Compiladores - Turma A - 0225 - Noturno
Prof. Thiago Fernandes Neves

ENTREGA LAB05

Integrantes:

- Isabela Garcia
- Joao Victor Azevedo dos Santos
- Luiz Eduardo Campos Dias
- Yago Peres dos Santos

Link para o Github

1. **Atividade 1**
   a. **Integrar as funções apresentadas ao código do LAB4.**
   b. **Refazer a função: parser_build_random_type_name()**

```
struct token* parser_build_random_type_name() {// LAB5
    static int counter = 0;
    char tmp_name[32];
    snprintf(tmp_name, sizeof(tmp_name), "__anonymous_type_%d", counter++);

    // Aloca memória para o nome e copia
    char* sval = malloc(strlen(tmp_name) + 1);
    if (!sval) {
        compiler_error(current_process, "Falha ao alocar memória para nome de
tipo anônimo\n");
        return NULL;
    }
    strcpy(sval, tmp_name);

    // Cria o token
    struct token* token = calloc(1, sizeof(struct token));
    if (!token) {
        free(sval);
        compiler_error(current_process, "Falha ao alocar memória para token de
tipo anônimo\n");
        return NULL;
    }

    token->type = TOKEN_TYPE_IDENTIFIER;
    token->sval = sval;
    token->pos = current_process->pos; // Mantém a posição atual do arquivo

    return token;
}
```

2. **Atividade 2**
   a. **Criar os arquivos scope.c e symresolver.c, adicionar ao Makefile**

scope.c:

```c
#include "compiler.h"
#include "helpers/vector.h"
#include <memory.h>
#include <stdlib.h>
#include <assert.h>

struct scope* scope_alloc() {
    struct scope* scope = calloc(1, sizeof(struct scope));
    scope->entities = vector_create(sizeof(void*));
    vector_set_peek_pointer_end(scope->entities);
    vector_set_flag(scope->entities, VECTOR_FLAG_PEEK_DECREMENT);
    return scope;
}

void scope_dealloc(struct scope* scope) {
    // Nao faz nada por enquanto.
}

struct scope* scope_create_root(struct compile_process* process) {
    assert(!process->scope.root);
    assert(!process->scope.current);
    struct scope* root_scope = scope_alloc();
    process->scope.root = root_scope;
    process->scope.current = root_scope;
    return root_scope;
}

void scope_free_root(struct compile_process* process) {
    scope_dealloc(process->scope.root);
    process->scope.root = NULL;
    process->scope.current = NULL;
}

struct scope* scope_new(struct compile_process* process, int flags) {
    assert(process->scope.root);
    assert(process->scope.current);
    struct scope* new_scope = scope_alloc();
    new_scope->flags = flags;
    new_scope->parent = process->scope.current;
```

```
    process->scope.current = new_scope;
    return new_scope;
}

void scope_iteration_start(struct scope* scope) {
    vector_set_peek_pointer(scope->entities, 0);
    if (scope->entities->flags & VECTOR_FLAG_PEEK_DECREMENT)
        vector_set_peek_pointer_end(scope->entities);
}

void scope_iteration_end(struct scope* scope) {
    // Nao faz nada por enquanto.
}

void* scope_iterate_back(struct scope* scope) {
    if (vector_count(scope->entities) == 0)
        return NULL;
    return vector_peek_ptr(scope->entities);
}

void* scope_last_entity_at_scope(struct scope* scope) {
    if (vector_count(scope->entities) == 0)
        return NULL;
    return vector_back_ptr(scope->entities);
}

void* scope_last_entity_from_scope_stop_at(struct scope* scope, struct scope*
stop_scope) {
    if (scope == stop_scope)
        return NULL;
    void* last = scope_last_entity_at_scope(scope);
    if (last)
        return last;
    struct scope* parent = scope->parent;
    if (parent)
        return scope_last_entity_from_scope_stop_at(parent, stop_scope);
    return NULL;
}

void* scope_last_entity_stop_at(struct compile_process* process, struct scope*
stop_scope) {
```

```
    return scope_last_entity_from_scope_stop_at(process->scope.current,
stop_scope);
}

void* scope_last_entity(struct compile_process* process) {
    return scope_last_entity_stop_at(process, NULL);
}

void scope_push(struct compile_process* process, void* ptr, size_t elem_size)
{
    vector_push(process->scope.current->entities, &ptr);
    process->scope.current->size += elem_size;
}

void scope_finish(struct compile_process* process) {
    struct scope* new_current_scope = process->scope.current->parent;
    scope_dealloc(process->scope.current);
    process->scope.current = new_current_scope;
    if (process->scope.root && !process->scope.current)
        process->scope.root = NULL;
}

struct scope* scope_current(struct compile_process* process) {
    return process->scope.current;
}
```

symresolver.c:

```
#include "compiler.h"
#include "helpers/vector.h"

static void symresolver_push_symbol(struct compile_process* process, struct
symbol* sym) {
    vector_push(process->symbols.table, &sym);
}

void symresolver_initialize(struct compile_process* process) {
    process->symbols.tables = vector_create(sizeof(struct vector*));
}

void symresolver_new_table(struct compile_process* process) {
    // Save the current table
    vector_push(process->symbols.tables, &process->symbols.table);
```

```
    // Overwrite the active table
    process->symbols.table = vector_create(sizeof(struct symbol*));
}

void symresolver_end_table(struct compile_process* process) {
    struct vector* last_table = vector_back_ptr(process->symbols.tables);
    process->symbols.table = last_table;
    vector_pop(process->symbols.tables);
}

struct symbol* symresolver_get_symbol(struct compile_process* process, const
char* name) {
    vector_set_peek_pointer(process->symbols.table, 0);
    struct symbol* symbol = vector_peek_ptr(process->symbols.table);
    while(symbol) {
        if (S_EQ(symbol->name, name)) break;
        symbol = vector_peek_ptr(process->symbols.table);
    }
    return symbol;
}

struct symbol* symresolver_get_symbol_for_native_function(struct
compile_process* process, const char* name) {
    struct symbol* sym = symresolver_get_symbol(process, name);
    if (!sym) return NULL;
    if (sym->type != SYMBOL_TYPE_NATIVE_FUNCTION) return NULL;
    return sym;
}

struct symbol* symresolver_register_symbol(struct compile_process* process,
const char* sym_name, int type, void* data) {
    if (symresolver_get_symbol(process, sym_name)) return NULL;
    struct symbol* sym = calloc(1, sizeof(struct symbol));
    sym->name = sym_name;
    sym->type = type;
    sym->data = data;
    symresolver_push_symbol(process, sym);
    return sym;
}

struct node* symresolver_node(struct symbol* sym) {
    if (sym->type != SYMBOL_TYPE_NODE) return NULL;
```

```
    return sym->data;
}

void symresolver_build_for_variable_node(struct compile_process* process,
struct node* node) {
    compiler_error(process, "Variables not yet supported\n");
}

void symresolver_build_for_function_node(struct compile_process* process,
struct node* node) {
    compiler_error(process, "Functions are not yet supported\n");
}

void symresolver_build_for_structure_node(struct compile_process* process,
struct node* node) {
    compiler_error(process, "Structures are not yet supported\n");
}

void symresolver_build_for_union_node(struct compile_process* process, struct
node* node) {
    compiler_error(process, "Unions are not yet supported\n");
}

void symresolver_build_for_node(struct compile_process* process, struct node*
node) {
    switch(node->type) {
        case NODE_TYPE_VARIABLE:
            symresolver_build_for_variable_node(process, node);
            break;
        case NODE_TYPE_FUNCTION:
            symresolver_build_for_function_node(process, node);
            break;
        case NODE_TYPE_STRUCT:
            symresolver_build_for_structure_node(process, node);
            break;
        case NODE_TYPE_UNION:
            symresolver_build_for_union_node(process, node);
            break;
        // Descartar outros tipos.
    }
}
```

b. **Caso de declaração de múltiplas variáveis na mesma linha.**
   i. **Incluir o código na função: void parse_variable().**
   ii. **Criar um node com type == NODE_TYPE_VARIABLE_LIST.**

```
void parse_variable(struct datatype* dtype, struct token* name_token,
struct history* history) {
    // Criar lista de variáveis usando a estrutura varlist
    struct node var_list_node = {
        .type = NODE_TYPE_VARIABLE_LIST,
        .var_list.list = vector_create(sizeof(struct node*))
    };

    if (!var_list_node.var_list.list) {
        compiler_error(current_process, "Falha ao criar lista de
variáveis\n");
        return;
    }

    // Adicionar a primeira variável
    struct node* value_node = NULL;
    int array_dims[8] = {0}; // Suporta até 8 dimensões
    int array_dim_count = 0;

    // Processa colchetes para arrays
    while (token_next_is_operator("[")) {
        token_next(); // Consome o '['
        struct token* size_token = token_next();
        if (size_token->type != TOKEN_TYPE_NUMBER) {
            compiler_error(current_process, "Esperado número como
tamanho do array\n");
            break;
        }
        array_dims[array_dim_count++] = size_token->inum;
        struct token* close_bracket = token_next();
        if (!token_is_operator(close_bracket, "]")) {
            compiler_error(current_process, "Esperado ']' após
tamanho do array\n");
            break;
        }
    }

    if (token_next_is_operator("=")) {
        token_next();
```

```
        parse_expressionable_root(history);
        value_node = node_pop();
    }

    // Criar node para a primeira variável
    make_variable_node(dtype, name_token, value_node);
    struct node* var_node = node_pop();
    if (!var_node) {
        compiler_error(current_process, "Falha ao criar node para
variável\n");
        vector_free(var_list_node.var_list.list);
        return;
    }
    // Armazenar as dimensões do array no node
    if (array_dim_count > 0) {
        var_node->var.type.flags |= DATATYPE_FLAG_IS_ARRAY;
        var_node->var.type.pointer_depth = array_dim_count;
        for (int i = 0; i < array_dim_count; i++) {
            // Usar o campo size para a primeira dimensão, e o campo
datatype_secondary para as demais
            if (i == 0) {
                var_node->var.type.size = array_dims[0];
            } else {
                // Para múltiplas dimensões, pode-se criar uma cadeia
de datatypes secundários
                if (!var_node->var.type.datatype_secondary) {
                    var_node->var.type.datatype_secondary = calloc(1,
sizeof(struct datatype));
                }
                var_node->var.type.datatype_secondary->size =
array_dims[i];
            }
        }
    }
    vector_push(var_list_node.var_list.list, &var_node);

    // Verificar se há mais variáveis (separadas por vírgula)
    while (token_next_is_operator(",")) {
        token_next();
        struct token* next_name_token = token_next();
        if (next_name_token->type != TOKEN_TYPE_IDENTIFIER) {
```

```
            compiler_error(current_process, "Esperado identificador
após vírgula\n");
            break;
        }
        // Processa colchetes para arrays
        array_dim_count = 0;
        while (token_next_is_operator("[")) {
            token_next();
            struct token* size_token = token_next();
            if (size_token->type != TOKEN_TYPE_NUMBER) {
                compiler_error(current_process, "Esperado número como
tamanho do array\n");
                break;
            }
            array_dims[array_dim_count++] = size_token->inum;
            struct token* close_bracket = token_next();
            if (!token_is_operator(close_bracket, "]")) {
                compiler_error(current_process, "Esperado ']' após
tamanho do array\n");
                break;
            }
        }
        value_node = NULL;
        if (token_next_is_operator("=")) {
            token_next();
            parse_expressionable_root(history);
            value_node = node_pop();
        }
        make_variable_node(dtype, next_name_token, value_node);
        var_node = node_pop();
        if (!var_node) {
            compiler_error(current_process, "Falha ao criar node para
variável\n");
            break;
        }
        if (array_dim_count > 0) {
            var_node->var.type.flags |= DATATYPE_FLAG_IS_ARRAY;
            var_node->var.type.pointer_depth = array_dim_count;
            for (int i = 0; i < array_dim_count; i++) {
                if (i == 0) {
                    var_node->var.type.size = array_dims[0];
                } else {
```

```
                    if (!var_node->var.type.datatype_secondary) {
                        var_node->var.type.datatype_secondary =
calloc(1, sizeof(struct datatype));
                    }
                    var_node->var.type.datatype_secondary->size =
array_dims[i];
                }
            }
        }
        vector_push(var_list_node.var_list.list, &var_node);
    }

    struct token* semicolon = token_next();
    if (!token_is_symbol(semicolon, ';')) {
        compiler_error(current_process, "Esperado ';' após declaração
de variáveis\n");
        vector_free(var_list_node.var_list.list);
        return;
    }

    struct node* created_node = node_create(&var_list_node);
    if (!created_node) {
        compiler_error(current_process, "Falha ao criar node da lista
de variáveis\n");
        vector_free(var_list_node.var_list.list);
        return;
    }
    for (int i = 0; i < vector_count(var_list_node.var_list.list);
i++) {
        struct node** var_ptr =
vector_at(var_list_node.var_list.list, i);
        if (!var_ptr || !*var_ptr) continue;
        struct token temp_token = {
            .type = TOKEN_TYPE_IDENTIFIER,
            .sval = (*var_ptr)->var.name,
            .pos = (*var_ptr)->pos
        };
        make_variable_node_and_register(history, dtype, &temp_token,
(*var_ptr)->var.val);
    }
    node_push(created_node);
}
```

iii. node_create(&(struct node){.type=NODE_TYPE_VARIABLE_LIST, .var_list=var_list});

```
void make_variable_node(struct datatype* dtype, struct token*
name_token, struct node* value_node) {
    const char* name_str = NULL;
    if (name_token) name_str = name_token->sval;
    node_create(&(struct node){.type = NODE_TYPE_VARIABLE, .var.name
= name_str, .var.type = *dtype, .var.val = value_node});
}
```

iv. Lidar com o “;” final da declaração.

v. Realizar testes, “test.c”: int a, b, c, d, e; float aa, bb, cc;

```
❯ ./main test1.c
Compiladores - TURMA A - GRUPO 7

#Input file: test1.c
#Output file: (null)

TOKEN   KE: int
TOKEN   ID: a
TOKEN   OP: ,
TOKEN   ID: b
TOKEN   OP: ,
TOKEN   ID: c
TOKEN   OP: ,
TOKEN   ID: d
TOKEN   OP: ,
TOKEN   ID: e
TOKEN   SY: ;
TOKEN   NL
TOKEN   NL
TOKEN   KE: float
TOKEN   ID: aa
TOKEN   OP: ,
TOKEN   ID: bb
TOKEN   OP: ,
TOKEN   ID: cc
TOKEN   SY: ;


Arvore de nodes:
└── VARIABLE_LIST (count: 5)
```

```
│    ├── VARIABLE (name: a, type: int)
│    ├── VARIABLE (name: b, type: int)
│    ├── VARIABLE (name: c, type: int)
│    ├── VARIABLE (name: d, type: int)
│    └── VARIABLE (name: e, type: int)


└── VARIABLE_LIST (count: 3)
│    ├── VARIABLE (name: aa, type: float)
│    ├── VARIABLE (name: bb, type: float)
│    └── VARIABLE (name: cc, type: float)


Todos os arquivos foram compilados com sucesso!
```

3. **Atividade 3**

   a. **Vetores**

      i. **Teste 1: int A[50][50];**

      ii. **Teste 2: float B[100];**

```
❯ ./main test2.c
Compiladores - TURMA A - GRUPO 7

#Input file: test2.c
#Output file: (null)

TOKEN   KE: int
TOKEN   ID: A
TOKEN   OP: [
TOKEN   NU: 50   PARENTESES: (null)
TOKEN   SY: ]
TOKEN   OP: [
TOKEN   NU: 50   PARENTESES: (null)
TOKEN   SY: ]
TOKEN   SY: ;
TOKEN   NL
TOKEN   NL
TOKEN   KE: float
TOKEN   ID: B
TOKEN   OP: [
TOKEN   NU: 100          PARENTESES: (null)
TOKEN   SY: ]
```

```
TOKEN   SY: ;


Arvore de nodes:
└── VARIABLE_LIST (count: 1)
|   └── VARIABLE (name: A, type: int, ARRAY[50][50])


└── VARIABLE_LIST (count: 1)
|   └── VARIABLE (name: B, type: float, ARRAY[100])


Todos os arquivos foram compilados com sucesso!
```

b. Structs

i. Teste 1: struct ABC {int A; float B; double C;}

ii. Teste 2: struct ABC {int A, B, C, D, E;}

```
❯ ./main test3.c
Compiladores - TURMA A - GRUPO 7

#Input file: test3.c
#Output file: (null)

TOKEN   KE: struct
TOKEN   ID: ABC
TOKEN   NL
TOKEN   SY: {
TOKEN   NL
TOKEN   KE: int
TOKEN   ID: A
TOKEN   SY: ;
TOKEN   NL
TOKEN   KE: float
TOKEN   ID: B
TOKEN   SY: ;
TOKEN   NL
TOKEN   KE: double
TOKEN   ID: C
TOKEN   SY: ;
TOKEN   NL
TOKEN   SY: }
TOKEN   SY: ;
```

```
TOKEN   NL
TOKEN   NL
TOKEN   KE: struct
TOKEN   ID: ABC
TOKEN   NL
TOKEN   SY: {
TOKEN   NL
TOKEN   KE: int
TOKEN   ID: A
TOKEN   OP: ,
TOKEN   ID: B
TOKEN   OP: ,
TOKEN   ID: C
TOKEN   OP: ,
TOKEN   ID: D
TOKEN   OP: ,
TOKEN   ID: E
TOKEN   SY: ;
TOKEN   NL
TOKEN   SY: }
TOKEN   SY: ;
TOKEN   NL


Arvore de nodes:

Processando struct:
Nome da struct: ABC
Membro adicionado: int A
Membro adicionado: float B
Membro adicionado: double C
Struct criada com 3 membros
└── STRUCT (name: ABC)



Processando struct:
Nome da struct: ABC
Membro adicionado: int A
Membro adicionado: int B
Membro adicionado: int C
Membro adicionado: int D
```

```
Membro adicionado: int E
Struct criada com 5 membros
└── STRUCT (name: ABC)


Todos os arquivos foram compilados com sucesso!
```