, retorna "0".

```

char *comparacao(Hash *h, char *id1, char *sinal, char *id2){
    case 4:
        if(strcmp(sinal, "<") == 0){
            return (*(double*)item1->valor < *(double*)item2->valor) ? "1" : "0";
        }
        if(strcmp(sinal, "==") == 0){
            return (*(double*)item1->valor == *(double*)item2->valor) ? "1" : "0";
        }
        if(strcmp(sinal, ">") == 0){
            return (*(double*)item1->valor > *(double*)item2->valor) ? "1" : "0";
        }
        if(strcmp(sinal, "<=") == 0){
            return (*(double*)item1->valor <= *(double*)item2->valor) ? "1" : "0";
        }
        if(strcmp(sinal, ">=") == 0){
            return (*(double*)item1->valor >= *(double*)item2->valor) ? "1" : "0";
        }
        if(strcmp(sinal, "!=") == 0){
            return (*(double*)item1->valor != *(double*)item2->valor) ? "1" : "0";
        }
        break;
    }
}
// Caso nenhum dos casos anteriores seja satisfeito, retorna "0"
return "0";
}

```

Figura 35: Função comapracao parte 2

Essa função é útil para realizar operações de comparação entre valores, considerando os tipos de dados envolvidos e retornando o resultado da comparação como uma string.

9.12 Função logica

A função logica realiza operações lógicas entre dois valores identificados pelos IDs id1 e id2, utilizando o operador lógico especificado em sinal. Ela retorna "1" se a operação for verdadeira e "0" caso contrário. Os passos da função são os seguintes:

- Calcula a chave hash para id1 e busca o item correspondente na tabela hash, armazenando-o em item1.
- Calcula a chave hash para id2 e busca o item correspondente na tabela hash, armazenando-o em item2.
- Verifica se id2 não existe na tabela hash. Em caso afirmativo, a função não realiza a operação lógica e retorna "0".
- Verifica se os tipos de item1 e item2 são iguais. Se forem, a função utiliza um switch para realizar a operação lógica de acordo com o tipo de dado (inteiro ou double) e o operador lógico especificado em sinal.
- Para tipos inteiros (tipo = 2), a função realiza as operações lógicas de "E" lógico (&&), "OU" lógico (||), deslocamento de bits para direita (>>) e deslocamento de bits para esquerda (<<).
- Para tipos double (tipo = 4), as operações de "E" lógico e "OU" lógico são realizadas.
- Se a operação lógica for verdadeira, a função retorna "1". Caso contrário, retorna "0".

```

char *logica(Hash *h, char *id1, char *sinal, char *id2){
    int key1 = hashing(h, id1);
    Item *item1 = busca(h->vet[key1], id1);

    int key2 = hashing(h, id2);
    Item *item2 = busca(h->vet[key2], id2);

    if(item2 != NULL){
        if(item1->tipo == item2->tipo){
            switch (item1->tipo){
                case 2:
                    if(strcmp(sinal, "&&") == 0){
                        return (*(int*)item1->valor && *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, "||") == 0){
                        return (*(int*)item1->valor || *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, ">>") == 0){
                        return (*(int*)item1->valor >> *(int*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, "<<") == 0){
                        return (*(int*)item1->valor << *(int*)item2->valor) ? "1" : "0";
                    }
                    break;
                case 4:
                    if(strcmp(sinal, "&&") == 0){
                        return (*(double*)item1->valor && *(double*)item2->valor) ? "1" : "0";
                    }
                    if(strcmp(sinal, "||") == 0){
                        return (*(double*)item1->valor || *(double*)item2->valor) ? "1" : "0";
                    }
                    break;
            }
        }
    }
    // Se nenhum dos casos anteriores for satisfeito, retorna "0"
    return "0";
}

```

Figura 36: Função logica

Essa função é útil para realizar operações lógicas entre valores, considerando os tipos de dados envolvidos e retornando o resultado da operação como uma string.

10 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.

11 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]