

**ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO**  
**Departamento de Engenharia de Computação e Sistemas Digitais**

**PCS3732 - Laboratório de Processadores**



**Relatório de Projeto – Compilador TinyC**

Integrantes: Henrique Gregory Gimenez  
João Victor Baréa e Silva  
Rodrigo Sinato Fernandes

13682677  
11806805  
13679457

15 de agosto de 2025

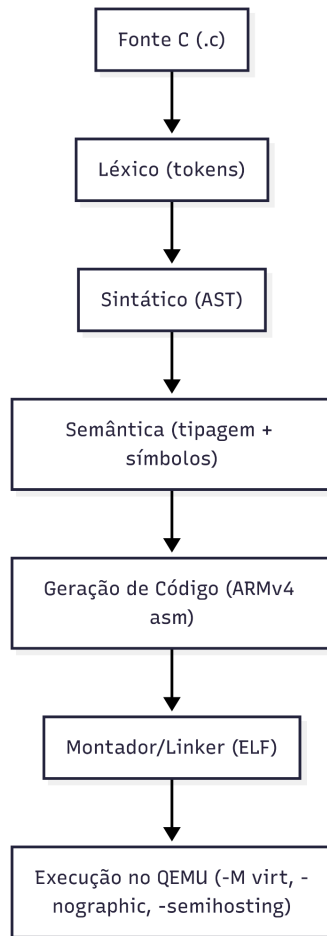
## **1. O que é o projeto?**

O objetivo do projeto é desenvolver um compilador escrito em C para compilar códigos de C---, ou seja, um C reduzido para código assembly da arquitetura ARMv4 de 32 bits, sem resolver endereços. Foi delimitado um escopo menor da linguagem C que seja possível de ser abordado até o fim do período de projetos. O compilador foi escrito em C e organizado em módulos bem definidos, visando facilitar a evolução incremental do projeto. Em suma, o propósito do projeto é tanto educativo quanto prático, proporcionando experiência com todas as etapas de compilação (análise léxica, sintática, semântica e geração de código) e com particularidades da plataforma ARM.

## **2. Fundamentação teórica**

### *2.1 Compiladores*

Um compilador é um programa de computador que traduz o código-fonte, escrito em uma linguagem de programação de alto nível (neste caso C), em um código de máquina de baixo nível (neste caso assembly ARMv4), que pode ser diretamente executado pelo processador de um computador. As principais fases do compilador incluem: análise léxica, análise sintática, análise semântica e geração de código.



*Figura - Diagrama das fases do compilador*

## 2.2 Análise léxica

Nesta etapa, o compilador lê o texto do programa fonte caractere por caractere e agrupa os caracteres em unidades léxicas válidas chamadas *tokens*. Comentários e espaços em branco são ignorados e descartados, enquanto palavras-chave, identificadores, literais e símbolos especiais são reconhecidos. Por exemplo, ao processar o código `int x = 42;`, o analisador léxico identificará tokens correspondentes a `int` (palavra-chave de tipo), `x` (identificador), `=` (operador de atribuição), `42` (literal inteiro) e `;` (terminador). Cada token geralmente é registrado com seu tipo e valor, além da posição (linha/coluna) no código fonte para fins de relato de erros. O resultado da análise léxica é, portanto, uma sequência linear de tokens que representam o código de entrada de forma simplificada.

## 2.3 Análise Sintática

Também chamada de *parsing*, essa fase recebe a lista de tokens e verifica se a sequência obedece à gramática da linguagem de programação. Ou seja, o analisador sintático valida se os tokens estão organizados em uma estrutura hierárquica correta.

Por exemplo, se após uma palavra-chave `if` ocorre uma expressão entre parênteses e um bloco de comandos. Durante a análise sintática, o compilador constrói uma **Árvore de Sintaxe Abstrata (AST)** que representa a estrutura hierárquica do programa. Cada nó da AST corresponde a um construto sintático relevante (como operadores, comandos ou declarações), organizando os tokens em uma forma em árvore que reflete a precedência e o aninhamento dos elementos. Se uma sequência de tokens viola a gramática (por exemplo, falta um `;` ao fim de uma declaração), o analisador sintático reporta um erro de sintaxe. A AST resultante torna explícita a estrutura do programa, facilitando as fases posteriores.

## 2.4 Análise semântica

Após a sintaxe do programa ser validada, a análise semântica é responsável por verificar aspectos de significado do programa que não são cobertos apenas pela gramática. Essa fase utiliza a AST construída e também mantém estruturas de símbolos para rastrear declarações de variáveis, tipos e funções. A análise semântica realiza checagens de consistência, incluindo: verificar se identificadores foram declarados antes do uso, checar incompatibilidades de tipo em operações (por exemplo, somar um número inteiro com um ponteiro seria inválido sem conversão), validar o número e tipos de argumentos em chamadas de função, garantir que expressões `return` correspondam ao tipo da função, entre outros. Por exemplo, atribuir um valor de tipo inteiro a uma variável declarada como `float` pode gerar um aviso ou erro de tipo. A análise semântica também normalmente realiza *coerção* de tipos quando necessário (por exemplo, converter implicitamente um inteiro para ponto flutuante se uma operação assim o exigir). Caso alguma verificação falhe, o compilador emite mensagens de erro semântico detalhadas, associadas às posições no código fonte, e aborta a compilação.

## 2.5 Geração de Código

Por fim, a geração de código traduz a AST anotada (isto é, os nós da árvore já enriquecidos com informações semânticas, como tipos) em instruções na linguagem de montagem alvo. Nessa etapa, o compilador aloca recursos do hardware (registradores e memória) para realizar as operações solicitadas pelo programa. Estratégias de alocação de registradores e gerenciamento de pilha são utilizadas: como a arquitetura ARM possui um conjunto limitado de registradores de propósito geral, o compilador precisa decidir quais valores temporários manter em registradores e quais *empilhar* na memória. Neste projeto, optou-se por usar até 4 registradores para avaliar expressões (em conformidade com a convenção de chamada ARM, que utiliza registradores R0–R3 para parâmetros e resultados). O empilhamento dos registradores na memória não foi implementado. A geração de código produz, para cada função, um código assembly completo com prólogo e epílogo padronizados, manipulação do *frame* de

ativação (armazenamento de variáveis locais na pilha) e instruções correspondentes às operações aritméticas, lógicas e de controle de fluxo do C.

## 2.6 Estrutura de memória de um processo

Um programa executável é organizado em segmentos de memória distintos, entre os quais destacam-se: o segmento de texto (contém o código executável), o segmento de dados (contém variáveis estáticas, globais e constantes) e a pilha (stack) onde são alocadas variáveis locais e informações de chamadas de função. Em muitos sistemas, o segmento de texto é carregado em endereços de memória baixos e é apenas leitura (não auto-modificável), o segmento de dados vem a seguir e pode crescer para cima (no sentido de endereços crescentes) conforme mais memória é alocada (por exemplo, via chamadas a `malloc` que usam a *heap*), e a pilha é alocada em endereços altos, crescendo para baixo (isto é, em direção a endereços menores) a cada chamada de função. No caso deste compilador, foi utilizado um linker script personalizado para definir esses segmentos na memória do emulador QEMU. Nesse script, por exemplo, reservamos 8 KB para a pilha no topo da memória disponível e definimos símbolos (`_stack_top` e `_stack_bottom`) para marcar o topo e base da pilha. Quando o programa inicia, o registrador de pilha (SP) é carregado com o endereço `_stack_top` pelo código de *startup* gerado, garantindo que a pilha comece vazia no ponto mais alto designado da RAM. A pilha vai sendo utilizada conforme funções são chamadas, armazenando registros de ativação (*frames*) contendo as variáveis locais e endereços de retorno de cada chamada não finalizada.

## 2.7 Stack Pointer e Frame Pointer

O *stack pointer* (registrador R13, chamado SP) aponta para o topo da pilha em qualquer instante. A cada chamada de função, é empilhada uma nova estrutura de ativação contendo, tipicamente, o endereço de retorno (para o qual a execução deve voltar ao encerrar a função) e as variáveis locais e temporárias que não couberam em registradores. Também se utiliza um *frame pointer* fixo dentro de cada função (registrador R11, chamado FP). No prólogo de cada função, o compilador gera instruções para salvar o valor anterior de FP e do registrador de link (LR, que contém o endereço de retorno) na pilha, e então ajustar FP para o novo topo da pilha. Com isso, FP funciona como uma referência fixa ao início do frame atual, e as variáveis locais podem ser acessadas via deslocamentos fixos de FP.

O compilador TinyC segue esse padrão: no início de cada função gerada, há um **prólogo** que executa `push {fp, lr}` (armazenando o antigo FP e o retorno) e `mov fp, sp` (ajustando o frame pointer). Em seguida, reserva-se espaço para variáveis locais fazendo `sub sp, sp, #N`, onde *N* é o tamanho necessário (em bytes) para todas as variáveis locais e temporárias que não ficaram em registradores. No epílogo da função, antes de retornar, o compilador gera `mov sp, fp` (descartando toda a área local alocada) e `pop {fp, pc}`, que restaura o FP anterior e carrega o valor de PC (contador

de programa) a partir de LR, efetivamente retornando para a função chamadora. Esse esquema garante que a pilha permaneça consistente e que cada chamada de função tenha seu próprio espaço isolado para variáveis locais (evitando interferência entre frames).

Importante notar que a pilha na arquitetura ARM é do tipo *full descending* alinhada a 8 bytes, o que significa que a cada push/pop de 32 bits o SP desloca-se em múltiplos de 4 bytes, devendo manter alinhamento de 8 bytes ao entrar em funções. Nosso linker script já alinha o topo da pilha a 8 bytes, e o prólogo com push de dois registradores (8 bytes) mantém esse alinhamento.

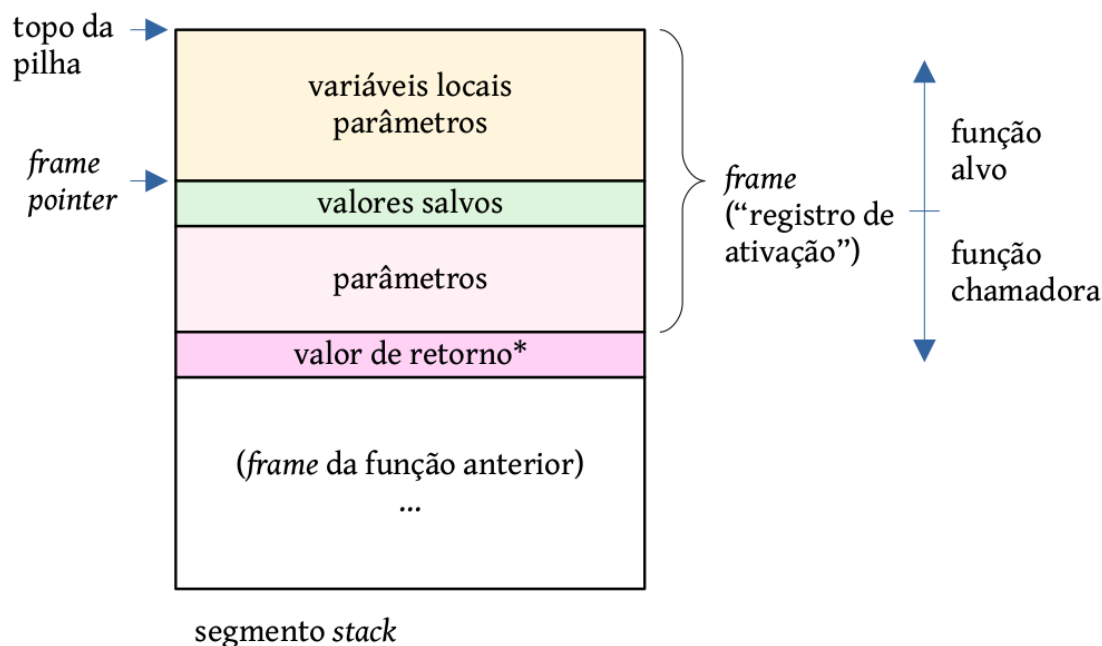


Figura ilustrativa do espaço de pilha (retirada da Wiki da disciplina)

## 2.8 Passagem de parâmetros e retorno

A convenção ABI do ARMv4 (AAPCS) especifica que os primeiros quatro argumentos de uma função são passados nos registradores R0–R3, e valores de retorno de funções de até 32 bits são retornados em R0. Nosso compilador implementa essa convenção: ao gerar a chamada de uma função, ele avalia os argumentos e os posiciona nos registradores apropriados antes de invocar a instrução de *branch-and-link* (bl) para transferir controle à função chamada. Especificamente, os argumentos são avaliados da direita para a esquerda de modo que o último argumento fique em R0, o penúltimo em R1, e assim sucessivamente. No trecho de geração de chamada abaixo, pode-se ver essa lógica implementada:

```
// Geração simplificada de chamada de função (até 4 args)
for (int i = node->arg_count - 1; i >= 0 && i < 4; i--) {
    gen_expr(node->args[i]);    // avalia argumento i em r0
    if (i != 0)
        fprintf(out, "    mov r%d, r0\n", i);
}
fprintf(out, "    bl %s\n", node->name);
```

*Trecho ilustrativo de geração de código para chamada de função, colocando argumentos em R0–R3 antes do branch.*

O compilador atual limita-se a utilizar no máximo 4 registradores e trata os demais como não suportados, esse detalhe fica simplificado. Após a execução da função chamada, o controle retorna via registrador PC restaurado pelo `pop {fp, pc}` do epílogo, e caso haja um valor de retorno, ele estará disponível em R0.

### 3. Metodologia

Para viabilizar a construção do compilador completo, o projeto foi planejado em etapas incrementais, cada uma adicionando funcionalidades sobre a anterior de forma controlada. A estratégia adotada envolve modularização clara das fases do compilador e um roteiro de evoluções sucessivas:

O compilador foi estruturado em módulos correspondentes a cada fase: um módulo de léxico (`src/lexer`), um de parsing (`src/parser`), um de semântica (`src/sema`) e um de geração de código (`src/code_generator`), além de um módulo principal que orquestra as chamadas (`src/main.c`). Cada fase tem sua implementação encapsulada e pode ser testada de forma independente através de opções de linha de comando do próprio compilador (por exemplo, `-tokens` para rodar apenas o lexer, `-ast` para obter a árvore sintática, etc.). Essa separação em componentes melhora a organização do código e permitiu desenvolver e depurar fase por fase. Uma decisão de design importante foi fazer o analisador léxico ler *todo o arquivo fonte de uma vez* e armazená-lo em memória, em vez de ler caractere a caractere sob demanda. Isso foi justificado para evitar operações de E/S frequentes (melhorando desempenho de leitura) e para simplificar o gerenciamento de posições (linha e coluna) dos tokens – já que com o texto em memória é fácil avançar e retroceder durante a tokenização. Assim, funções auxiliares como `read_file` carregam o conteúdo em um buffer único, e o lexer então itera sobre esse buffer para produzir os tokens.

A fase atual alcançada corresponde ao compilador básico funcionando, sobre o qual as próximas melhorias podem ser feitas.

## 4. Implementação

### 4.1 Análise Léxica (Lexer)

A implementação do analisador léxico reside em `src/lexer/lexer.c`, com definições importantes no arquivo de cabeçalho `include/token.h`. A estrutura central é o `Token`, que representa cada unidade léxica identificada.

```
typedef enum {
    TK_EOF = 0,
    TK_KW_INT,      TK_KW_VOID,      TK_KW_RETURN, TK_KW_CHAR,
    TK_KW_IF,       TK_KW_ELSE,      TK_KW_WHILE,  TK_KW_FOR,
    TK_IDENT,       TK_NUM,
    TK_SYM_PLUS,    TK_SYM_MINUS,    TK_SYM_STAR,  TK_SYM_SLASH,
    TK_AND,         TK_OR,           // '&&' '||'
    TK_SYM_SEMI,    TK_SYM_AMP,      TK_SYM_COMMA,
    TK_SYM_LPAREN,  TK_SYM_RPAREN,    TK_SYM_LBRACE, TK_SYM_RBRACE,
    TK_SYM_LT,      TK_SYM_GT,      TK_SYM_ASSIGN,
    TK_INC, TK_DEC,           // '++' '--'
    TK_EQ,  TK_NEQ,          // '==' '!='
    TK_LE,  TK_GE            // '<=' '>='
} TokenKind;

typedef struct Token {
    TokenKind kind;
    const char *lexeme;
    size_t len;
    int line, col;
    int ival; // valor inteiro, se for TK_NUM
} Token;
```

*Definição dos tipos de token (`TokenKind`) e da estrutura `Token`.*

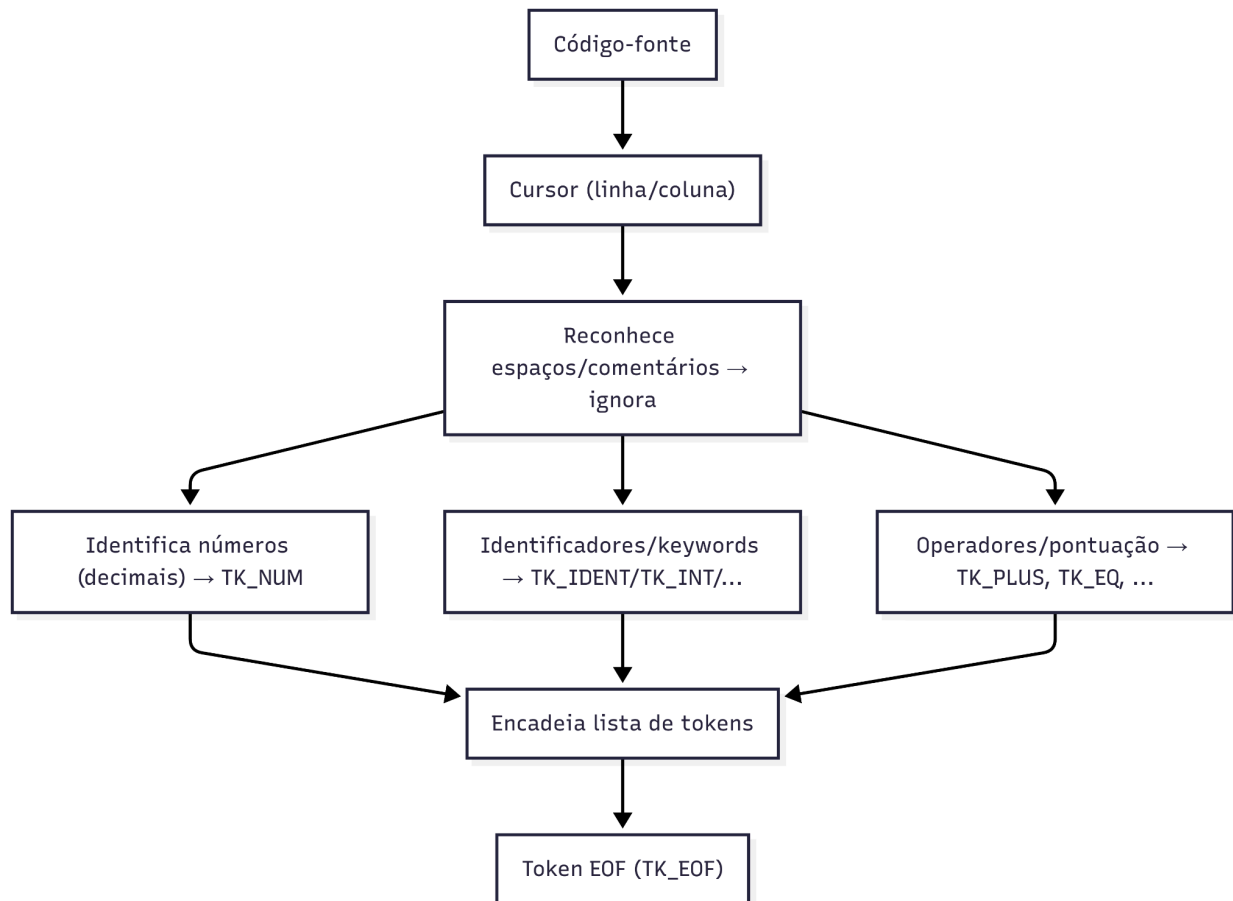
A enumeração `TokenKind` lista todos os tipos de tokens reconhecidos: palavras-chave (`TK_KW_INT`, `TK_KW_IF`, etc.), identificadores (`TK_IDENT`), literais numéricos (`TK_NUM`), operadores e símbolos de pontuação (como `TK_SYM_PLUS` para '+', `TK_EQ` para '==', etc.), incluindo também tokens compostos como operadores lógicos `&&` (`TK_AND`) e incrementos/decrementos. O token `TK_EOF` marca o fim do arquivo. A estrutura `Token` armazena o tipo (`kind`), um ponteiro para a lexema (a sequência de caracteres correspondente no texto fonte) com seu tamanho (`len`), a



posição de ocorrência (linha e coluna inicial) para mensagens de erro, e um campo `ival` para literais numéricos (armazenando o valor inteiro do número).

O lexer funciona lendo o buffer do código fonte e construindo tokens sequencialmente. Ele descarta espaços em branco e comentários (a implementação suporta comentários de linha `//` e de bloco `/* ... */`) e, a cada palavra ou símbolo reconhecido, cria um novo Token. Para facilitar o gerenciamento da sequência de tokens, todos os tokens gerados são encadeados em uma lista (ligada por ponteiros ou armazenada em vetor, conforme implementado). Ao final do processo de tokenização, um token especial `TK_EOF` é acrescentado para indicar término. Essa lista ligada de tokens é então entregue ao parser. A função principal do lexer é normalmente chamada `tokenize` e retorna um ponteiro para o início da lista de tokens. Caso encontre um caractere inválido ou uma sequência desconhecida, o lexer reporta um erro léxico (por exemplo, se houvesse um `@` isolado no código, não pertencente a nenhum token válido em C).

Um aspecto importante é que o lexer distingue tokens que são palavras-chave de identificadores comuns. Para isso, após ler um lexema alfabético, compara-se com a lista de palavras reservadas (`int`, `return`, etc.). Na implementação do TinyC, isso é feito de forma simples com comparações de strings; caso haja correspondência exata com uma palavra-chave conhecida, o token recebe o `TokenKind` apropriado (como `TK_KW_INT`), senão é categorizado como `TK_IDENT`. Literais numéricos são convertidos via `strtoul` ou lógica similar e armazenados em `ival`. Caracteres de pontuação e operadores de um ou dois caracteres são identificados com uma série de condicionais verificando o próximo caractere (por exemplo, se o próximo caractere após um `=` formar `==`, gera um token `TK_EQ` em vez de dois tokens separados).



*Figura do Diagrama do algoritmo do Lexer*

#### 4.2 Análise sintática (parser)

O parser implementado está no arquivo `src/parser/parser.c` (com definições em `src/parser/parser.h`). Trata-se de um parser recursivo descendente escrito manualmente, seguindo a gramática definida para o C--- suportado. A implementação define uma função recursiva para cada não-terminal relevante, organizando-as de forma a respeitar as precedências (por exemplo, há uma função `parse_expression` que internamente chama `parse_assignment`, que chama `parse_logical_or`, e assim por diante, seguindo a hierarquia da gramática definida em `parser.h`).

Conforme o parser consome tokens, ele constrói nós da Árvore de Sintaxe Abstrata (AST). A AST é representada pela estrutura `Node` definida em `parser.h`:

```

typedef enum {
    ND_ADD, ND_SUB, ND_MUL, ND_DIV, ND_LOGAND, ND_LOGOR,
    ND_EQ, ND_NE, ND_LT, ND_LE,
    ND_ASSIGN, ND_VAR, ND_NUM, ND_DEREF, ND_ADDR,
    ND_RETURN, ND_IF, ND_WHILE, ND_FOR,

```

```

    ND_BLOCK, ND_POSTINC, ND_POSTDEC, ND_CALL, ND_FUNC, ND_DECL,
} NodeKind;

typedef struct Node {
    NodeKind kind;
    Token *token;           // ponteiro para o token correspondente
    (opcional, p/ erros)
    Type *type;             // tipo do nó (determinado na análise
    semântica)
    struct Node *lhs;        // filho esquerdo / condição (depende do nó)
    struct Node *rhs;        // filho direito / bloco "then" (depende do
    nó)
    struct Node *els;        // filho "else" (se aplicável)
    int val;                 // valor literal (para ND_NUM)
    char *name;              // nome (para identificadores, nome de
    função ou variável)
    struct Node **args;      // lista de argumentos (para chamadas de
    função)
    int arg_count;
    struct Node **stmts;     // lista de statements (para blocos e corpo
    de funções)
    int stmt_count;
    struct Node *init, *cond, *inc; // componentes de for-loop (init,
    cond, inc)
} Node;

```

*Enumeração dos tipos de nó AST (NodeKind) e estrutura Node com seus campos.*

O enum NodeKind define todos os tipos de nós da AST que o compilador utiliza. Há nós para operações binárias aritméticas (ND\_ADD, ND\_SUB, etc.), operações lógicas (ND\_LOGAND, ND\_LOGOR), comparações (ND\_EQ, ND\_NE, ND\_LT, ND\_LE), atribuição (ND\_ASSIGN), uso de variável (ND\_VAR), literal numérico (ND\_NUM), operadores unários de ponteiro (ND\_ADDR para operador & e ND\_DEREF para \*), instruções e construções de controle (ND\_RETURN, ND\_IF, ND\_WHILE, ND\_FOR), bloco de código (ND\_BLOCK), operadores de pós-incremento/decremento (ND\_POSTINC, ND\_POSTDEC), chamada de função (ND\_CALL), definição de função (ND\_FUNC) e declaração de variável (ND\_DECL). Essa enumeração serve para que o código do compilador (especialmente o gerador de código) saiba distinguir que ação tomar para cada tipo de nó.

A estrutura Node em si é bastante versátil: alguns campos só são usados para certos tipos de nó, de acordo com a necessidade. Por exemplo, para um nó de tipo ND\_NUM, o campo val armazena o valor do literal; para um ND\_VAR ou ND\_FUNC, o

campo `name` guarda o nome do identificador; para nós `ND_IF`, `lhs` aponta para a expressão condicional, `rhs` para o bloco "then" e `els` para o bloco "else" (se houver); em um `ND_FOR`, são usados os campos `init`, `cond`, `inc` para os componentes do laço, e `rhs` para o corpo do loop. O campo `args` e `arg_count` são utilizados em nós de chamada de função `ND_CALL` (lista de expressões dos argumentos) *ou* em nós de definição de função `ND_FUNC` (lista de nós representando parâmetros formais, cada qual geralmente um `ND_DECL`). Já `stmts` e `stmt_count` são utilizados para nós de bloco `ND_BLOCK` (várias statements dentro de `{}`) e no corpo de funções `ND_FUNC`. O ponteiro `type` inicialmente pode ser preenchido como `NULL` durante a construção da AST, sendo atribuído posteriormente na análise semântica para indicar o tipo resultante daquela expressão ou o tipo de uma variável. O ponteiro `token` pode referenciar o token original associado ao nó.

O parser constrói a AST alocando nós dinamicamente (feita por funções auxiliares como `new_node(kind)`). Cada função de parsing retorna um ponteiro para um `Node` representando aquela construção. Por exemplo, a função `parse_expression()` pode chamar `parse_assignment()`, que retorna um nó do tipo `ND_ASSIGN` ou algo mais simples; `parse_assignment` internamente chama `parse_logical_or()`, e assim por diante.

A ordem das funções geralmente segue a ordem inversa de precedência: funções para *primary* (identificadores, literais, chamadas de função e expressões entre parênteses), depois unários, multiplicativos, aditivos, relacionais, igualdade, AND lógico, OR lógico, atribuição e finalmente expressões completas. Também há funções para cada tipo de statement e declarações.

Tome o seguinte exemplo de um código:

```
x = a + 5;.
```

O lexer produzirá tokens `[TK_IDENT("x"), TK_SYM_ASSIGN("="), TK_IDENT("a"), TK_SYM_PLUS("+"), TK_NUM(5), TK_SYM_SEMI(";")]`. O parser irá construir nós aninhados correspondentes: um nó `ND_ASSIGN` cuja parte esquerda `lhs` é um nó `ND_VAR("x")` e cuja parte direita `rhs` é um nó `ND_ADD`. Este nó `ND_ADD` terá como `lhs` um `ND_VAR("a")` e como `rhs` um `ND_NUM(5)`. A árvore resultante representaria a atribuição com a hierarquia correta (a soma de `a` e `5` como valor a ser atribuído em `x`). Essa AST pode ser visualizada:

```
(ASSIGN
  (VAR x)
  (ADD
    (VAR a)
    (NUM 5)
  )
)
```

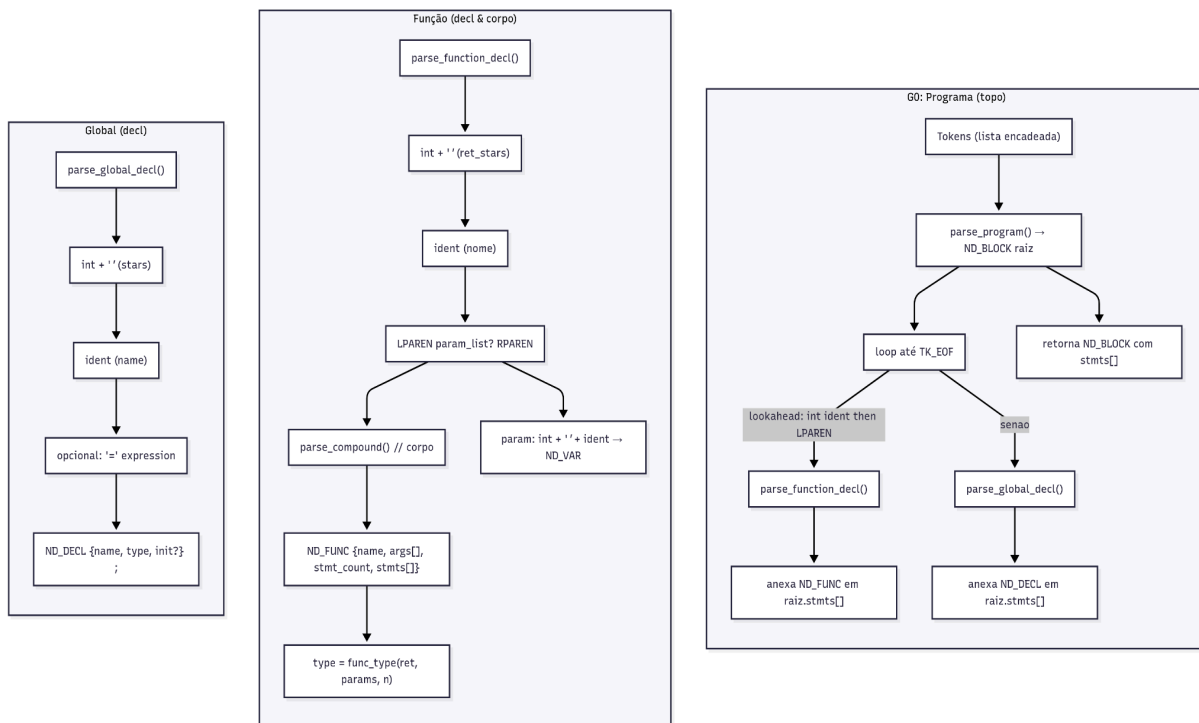
)

que é exatamente o formato impresso pela função `print_ast` no código.

Outro exemplo, para um `if` com `else`:

```
if (n <= 0) return 1; else return n * f(n-1);
```

a AST teria um nó `ND_IF` cuja expressão condicional (lhs) é um nó de comparação `ND_LE` (`n <= 0`), o bloco `then` (rhs) é um nó `ND_RETURN` com filho `NUM 1`, e o bloco `else` (els) é um nó `ND_RETURN` com filho representando `n * f(n-1)` (que por sua vez é um `ND_MUL` combinando `VAR n` e um `ND_CALL` do `f` com argumento `(n-1)`). Essa estrutura em árvore facilita as próximas fases, pois cada nó pode ser processado recursivamente de forma natural.



*Figura do diagrama de implementação do parser*

### 4.3 Análise Semântica

Após construir a AST, o compilador realiza a análise semântica, implementada em `src/sema/sema.c` (com definições em `src/sema/sema.h`). Essa fase percorre a AST e verifica várias condições semânticas, ao mesmo tempo em que anota os nós

com informações de tipo e organiza os símbolos (variáveis e funções) em tabelas de símbolos divididas por escopo.

Abaixo estão as principais estruturas definidas em `sema.h`.

```
typedef enum {
    SEMA_OK = 0,
    SEMA_UNDECLARED_IDENT,    // identificador não declarado
    SEMA_REDECLARED_IDENT,    // identificador já declarado
    SEMA_TYPE_MISMATCH,       // tipos incompatíveis em operação
    SEMA_TOO_MANY_ARGS,       // muitos argumentos em chamada de
    função
    SEMA_ARG_TYPE_MISMATCH,    // tipo de argumento incompatível com
    parâmetro
    // ... outros códigos de erro
} SemaErrorCode;

typedef struct SemaSymbol {
    const char *name;
    NodeKind kind;           // ND_VAR ou ND_FUNC
    Type *type;
    int stack_offset;        // deslocamento no frame (para variáveis
    locais)
    struct SemaSymbol *next;
} SemaSymbol;

typedef struct SemaScope {
    SemaSymbol *symbols;
    struct SemaScope *parent;
} SemaScope;

typedef struct {
    SemaScope *current_scope;
    bool error_reported;
    Type *current_ret;       // tipo de retorno da função atual (para
    validar return)
    int next_offset;         // próximo offset disponível para local
    (negativo, acumulando tamanho)
} SemaContext;
```

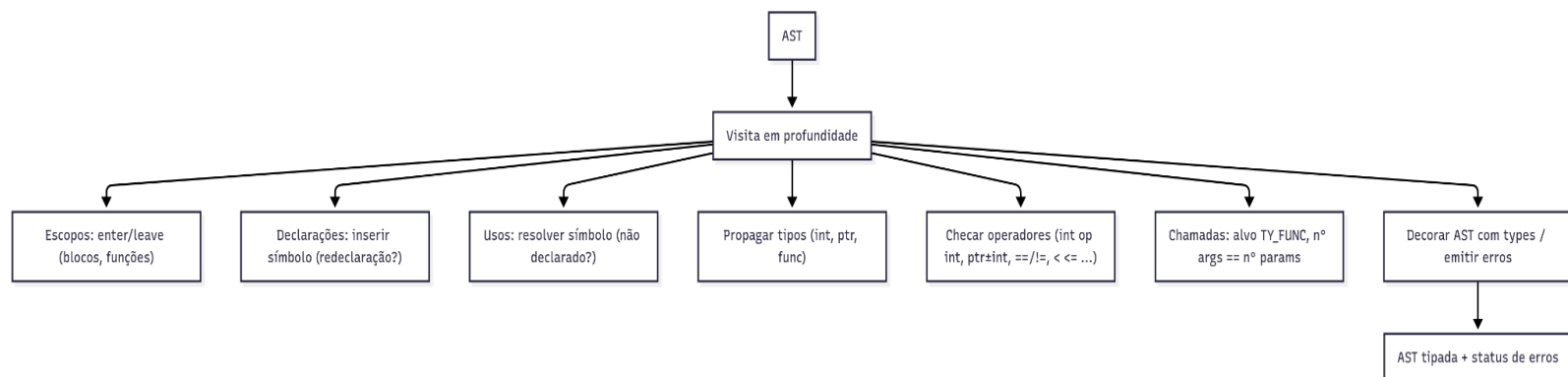
## *Definições de códigos de erro semântico e estruturas para símbolos, escopos e contexto semântico.*

SemaSymbol representa a entrada de um símbolo na tabela de símbolos. Cada símbolo tem um nome (`name`), um tipo (`type`, usando a estrutura `Type` definida em `include/type.h`), e um *kind* que indica se é uma variável (local ou global) ou uma função (usamos o próprio `NodeKind` do nó correspondente, `ND_VAR` ou `ND_FUNC`). O campo `stack_offset` armazena, para variáveis locais, o deslocamento em bytes a partir do frame pointer onde essa variável foi alocada. Esse campo é determinado durante a análise semântica e depois usado na geração de código para acessar a variável correta na pilha. Os símbolos estão encadeados através do ponteiro `next`, o que permite que múltiplos símbolos residam em uma mesma lista (por exemplo, símbolos do mesmo escopo).

SemaScope representa um escopo léxico (por exemplo, nível global, ou dentro de uma função, ou dentro de um bloco composto). Cada escopo tem um ponteiro para uma lista de símbolos declarados nele (`symbols`) e um ponteiro para o escopo pai (o escopo envolvente). Isso permite uma busca hierárquica: se um identificador não é encontrado no escopo corrente, a busca pode subir para escopos superiores.

SemaContext guarda o estado global durante a análise semântica. Contém um ponteiro para o escopo atual (`current_scope`), um sinalizador `error_reported` para indicar se algum erro semântico já foi encontrado (evitando mensagens redundantes), o tipo de retorno da função atual sendo analisada (`current_ret`), e um contador `next_offset` para calcular os offsets das variáveis locais na stack. O offset começa em 0 no início de uma função e vai acumulando valores negativos (decrementando de 4 em 4 bytes, por exemplo) conforme novas variáveis locais são declaradas, já que cada nova variável ocupará espaço no frame da função. Assim, `next_offset` acaba representando o tamanho total necessário para locais, em valor negativo.

O processo de análise semântica envolve percorrer a AST (tipicamente com uma função recursiva `sema_analyze(Node *node)` ou similar) e tomando ações dependendo do tipo de nó:



### *Figura do diagrama de implementação da análise semântica*

**Declarações de Variáveis (ND\_DECL):** Ao encontrar um nó de declaração, o analisador semântico insere o símbolo na tabela de símbolos do escopo atual. Isso é feito pela função `sema_declare(ctx, name, kind, type)`. Ela verifica se já existe um símbolo com o mesmo nome no mesmo escopo; em caso afirmativo, registra um erro de símbolo redeclarado (`SEMA_REDECLARED_IDENT`). Se for uma variável local, calcula um novo offset: `ctx->next_offset` é decrementado em 4 (tamanho de um int, no caso atual) e esse valor é atribuído a `stack_offset` do símbolo. Por exemplo, se `next_offset` era 0 e encontra-se `int x;`, passa a -4 e `x` recebe offset -4 (que significa que `x` ficará em `[FP, #-4]`). Uma segunda variável local receberia offset -8, e assim por diante. Variáveis globais também são registradas como símbolos (provavelmente com offset 0 ou ignorado, já que acessos a globais serão resolvidos de outra forma, em nosso caso gerando labels no assembly). Cada símbolo aponta para um `Type` apropriado. O sistema de tipos é simples: `include/type.h` define a enum `TypeKind { TY_INT, TY_PTR, TY_FUNC, TY_VOID, ... }` e a estrutura `Type` que pode referenciar outro tipo base (para ponteiros ou arrays) ou uma lista de parâmetros (para funções). Durante a análise semântica de uma declaração, se for encontrado, por exemplo, `int *p;`, o compilador criaria um `Type` de tipo `TY_PTR` cujo base aponta para `ty_int`, e associaria isso ao símbolo `p`.

**Uso de Variáveis (ND\_VAR) e Resolução de Identificadores:** Quando o analisador semântico encontra um nó de variável (que o parser criou ao ver um identificador em uma expressão, por exemplo), ele precisa resolver esse nome para um símbolo declarado. Isso é feito com `sema_resolve(ctx, name)`, que procura o nome nos escopos do `ctx` começando pelo atual e subindo para os pais. Se encontrar, associa o tipo do símbolo ao campo `node->type` e pode armazenar no nó uma referência ao símbolo ou seu offset conforme necessário. Se não encontrar o nome em nenhum escopo, emite um erro `SEMA_UNDECLARED_IDENT` indicando uso de identificador não declarado. No caso de função chamada (`ND_CALL`), é semelhante: verifica-se se existe um símbolo de função com aquele nome. Se não, erro de função não declarada; se sim, obtém-se a lista de parâmetros esperados para comparar número de argumentos e tipos (checagens que podem disparar erros `SEMA_TOO_MANY_ARGS` ou `SEMA_ARG_TYPE_MISMATCH`). Durante essa resolução, o compilador também aproveita para anotar o nó de chamada com o tipo de retorno da função (para uso em expressões maiores).

**Expressões e Verificação de Tipos:** Para operações binárias (como soma, comparação, etc.), a análise semântica verifica os tipos dos operandos e determina o tipo resultante. Como só há um tipo inteiro, a verificação é simples: espera-se que ambos operandos sejam do mesmo tipo básico. Se `lhs->type` e `rhs->type` divergem de forma incompatível, lança um erro `SEMA_TYPE_MISMATCH`. O nó da AST é então anotado: por exemplo, para `ND_ADD`, define-se `node->type = ty_int` (porque soma de int com int resulta int). Para comparações relacionais ou igualdade, pode-se decidir



que o tipo resultante é um int (em C, booleanos são representados como int 0 ou 1), então esses nós também ficam com `ty_int`.

**Retorno de Função:** Ao entrar na análise de uma função (`ND_FUNC`), o contexto `current_ret` é ajustado para o tipo de retorno dessa função (por exemplo, `ty_int` para uma função que retorna int). Assim, ao analisar um nó `ND_RETURN` dentro dessa função, o analisador verifica se a expressão de retorno existe e se seu tipo é compatível com `current_ret`. Se a função deve retornar int e encontra-se um retorno vazio ou de tipo incompatível, isso seria um erro. No C---, todas as funções retornam int e sempre se espera um valor após `return`. A análise semântica ajusta o tipo do nó `ND_RETURN` para ser o tipo da função.

**Contextos de Escopo:** O `SemaContext` maneja escopos aninhados. Ao iniciar a análise de uma função, é criado um escopo local novo para seus parâmetros e variáveis locais, cujo pai é o escopo global. Ao entrar em um bloco `{ ... }` dentro de uma função (por exemplo, corpo de um `if` ou `laço`), cria-se outro escopo filho (via `sema_enter_scope`). Isso permite declarar variáveis locais com nomes que não interferem fora do bloco. Ao sair do bloco, faz-se `sema_leave_scope`, retornando ao escopo pai.

Durante a passagem semântica, o compilador também aproveita para contabilizar o espaço necessário para variáveis locais de cada função. Conforme mencionado, `ctx->next_offset` acumula o tamanho (negativo) total. Quando a análise de uma função termina, esse valor (tomado em valor absoluto) representa quantos bytes devem ser reservados no prólogo da função. Por exemplo, se foram declaradas 3 variáveis locais do tipo int, `next_offset` chegaria a -12, indicando 12 bytes (cada int 4 bytes) de armazenamento local necessários. Esse valor é guardado provavelmente no próprio nó da função ou em alguma estrutura associada.

Além disso, os símbolos das variáveis locais recebem os offsets calculados (e esses offsets são armazenados em cada `SemaSymbol` correspondente).

Em termos de implementação, a função principal é `sema_analyze(SemaContext *ctx, Node *root)`, que retorna um código de erro geral (`SEMA_OK` se tudo certo, ou algum dos códigos se houve erro). Ela inicializa o contexto (escopo global, etc.), declara eventualmente símbolos das funções encontradas no nível global antes de analisá-las (para permitir chamadas recíprocas simples, dependendo da abordagem) e então percorre a AST recursivamente.

Para ilustrar, suponha o seguinte pequeno código C de exemplo que o compilador suporta:

```
int a;
int f(int n) {
    if (n <= 1) return 1;
```

```
int x = f(n-1);  
return n * x;  
}
```

Durante a análise semântica: - O símbolo da variável global `a` (tipo `int`, global) é adicionado no escopo global. - O símbolo da função `f` é adicionado no escopo global, com tipo de função que retorna `int` e um parâmetro `int`. - Ao analisar a definição de `f`, entra-se em um novo escopo local; o parâmetro `n` é declarado nesse escopo com offset -4 (primeiro local, ainda que seja parâmetro, podemos tratá-lo como variável local armazenada no frame) e símbolo de nome "`n`". - Entra-se no corpo: ao encontrar o `if`, checa-se que `n` está declarado (resolve para o símbolo do parâmetro, do tipo `int`) e que a constante `1` é `int`; `n <= 1` é ok e resulta em `int` (bem que poderíamos considerar `bool`, mas no nosso contexto `int` serve). - O `return 1;` é verificado: a função espera retorno `int`, e está retornando um `int` literal, então tudo certo. - Sai do `if` (volta ao escopo de `f`); encontra a declaração `int x = f(n-1);`. Declara símbolo `x` no escopo atual: offset -8 para `x`. Depois analisa a inicialização: `f(n-1)`: - Verifica chamada de `f`: símbolo `f` resolvido no global, é função que espera 1 argumento `int` e retorna `int`. - Analisa argumentos: `n-1` => `n` é `int`, `1` é `int`, operação - resulta `int`, ok. - Número de argumentos batem (1 esperado, 1 fornecido), tipos batem (`int` para `int`), então sem erro. - O nó `ND_CALL f` recebe `type = ty_int` (tipo de retorno). - A declaração de `x` então é válida e `x` recebe o valor retornado de `f(n-1)`. O tipo de `x` é `int` (pela declaração). - Por fim o `return n * x;`: Verifica-se `n` e `x` (ambos `int`), `*` resulta `int`, que é compatível com `current_ret` (`int`), ok. - Sem erros semânticos; total de offset para `f` é -8 (duas variáveis locais: `n` e `x`), indicando 8 bytes a reservar no frame (parâmetro geralmente já passado via registrador, mas nosso compilador guarda em frame para simplicidade).

Esse passo semântico garante que, ao prosseguir para a geração de código, todos os identificadores referenciem posições ou endereços válidos, e que a AST tenha informação suficiente de tipos para orientar a seleção de instruções (por exemplo, no futuro, se houvesse `float`, saberíamos que teríamos que usar instruções de ponto flutuante ou conversões).

#### 4.4 Geração de código

A última fase, a geração de código assembly ARM, é implementada em `src/code_generator/code_generator.c`. Essa fase visita a AST (possivelmente utilizando as anotações de tipos e símbolos feitas pela semântica) e emite texto assembly equivalente. O gerador está organizado para produzir um arquivo `.s` completo contendo: diretivas de seção (`.text`, `.data`), um código de *startup* (rótulo `_start`) para inicialização bare-metal, definições de dados globais, e as seções de código de cada função definida no fonte em C.

Alguns elementos principais da implementação:

- **Gerenciamento de Registradores Locais e Pilha:** O gerador define estruturas internas para lidar com variáveis locais. Em vez de depender apenas da tabela de símbolos construída na semântica, o código atual reconstrói uma lista de variáveis locais por função. No topo de `code_generator.c`, define-se a seguinte struct:

```
typedef struct { const char *name; int offset; } Local;  
static Local locals[256];  
static int local_count;  
static int stack_size;
```

Ou seja, um array estático para armazenar até 256 variáveis locais por função, e variáveis globais estáticas para contar locais e a quantidade de bytes `stack_size` ocupados pelos locais. Há também uma função `add_local(const char *name)` que incrementa o `stack_size` e registra o nome com um offset negativo correspondente. Essa função simplesmente considera cada local ocupando 4 bytes: ela calcula `off = stack_size + 4`, soma 4 a `stack_size` e então armazena um registro `Local{name, -off}` na lista `locals`. Assim, a primeira variável adicionada terá offset -4, a segunda -8, etc., condizente com nossa expectativa da análise semântica. Existe também `lookup_local(const char *name)` que busca na lista pelo nome e retorna o offset encontrado. Essa simplificação supõe que nomes de variáveis locais são únicos por função (o que é verdade, dada a verificação semântica de redeclaração no mesmo escopo).

Antes de gerar o código de uma função, o gerador precisa montar a lista de locais. Isso é feito percorrendo a AST da função em busca de nós `ND_DECL` que representem declarações locais. A função `collect_locals(Node *node)` cumpre esse papel recursivamente. Ela percorre recursivamente a árvore da função e, para cada nó de declaração encontrado, invoca `add_local`. Isso coleta variáveis declaradas dentro de qualquer bloco (inclusive internos de `for`, etc.). Também considera parâmetros da função (inseridos logo antes da coleta via um loop que faz `add_local` em cada parâmetro formal). Ao final desse processo, `stack_size` conterá o total de bytes necessários para todas as variáveis locais e parâmetros. O código ainda alinha `stack_size` a múltiplo de 4 (já está, pois cada local conta 4 bytes, mas por precaução há um ajuste para garantir alinhamento de word).

- **Emissão do Prólogo e Epílogo de Funções:** O gerador de código, ao processar um nó de função `ND_FUNC`, emite primeiro o rótulo global da função e as instruções de prólogo padrão. No código:

```
fprintf(out, ".global %s\n", fn->name);  
fprintf(out, "%s:\n", fn->name);  
fprintf(out, "    push {fp, lr}\n");  
fprintf(out, "    mov fp, sp\n");  
if (stack_size)
```

```
fprintf(out, "    sub sp, sp, #d\n", stack_size);
```

*Geração do prólogo de função, com ajuste de frame e alocação de locais.*

Isso corresponde ao comportamento teórico: salvar FP e LR, ajustar FP e reservar espaço na pilha para variáveis locais. O valor de `stack_size` calculado anteriormente é usado aqui – se for zero (nenhuma variável local), a instrução de subtração é omitida. Em seguida, o gerador move os argumentos que chegaram em registradores para suas posições na stack se necessário. Sabe-se que, conforme convenção, ao entrar na função, R0–R3 contêm até quatro argumentos. O código então executa:

```
for (int i = 0; i < fn->arg_count && i < 4; i++) {  
    int off = lookup_local(fn->args[i]->name);  
    fprintf(out, "    str r%d, [fp, #d]\n", i, off);  
}
```

*Armazenamento dos parâmetros nos endereços locais na pilha.*

Isso pega cada parâmetro formal (que também foi inserido em `locals` com um offset negativo) e gera uma instrução STR do registrador correspondente para a posição relativa a FP. Por exemplo, para o primeiro parâmetro (`i=0`, offset tipicamente `-4`), gera `str r0, [fp, #-4]`. Dessa forma, mesmo após o código do usuário alterar R0 etc., os valores originais dos parâmetros ficam guardados no frame da função, acessíveis pelo offset fixo. Essa abordagem facilita nosso simples gerador, pois a partir daqui podemos tratar parâmetros como variáveis locais (acessando-as na pilha). Observa-se que não são tratados mais que 4 parâmetros – caso `arg_count > 4`, ou seja, o compilador simplesmente ignora parâmetros além de 4.

Ao final da geração do corpo da função, o gerador prepara o epílogo. Foi adotada a convenção de inserir dois rótulos auxiliares: um rótulo de *fallthrough* (queda) e um rótulo de epílogo propriamente. Isso serve para tratar funções que chegam ao fim sem um `return` explícito: o padrão em C é que, se `main` acabar sem `return`, retorne 0, e para outras funções não-void o comportamento é indefinido, mas aqui escolhemos forçar um retorno 0. Assim, o gerador, após emitir código de todos os `statements` do corpo, coloca:

```
char epilogue[32], fallthrough[32];  
snprintf(epilogue, sizeof epilogue, ".Lep_%s", fn->name);  
snprintf(fallthrough, sizeof fallthrough, ".Lftr_%s", fn->name);  
...  
// após gerar todos stmts:  
fprintf(out, "%s:\n    mov r0, #0\n    b %s\n", fallthrough,
```

```
epilogue);
// epílogo:
fprintf(out, "%s:\n    mov sp, fp\n    pop {fp, pc}\n", epilogue);
```

*Inserção de código para retorno implícito 0 e epílogo de restauração do frame*

O rótulo `.Lftr_nomeFunc` marca o ponto de saída normal (queda pelo fim da função) definindo `r0 = 0` como valor de retorno padrão e saltando para `.Lep_nomeFunc`. O rótulo `.Lep_nomeFunc` marca o epílogo comum onde restaura `sp` e faz o `pop {fp, pc}`. Durante a geração dos statements da função, se o analisador semântico detectou que a função deveria retornar algo mas não retornou em algum caminho, sempre se garante um retorno zero implícito.

#### 4.4.1 Geração de Código de Comandos e Expressões

O gerador de código define funções recursivas para processar expressões (`gen_expr(Node *node)`) e comandos/statements (`gen_stmt(Node *node, const char *ret_label)`). A função `gen_stmt` lida com nós de controle de fluxo e declarações, enquanto `gen_expr` lida com nós que produzem valores.

**Variáveis e Literais:** Para um nó `ND_NUM`, a geração é feita da seguinte maneira: coloca-se o valor imediato em um registrador (`R0`). O gerador emite `MOV r0, #<valor>`. Para um nó `ND_VAR`, há duas situações: se a variável é local (existe um offset em `locals`), o gerador precisa carregar de memória; se é global, o gerador acessa via um literal de endereço. O código verifica `int off = lookup_local(name)`: se encontrar um offset (diferente de 0), significa variável local, então em vez de diretamente carregar valor, primeiro obtém o endereço: `add r0, fp, #off` e depois faz `ldr r0, [r0]` para carregar o conteúdo. Caso `lookup_local` retorne 0 (não achou), assume-se tratar de uma variável global, então o gerador utiliza uma *pseudo-instrução* do assembler: `ldr r0, =nome`. Essa sintaxe carrega em `r0` o endereço do símbolo global `nome` (resolvido pelo linker) e depois o código faz `ldr r0, [r0]` para obter o valor armazenado naquele endereço. Assim, tanto para locais quanto para globais, ao final de um `gen_expr` de `ND_VAR`, `R0` contém o valor da variável.

**Atribuição (`ND_ASSIGN`):** A atribuição envolve avaliar primeiro a localização esquerda e depois o valor direito, e armazená-lo. O compilador gera código para obter o endereço do operando esquerdo, depois o valor do direito, e então efetua o store. No código:

```
gen_addr(node->lhs);      // calcula endereço do destino em r0
fprintf(out, "    push {r0}\n"); // salva endereço na pilha
gen_expr(node->rhs);      // calcula valor em r0
```

```
fprintf(out, "    pop {r1}\n");    // recupera endereço em r1
fprintf(out, "    str r0, [r1]\n");
```

*Sequência gerada para atribuição  $Lhs = rhs$*

A pilha é usada para guardar temporariamente o endereço do destino enquanto computamos a expressão do lado direito. A função auxiliar `gen_addr` gera o endereço de um nó (ao contrário de `gen_expr` que gera o valor). Para um nó `ND_VAR`, por exemplo, `gen_addr` gera exatamente o que `gen_expr` faria até antes do `ldr`: se for local, faz `add r0, fp, #off` (endereço da variável); se for global, faz `ldr r0, =nome` (endereço do símbolo global). Assim, após `gen_addr(lhs)`, `R0` aponta para onde o valor deve ser armazenado. O endereço é empilhado, calcula `rhs` em `R0`, depois restaura o endereço para `R1` e finalmente grava `r0` em `[r1]`.

**Operações Aritméticas e Lógicas:** Para uma operação binária como soma (`ND_ADD`), a convenção escolhida é: computa o operando esquerdo, empilha; computa o direito; desempilha o esquerdo para outro registrador; aplica a operação. No caso de `ADD`:

```
gen_expr(node->lhs);
fprintf(out, "    push {r0}\n");
gen_expr(node->rhs);
fprintf(out, "    pop {r1}\n");
fprintf(out, "    add r0, r1, r0\n");
```

*Geração simplificada para  $r0 = Lhs + rhs$  (similar para `SUB`)*

Após isso, `R0` contém a soma. `SUB` é semelhante, mas atenta à ordem (faz `r1 - r0` invertendo antes de subtrair). Multiplicação usa a instrução `MUL r0, r1, r0` do ARM (que faz `r0 = r1 * r0`), após preparar `r1` e `r0` da mesma forma. Divisão inteira não tem instrução única; a implementação usa a convenção da EABI: move os operandos para `R0` e `R1` conforme esperado por `__aeabi_idiv` (que computa `r0/r1`). O código:

```
gen_expr(lhs);
push {r0}
gen_expr(rhs);
pop {r1}
mov r2, r0    @ preserva segundo operando
mov r0, r1    @ primeiro operando em r0
mov r1, r2    @ segundo em r1
bl __aeabi_idiv
```

Assim, após chamar a função de divisão (fornecida pela biblioteca padrão), o resultado estará em `r0` (convenção EABI). Esse é um exemplo de interação com código

externo (rotina de runtime), mas que o linker resolve automaticamente ao vincular com libgcc.

Para operadores lógicos e de comparação (==, !=, <, <= etc.), a implementação segue um padrão: avalia lhs e rhs similarmente empilhando um dos valores, depois utiliza instruções de comparação ARM. Por exemplo, para ==:

```
cmp r1, r0
mov r0, #0
moveq r0, #1
```

Este trecho compara r1 e r0 e então coloca 0 em r0 (falso) e, se a condição de igualdade for verdadeira (EQ flag set), move 1 para r0. Dessa forma, após a sequência, r0 terá 1 se r1==r0, senão 0, implementando a lógica booleana. O gerador escolhe o sufixo condicional com base no tipo de comparação (por exemplo, "ne" para !=, "lt" para <, "le" para <=).

**Controle de Fluxo:** O gerador gen\_stmt lida com nós como ND\_IF, ND\_WHILE, ND\_FOR, ND\_RETURN, ND\_BLOCK:

- Para ND\_RETURN, se há uma expressão, chama gen\_expr nela para colocar o valor em r0, então emite um branch incondicional para o rótulo de epílogo (recebido via parâmetro ret\_label). Isso garante que a execução pule para a rotina de finalização da função com o valor de retorno pronto em r0.
- Para ND\_IF, o gerador cria labels para a parte else e fim (.LelseX e .LendX com X sendo um id único). Ele gera código da condição (colocando resultado em r0), depois CMP r0, #0 e um BEQ .LelseX (se condição falsa, pula para else). Em seguida, gera o bloco then (gen\_stmt no rhs). Depois, se havia um else, insere um B .LendX para pular o else, emite o rótulo .LelseX e gera o bloco else (gen\_stmt no els), e por fim emite o rótulo .LendX. Se não há else, o BEQ pula direto para .LendX e o código then cai diretamente em .LendX sem jump extra. Isso implementa corretamente o fluxo do if.
- ND\_WHILE é similar: cria labels .LbeginY e .LendwY, emite o label begin, gera a condição, faz CMP r0, #0 e BEQ .LendwY para sair do loop se condição falsa, caso contrário gera o corpo do loop, depois emite um B .LbeginY para voltar ao início. Ao final emite .LendwY.
- ND\_FOR precisa tratar até três componentes (init, cond, inc). O gerador define .LforZ e .LendfZ labels. Gera primeiro a parte de inicialização (que pode ser uma declaração ou expressão; na AST nosso init pode ser um nó ND\_DECL ou ND\_EXPR sem efeito de resultado). Depois emite o label do início do loop, gera a condição se existir: se não existir,



significa laço de tipo `for(;;)` com condição implícita verdadeira. Se gerar a condição, insere `CMP r0, #0` e `BEQ .LendfZ` para terminar caso falsa. Então gera o corpo (rhs). Ao final do corpo, gera a expressão de incremento se existir, e então um branch de volta para `.LforZ`. Finalmente, emite o label `.LendfZ` após o loop. Com isso cobre-se todas as variações de `for` (inicialização ausente, condição ausente, incremento ausente são tratados com os ifs correspondentes).

- `ND_BLOCK` simplesmente itera por todos os sub-nós em `node->stmts` e chama `gen_stmt` para cada um. Em termos de escopo, as variáveis locais definidas nesse bloco já foram coletadas e inseridas na lista `locals`, mas nada especial é necessário na geração além de gerar os statements.
- `ND_DECL` dentro de `gen_stmt`: Uma declaração de variável local não produz, em si, código executável de ação (a alocação de espaço já foi resolvida no prólogo). Contudo, se a declaração tiver uma inicialização (e.g. `int x = 5;` ou `int x = expr;`), é preciso gerar código para avaliar a expressão e armazená-la na variável. O gerador trata isso no case `ND_DECL`: obtém o offset via `lookup_local`, gera o código da expressão de inicialização (se houver) e então emite `STR r0, [fp, #<off>]` para guardar o valor calculado na posição da variável. Caso não haja inicializador, simplesmente não há código.

Até aqui, cobrimos os principais tipos de nó. Nota-se que nós como `ND_ADDR` e `ND_DEREF` (operadores unários `&` e `*`) estão parcialmente implementados: `gen_expr(ND_ADDR)` apenas delega para `gen_addr` do operando, efetivamente colocando em `r0` o endereço do operando; `gen_expr(ND_DEREF)` avalia a subexpressão (que deve resultar em um endereço em `r0`), e então faz `ldr r0, [r0]` para obter o valor apontado. Isso indicaria suporte a ponteiros, embora na fase atual talvez não houvesse sintaxe para produzir esses nós.

**Dados Globais:** Além do código das funções, o gerador lida com variáveis globais. No AST, as declarações globais aparecem como nós `ND_DECL` no nível do programa (filhos diretos do nó raiz da AST, que é um bloco representando o programa inteiro). O gerador de código, na função `codegen_to_file`, passa primeiro pelos nós do topo identificando declarações globais. Se encontrar ao menos uma, ele emite a diretiva `.data` (seção de dados) e para cada global gera um rótulo com o nome da variável seguido de um *word* com o valor inicial. Por exemplo, para `int g = 10;` gera:

```
.data
g:
    .word 10
```

e para `int h;` (sem inicializador) gera `.word 0` como valor padrão. Isso aloca 4 bytes para cada global, iniciando com o valor especificado (ou zero se não especificado).



Essas labels serão referenciadas no código .text via instruções ldr r0, =g seguidas de ldr/str conforme necessário, como vimos.

**Seção de Texto e Ponto de Entrada:** Por fim, o gerador escreve a seção de código. Ele começa emitindo a diretiva .text seguida do rótulo \_start global. Em \_start, o código inicializa o ambiente de execução minimal: carrega o registrador SP com o endereço \_stack\_top (fornecido pelo linker script) e chama a função main do programa. Após retornar de main, insere as instruções de semihosting para encerrar (R7 = 0x18, SVC 0x123456) como já descrito. Portanto, \_start atua como a rotina de bootstrap que num sistema operacional real seria substituída pelo carregador invocando o main do programa. Aqui foi gerado manualmente para poder rodar em QEMU sem um SO.

**Exemplo completo:** tome a seguinte função que calcula fatorial.

```
int f(int n) {
    if (n <= 1) return 1;
    int x = f(n-1);
    return n * x;
}
```

O código assembly gerado (simplificado) seria algo como:

```
.text
.global f
f:
    push    {fp, lr}
    mov     fp, sp
    sub     sp, sp, #8        @ aloca 8 bytes (x e n salvam)
    str     r0, [fp, #-4]     @ salva param n em [FP-4]
.Lbegin0:
    ldr     r0, [fp, #-4]     @ carrega n
    cmp     r0, #1
    ble     .Lret1            @ se n <= 1, retorna 1
    @ else:
    ldr     r0, [fp, #-4]     @ carrega n
    sub     r0, r0, #1
    bl      f                 @ chama f(n-1), resultado em r0
    str     r0, [fp, #-8]     @ guarda retorno em x (offset -8)
    ldr     r1, [fp, #-4]     @ carrega n em r1
    ldr     r0, [fp, #-8]     @ carrega x em r0
    mul     r0, r1, r0        @ r0 = n * x
```

```

    b        .Lep_f
.Lret1:
    mov     r0, #1
.Lep_f:
    mov     sp, fp
    pop     {fp, pc}

```

(O rótulo `.Lret1` seria análogo ao `.Lftr_f` ou `.Lelse0` gerados na lógica do `if`). Esse código ilustra todos os elementos: prólogo/epílogo, decisão condicional, chamada recursiva com preservação de LR e armazenamento do resultado, e cálculo final, condizente com o comportamento esperado.

## 5. Etapa Concluída (Estado Atual do Projeto)

A implementação atual corresponde ao término da Fase 1 planejada. Em resumo, as funcionalidades já implementadas com sucesso incluem:

- **Tipos e Literais Suportados:** O compilador lida com o tipo básico `int` (32 bits) e reconhece literais inteiros decimais, hexadecimais (se implementado) e literais de caractere. Literais de caractere, como `'A'`, são tratados internamente como inteiros correspondentes ao código ASCII do caractere. Não há suporte ainda para outros tipos primitivos (como `float`, `double`) ou tipos compostos (arrays, structs) nesta fase.
- **Estruturas de Controle:** Estão implementados os comandos de fluxo essenciais da linguagem C: condicionais `if` com cláusula opcional `else`, loops `while` e `for`, incluindo as variações com inicialização, condição e incremento opcionais, conforme demonstrado pela gramática e pelo código gerado. Também é suportado o comando `return` dentro de funções para retornar do procedimento (nas funções não `void`, espera-se uma expressão após o `return`; funções do tipo `void` ainda não foram introduzidas, portanto qualquer `return`; sem valor seria tratado como erro).
- **Escopos e Variáveis:** O compilador permite declarações de variáveis globais (fora de qualquer função) do tipo `int`, bem como variáveis locais dentro de funções e dentro de blocos. Variáveis locais e parâmetros são alocadas na pilha com endereços fixos (offsets do FP) determinados na análise semântica. Verificações estão implementadas para impedir uso de variáveis não declaradas em um escopo e para evitar declarações duplicadas no mesmo escopo. Cada função pode ter no máximo 4 parâmetros formais (devido à limitação na passagem de argumentos via registradores). Variáveis globais são alocadas na seção de dados e acessadas pelo código gerado via referências absolutas (pseudo-instrução `ldr =label`). Também foi configurado um endereço base para a

memória RAM no linker script (0x40000000) e um tamanho fixo para a pilha (8 KiB) cujos limites são marcados por símbolos no binário. O endereço inicial da pilha é definido pelo símbolo `_stack_top` no script de linker e carregado em SP pelo código `_start` gerado (“gambiarra no QEMU”), aqui configurado para ser 8K acima do início da BSS. Na prática, isso significa que o stack pointer inicia em (`endereço_inicial_RAM + 0x10000 + tamanho_texto + tamanho_data + tamanho_bss + 8KB`). Essa configuração foi escolhida para QEMU (máquina virt) e funciona adequadamente para os testes conduzidos.

- **Expressões e Operadores:** A maioria dos operadores aritméticos e lógicos do C foram implementados no subconjunto: soma, subtração, multiplicação, divisão inteira, comparação (`==`, `!=`, `<`, `<=`, `>`, `>=`) e operadores lógicos booleanos `&&` e `||` (estes últimos, no entanto, na implementação atual podem não ter avaliação curto-circuito; a geração de código indicada compara valores inteiros resultantes de expressões tratadas como booleanas, mas não implementa a lógica de saltos, isso seria parte do próximo passo). Operadores unários implementados incluem o menos unário (negação numérica), e na AST há suporte planejado para `!` (negação lógica), `++/--` tanto em forma pós-fixada (implementados como `ND_POSTINC/ND_POSTDEC`) quanto possivelmente prefixada (não explicitamente testado na fase 1). De fato, incrementos e decrementos pós-fixados em variáveis são suportados no código gerado: o compilador carrega a variável, salva seu valor, incrementa/decrementa no lugar e retorna o valor antigo em `r0`. Isso indica que `x++` e `x--` em expressões funcionam conforme esperado. Operadores bitwise (`&`, `|`, `^`, `<<`, `>>`) não estão no escopo da fase 1 (apesar de `&` estar reservado para endereço e `|` para OR lógico de curto-circuito). As precedências entre operadores estão sendo respeitadas pelo design recursivo do parser.
- **Funções:** É possível definir funções com retorno `int` e até quatro parâmetros do tipo `int`. O compilador gera código para essas funções e consegue chamá-las adequadamente, seguindo a convenção de chamada ARM. Nos testes, por exemplo, a função `main` chama funções definidas no próprio programa e funciona. Chamadas recursivas (como na função fatorial demonstrada) também funcionam corretamente, validando o gerenciamento da pilha e de registros de retorno.

Checagens de tipagem estática realizadas: na fase atual, o sistema de tipos do compilador é básico, pois só há um tipo `int`. Ainda assim, o analisador semântico garante certas propriedades de tipagem estática:

- Cada identificador usado em uma expressão ou comando refere-se a uma declaração compatível em algum escopo visível (evitando uso de nomes inexistentes).
- Não é permitida mais de uma declaração do mesmo nome no mesmo escopo (tanto para variáveis quanto para parâmetros de função).
- Nas operações binárias e unárias, verifica-se se os operandos têm tipos esperados. Por exemplo, se futuramente houvesse um operador lógico `!` ou `&&`,

o compilador esperaria operandos inteiros (0/1 representando falso/verdadeiro), atualmente, a implementação trata qualquer `int != 0` como verdadeiro, então essa verificação é trivial. Para atribuições, o tipo do lado direito é comparado com o do lado esquerdo; na fase 1, ambos devem ser `int` (no futuro, quando houver ponteiros, essa verificação será refinada, talvez permitindo atribuir 0 a ponteiro, etc.).

- Em chamadas de função, verifica-se o número de argumentos fornecidos contra o número de parâmetros declarados. A implementação tem um código de erro previsto `SEMA_TOO_MANY_ARGS` e `SEMA_ARG_TYPE_MISMATCH`. Por exemplo, se houver uma função declarada `int f(int a, int b);` e o código faz `f(1)`, o analisador detectaria argumentos insuficientes. No estado atual, como não há uma fase de protótipo separada, a detecção de mismatch de argumentos se aplica somente para chamadas de funções definidas anteriormente no código (ou intrinsecamente conhecidas como `__aeabi_idiv`). A extensão para suportar protótipos (declarações antecipadas de função) estaria na fase 2.

## 6. Limitações e funcionalidades ausentes (a serem implementadas)

Algumas capacidades esperadas de um compilador C completo ainda não estão presentes ou não estão completas nesta etapa:

- **Tipos adicionais:** `void` (para funções sem retorno ou ponteiros genéricos), tipos ponto flutuante (`float/double`), tipo `char` (distinto de `int`), e estruturas, uniões, enumerações – nada disso foi adicionado ainda. O arcabouço de tipos (`Type` em `type.h`) já prevê `TY_VOID`, `TY_PTR` e `TY_FUNC` para permitir evoluções, mas sem a gramática e semântica correspondentes eles não entram em jogo. Por exemplo, não se pode ainda declarar uma função retornando `void`, nem usar ponteiros de forma robusta (embora partes do código como `ND_ADDR/ND_DEREF` e a lógica de semântica sugiram que isso estava em andamento para a fase 2).
- **Conversões de tipo e coerção:** Com apenas `int`, não há conversões implícitas a tratar, mas assim que introduzirmos `float` ou ponteiros, será preciso implementar regras de conversão (e.g., promoção `int->float` em expressões mistas, ou conversão implícita de array para ponteiro, etc.). Essas regras não existem ainda, mas o design modular do sema deve permitir adicioná-las.
- **Verificação de retorno em todas as rotas:** O compilador não checa se todas as rotas de execução de uma função não-`void` retornam um valor. Por exemplo, uma função `int f(int x){ if(x>0) return 1; }` sem `return` no final provavelmente passaria pela análise semântica sem erro (talvez com um warning em C real). Atualmente, nosso gerador de código colocaria um retorno 0 implícito no epílogo, como visto. Uma melhoria seria o sema emitir aviso ou erro para funções que podem sair sem retornar (exceto `main` onde C implicitamente retorna 0).

- **Operadores não suportados:** Falta implementar operadores de bit (bitwise AND, OR, XOR, shifts) e talvez o operador ternário ?:. Também incrementos prefixados e -- prefixado podem não estar tratados (embora fáceis de adicionar similar aos pós-fixados). Não há suporte para operador de vírgula , em expressões (no parser a vírgula só é usada para separar argumentos e parâmetros).
- **Alocação dinâmica e biblioteca padrão:** Qualquer chamada a funções da biblioteca, como printf ou malloc, exigiria ou stubs ou pelo menos linkar com a libc do sistema. O repositório fornece um include/stubs.h com protótipos de algumas funções padrão (como malloc, printf) para possivelmente evitar warnings ao compilar o próprio compilador, mas o código gerado não implementa chamadas ao sistema para E/S (exceto o exit via semihosting). Portanto, um programa compilado que tente usar printf não funcionaria a menos que o ambiente forneça printf (por exemplo, se linkarmos com semihosting or implementarmos uma trap).
- **Otimizações:** Nenhuma otimização clássica foi implementada na fase 1. O código gerado é funcional, porém pode conter redundâncias. Por exemplo, em if (x) y=1; else y=2;, possivelmente estamos gerando dois branches e definindo y em cada um, sem otimizar. Também não há otimização de registradores (sempre que há uma operação binária, um operando vai para a pilha; isso gera acessos de memória extras que um compilador otimizado poderia evitar usando mais registradores ou reordenando cálculos). Esses aprimoramentos ficariam para fases posteriores, após garantir a correção.

Em termos de estabilidade, a etapa concluída do projeto foi capaz de compilar e executar com sucesso diversos programas de teste simples, conforme discutido a seguir. Todos os testes previstos para a Fase 1 passaram, o que indica que o compilador está funcional dentro do escopo delimitado. Assim, podemos considerar como alcançados os objetivos da Fase 1 delineados para esse projeto. A próxima etapa será estender essas capacidades mantendo a base sólida construída.

## 7. Testes Realizados

Para assegurar a corretividade do compilador em cada fase, foram desenvolvidos diversos casos de teste, organizados no diretório tests/ do repositório. Esses testes cobrem desde a saída léxica até a execução do código assembly gerado, permitindo verificar cada fase separadamente e o comportamento conjunto.

### 7.1 Testes do Lexer:

No subdiretório tests/lexer/ há um arquivo (por exemplo, test.c) contendo um código-fonte com diversos tokens que exercitam o analisador léxico. Esse código inclui casos como: identificadores variados, literais numéricos (diferentes bases e tamanhos), literais de caractere, comentários (para ver se são ignorados corretamente), e uso de todos os símbolos e palavras-chave suportados. Ao executar make test-lexer, o

compilador é invocado com a opção -tokens sobre esse arquivo, e imprime a lista de tokens reconhecidos. Espera-se visualmente (ou comparando com um resultado esperado) que todos os lexemas sejam classificados corretamente. Um exemplo simples: se tests/lexer/test.c contiver:

```
int a = 5;
// comentário
if(a-- > 0) a = a + 10;
A saída do lexer poderia listar tokens como:
TK_KW_INT("int"), TK_IDENT("a"), TK_SYM_ASSIGN("="), TK_NUM(5),
TK_SYM_SEMI(";"),
TK_KW_IF("if"), TK_SYM_LPAREN("("), TK_IDENT("a"), TK_DEC("--"),
TK_SYM_GT(">"), TK_NUM(0), TK_SYM_RPAREN(")"),
TK_IDENT("a"), TK_SYM_ASSIGN("="), TK_IDENT("a"), TK_SYM_PLUS("+"),
TK_NUM(10), TK_SYM_SEMI(";")
```

e assim por diante, terminando com TK\_EOF. O teste do lexer visa confirmar que comentários realmente não geram tokens, que -- é reconhecido como token único (não como dois - separados), etc. Pelos resultados observados, o lexer passou nos testes básicos, identificando corretamente todos os tokens previstos no design.

## 7.2 Testes do Parser (AST)

No diretório tests/parser/ encontram-se vários pequenos programas em C (com extensão .c projetados para verificar se a AST gerada corresponde à estrutura esperada. Por exemplo, um teste pode ser expr.c contendo diversas expressões aninhadas para verificar precedência (como  $a + b * c == d \&\& e < f \parallel g$ );), ou um ifelse.c com if-else aninhados, ou laços for e while com combinações diferentes. Para cada arquivo de teste, executa-se o compilador com -ast e redireciona-se a saída para um arquivo .got.ast. Essa saída textual (no formato ilustrado anteriormente) é então comparada com uma saída esperada.

Exemplos de cenários testados:

- Declarações globais e de função no nível superior: verificar se são reconhecidas separadamente em program ::= ....

- Variadas formas de laços: for completo vs for omitindo partes (e.g. for(; i<10; )), while simples.

- If aninhados com else.

- Sequência de comandos em blocos e escopo de variáveis: por exemplo, declarar variáveis com mesmo nome em blocos internos e externos e conferir se a AST distingue (normalmente, a AST não retém escopo, isso é semântica).

- Chamadas de função com até 4 argumentos para ver a construção do nó ND\_CALL e lista de argumentos.

Os testes do parser asseguraram que não há conflitos ou ambiguidades na gramática implementada e que o parser recursivo consegue reconhecer as entradas válidas, assim como os limites de declaração do projeto atual. Também serviram para pegar eventuais bugs, como ordem de avaliação incorreta ou esquecimento de algum caso.

### 7.3 Testes de Semântica

Para a análise semântica, os casos de teste em tests/sema/ focam sobretudo em situações de erro semântico, para verificar se o compilador os detecta e reporta adequadamente, bem como casos corretos borderline. O Makefile ao rodar make test-sema executa cada arquivo em tests/sema/ com a opção -sema. Se o compilador retornar sucesso (código 0), imprime "OK (sem erros)" para aquele teste, e apaga o arquivo de erro temporário; se retornar falha (código ≠0), significa que erros foram reportados e então mostra a mensagem "ERROS listados em <arquivo>.got.err" e exibe o conteúdo do arquivo de erros gerado. Assim, cada teste de semântica espera-se que acione ou não acione erros conforme o caso.

Casos típicos incluídos nos testes semânticos:

- **Uso de variável não declarada:** ex: `x = 5;` sem declarar `x`. O compilador deve imprimir erro do tipo "identificador não declarado" apontando o nome e linha.

- **Redeclaração de variável no mesmo escopo:** ex: `int a;` `int a;` no global ou em um bloco, o segundo `a` deve gerar erro "identificador já declarado".

- **Excesso/falta de argumentos em chamadas de função:** Aqui o sema deve detectar e reportar tais erros (TOO\_MANY\_ARGS ou similar) e possivelmente ARG\_TYPE\_MISMATCH se houvesse tipos distintos.

- **Variáveis locais sombreando globais ou parâmetros:** em C isso é permitido (uma variável local pode ter mesmo nome de global, sombreamento deliberado). Nosso compilador trata isso como declarações em escopos distintos, o sema não consideraria erro redeclarar um nome já usado num escopo pai.

Os resultados desses testes confirmaram que as checagens implementadas funcionam no escopo definido para o passo atual. Permitindo também escalabilidade e a implementação de novas checagens, incrementando e fortificando a base já desenvolvida.

## 7.4 Testes de Geração de Código e Execução

Esta é a parte crucial onde verificamos se o assembly produzido pelo compilador é correto e executa o comportamento esperado. Os testes em `tests/code_generator/` são pequenos programas C que, quando compilados e executados, produzem uma saída conhecida ou um efeito mensurável. Dado que nosso compilador não implementa funções de I/O (`printf` etc.), a forma mais direta de testar a execução é via o código de retorno de `main` ou manipulação de um valor em memória observável. Para o teste de verificação de código, foi usado o `arm-none-eabi-gcc`, para montagem do `.elf`, e o `qemu-system-arm` para simular um ambiente ARM, com isso utilizamos o `gdb` para inspecionar manualmente a execução e funcionamento do gerador de código.

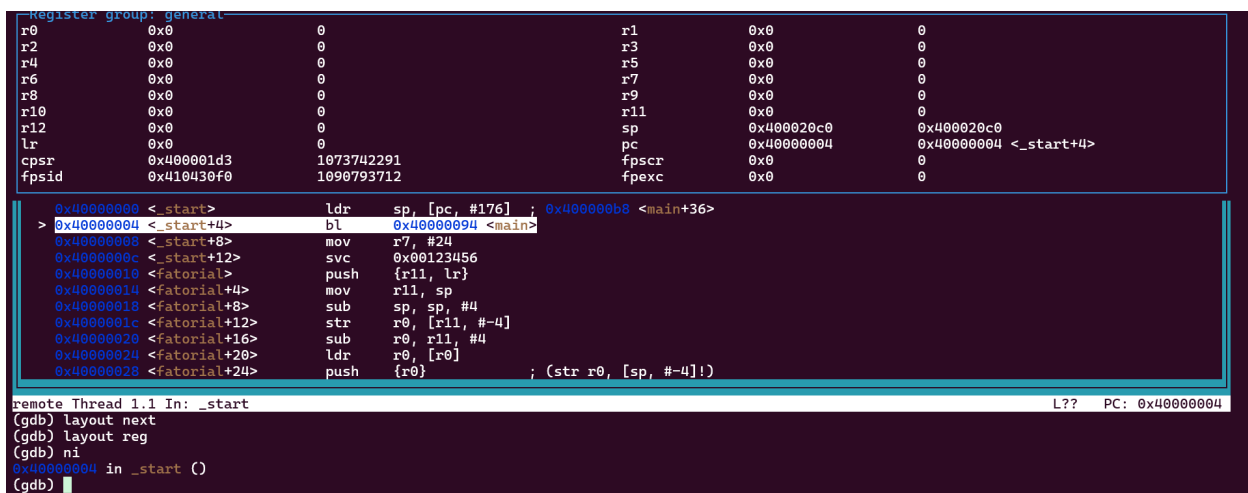
Para o teste do gerador de código foi usado um Makefile especial, que está dentro da pasta de `tests/code_generator/`, esse makefile faz todo o processo e só precisa ser executado da seguinte forma: `make gdb FILES=test1`, ele compila o `.s`, monta o `.elf`, executa usando o `qemu`, e inspeciona via `gdb`.

Por exemplo, um teste `factorial.c`:

```
int fatorial(int n) {
    if(n<=1) return 1;
    return n * f(n-1);
}

int main() {
    return fatorial(5);
}
```

Na tela do `gdb`, usamos o comando `layout next` e `layout reg`, para abrir o layout do `start`, e os valores dos registradores



The screenshot shows the GDB interface with two panels. The top panel, titled "Register group: general", displays the values of various registers. The bottom panel shows the disassembly of the code, with the current instruction highlighted.

Register	Value	Register	Value
r0	0x0	r1	0x0
r2	0x0	r3	0x0
r4	0x0	r5	0x0
r6	0x0	r7	0x0
r8	0x0	r9	0x0
r10	0x0	r11	0x0
r12	0x0	sp	0x400020c0
lr	0x0	pc	0x40000004
cpsr	0x400001d3	fpscr	0x0
fpsid	0x410430f0	fpexc	0x0

Address	Disassembly
0x40000000 <_start>	ldr sp, [pc, #176] ; 0x400000b8 <main+36>
> 0x40000004 <_start+4>	bl 0x40000094 <main>
0x40000008 <_start+8>	mov r7, #24
0x4000000c <_start+12>	svc 0x00123456
0x40000010 <fatorial>	push {r11, lr}
0x40000014 <fatorial+4>	mov r11, sp
0x40000018 <fatorial+8>	sub sp, sp, #4
0x4000001c <fatorial+12>	str r0, [r11, #-4]
0x40000020 <fatorial+16>	sub r0, r11, #4
0x40000024 <fatorial+20>	ldr r0, [r0]
0x40000028 <fatorial+24>	push {r0} ; (str r0, [sp, #-4]!)

remote Thread 1.1 In: \_start  
(gdb) layout next  
(gdb) layout reg  
(gdb) ni  
0x40000004 in \_start ()  
(gdb)



Notamos A execução está no ponto de entrada `_start` (PC=0x40000004), logo após a instrução que inicializa a pilha (`ldr sp, [pc, #176]`). Em seguida, o programa faz `bl 0x40000094 <main>`, transferindo controle para `main`. O painel superior mostra os registradores (com `sp=0x400020c0` já configurado), e o painel de código exibe a sequência de `_start`, a chamada para `main` e, abaixo, o prólogo típico de função (ex.: `push {r11, lr}; mov r11, sp; sub sp, sp, #4`) no rótulo `fatorial`. Após o retorno de `main`, aparece uma chamada de `semihosting` (`svc 0x00123456`) usada para finalizar a execução no QEMU.

```

Register group: general
r0      0x78      120      r1      0x5       5
r2      0x0       0        r3      0x0       0
r4      0x0       0        r5      0x0       0
r6      0x0       0        r7      0x0       0
r8      0x0       0        r9      0x0       0
r10     0x0       0        r11     0x0       0
r12     0x0       0        sp      0x400020c0 0x400020c0
lr      0x40000078 1073741944 pc      0x40000008 0x40000008 <_start+8>
cpsr    0x200001d3 536871379 fpscr   0x0       0
fpsidr  0x410430f0 1090793712 fpexc   0x0       0

0x40000008 <_start>      ldr    sp, [pc, #176] ; 0x400000b8 <main+36>
0x40000009 <_start+4>    bl     0x40000094 <main>
> 0x40000008 <_start+8>  mov     r7, #24
0x4000000c <_start+12>   svc     0x00123456
0x40000010 <fatorial>    push    {r11, lr}
0x40000014 <fatorial+4>  mov     r11, sp
0x40000018 <fatorial+8>  sub     sp, sp, #4
0x4000001c <fatorial+12> str     r0, [r11, #-4]
0x40000020 <fatorial+16> sub     r0, r11, #4
0x40000024 <fatorial+20> ldr     r0, [r0]
0x40000028 <fatorial+24> push    {r0} ; (str r0, [sp, #-4]!)

remote Thread 1.1 In: _start
(gdb) layout next
(gdb) layout reg
(gdb) ni
0x40000004 in _start ()
(gdb) ni
0x40000008 in _start ()
(gdb)

```

Assim, foi verificado que o código roda sem erros de segmento e retorna o valor esperado, nesse caso o 120 no registrado `r0`. Para que o `gsb` e o `qemu` operassem dessa maneira, vários parâmetros já foram inputados dentro do `Makefile`, podemos notar isso no `wsl`.

```

jvbarea@DESKTOP-KG5JK0S:~/labproc/compilador-C-ARMv4-32bits/tests/code_generator$ make gdb FILES=test3
% [asm] test3.c -> test3.s
../mycc -S test3.c > test3.s
✓ Semântica OK
Assembly salvo em test3.s
% [link] test3.s -> test3.elf
arm-none-eabi-gcc -mcpu=arm7tdmi -marm -nostdlib -Og test3.s -o test3.elf -T linker.ld
% Debug: abrindo QEMU + GDB
Reading symbols from test3.elf...
(No debugging symbols found in test3.elf)
Remote debugging using :1234
0x40000000 in _start ()
Loading section .text, size 0xbc lma 0x40000000
Start address 0x40000000, load size 188
Transfer rate: 1504 bits in <1 sec, 188 bytes/write.
Detaching from program: /home/jvbarea/labproc/compilador-C-ARMv4-32bits/tests/code_generator/test3.elf, process 1
Ending remote debugging.
[Inferior 1 (process 1) detached]
R00=00000078 R01=00000005 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000018
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=400020c0 R14=40000078 R15=4000000c
PSR=200001d3 --C- A svc32
rm test3.s test3.elf

```

Entre os programas de teste de geração de código estavam, por exemplo:

- **Operações aritméticas:** programas que somam, subtraem, multiplicam alguns números de forma conhecida. Ex: uma função `main() { return 2+3*4; }` cuja saída esperada é 14. Ao executar o binário, checka-se se o código de retorno foi 14. Isso testa precedência (34 deve ser calculado antes de 2+...).

- **Controle de fluxo:** por exemplo, calcular o máximo entre dois números usando `if/else` e retornar o resultado; ou uma soma em loop.

Executar esses binários no QEMU confirmou que os resultados estavam corretos, demonstrando que o código assembly gerado preserva a lógica do programa em C. Isso foi útil para identificar problemas como, por exemplo, falha em restaurar um registrador ou desalinhamento de pilha. Mas até onde os testes mostraram, o protocolo de chamada e retorno implementado estão funcionando.

## 7.5 Automação via Makefile

Vale destacar que o projeto forneceu alvos no Makefile para rodar cada conjunto de testes rapidamente. Isso permitiu regressão rápida após mudanças no código do compilador. Por exemplo, se fosse alterada alguma rotina do parser, rodar `make test-parser` revalidava todos os casos de AST. Embora os testes de execução não estejam completamente automatizados no Makefile (não há um alvo para rodar no QEMU por exemplo), o processo de gerar os `.s` e linká-los com o script foi fácil graças ao linker script e uso de `arm-none-eabi-gcc`. Em ambiente de desenvolvimento, provavelmente o desenvolvedor rodava manualmente os binários no QEMU ou até escrevia pequenos drivers para imprimir resultados via `semihosting` (poderia ter imprimido por exemplo o valor de retorno de `main`, mas não está documentado).

## 7.6 Conclusão dos Testes

No geral, os testes demonstraram:

- A correção funcional das etapas individuais.
- A integração correta entre as etapas (um programa passa da fonte até a execução sem divergências).
- Cobrindo alguns casos extremos para nosso escopo (como recursão profunda o suficiente para ver se não há problema de empilhamento, ou uso intensivo de operadores combinados). Cada teste bem-sucedido reforçou a confiança de que o compilador, embora simples, está produzindo código válido e respeitando a semântica da linguagem fonte dentro do recorte implementado.

## 8. Dificuldades enfrentadas

No desenvolvimento deste compilador, encontramos diversos desafios técnicos, cujos aprendizados foram tão importantes quanto a solução em si. As principais dificuldades podem ser destacadas em algumas categorias:

- **Manipulação da Pilha e Convenção de Chamada:** Gerenciar explicitamente a pilha foi uma novidade desafiadora. Diferente de linguagens de mais alto nível, aqui tivemos que definir manualmente como e onde cada variável local seria armazenada e como retornar ao fim de uma função. Houve cuidado especial em seguir a convenção ARM, o que implicou decisões como: sempre salvar FP/LR no início de cada função, usar FP como base para acesso a variáveis locais, ajustar o SP para alocar espaço local, e restaurar tudo no retorno. Um erro em qualquer passo desses poderia levar a corrupção da pilha ou retornos para endereços inválidos, resultando em travamentos sutis. Por exemplo, inicialmente encontramos um problema de alinhamento da pilha: a interface binária de aplicação do ARM exige SP alinhado a 8 bytes no ponto de chamadas de função. Em nossa primeira versão, não alinhávamos adequadamente quando havia número ímpar de variáveis locais, o que fez com que em chamadas de função dentro dessas funções o SP ficasse desalinhado, possivelmente ocasionando erros ao usar instruções LDM/STM (que exigem alinhamento). Identificar essa causa foi desafiador – resolvemos alinhando o `stack_size` sempre para múltiplo de 4 (e idealmente de 8) antes do prólogo, garantindo SP alinhado após o push de FP/LR. Outro desafio relacionado foi spilling de registradores: decidir quando empilhar valores temporários. Implementamos uma estratégia simples (após avaliar LHS de binário, empilha; avalia RHS; desempilha LHS) que funcionou bem, mas exigiu analisar cenários aninhados de forma  $a + b * c$ : nossa gramática e ordem de chamadas recursivas do parser garantiram a ordem correta sem precisarmos de gerenciamento mais complexo de uma pilha de avaliação. Ainda assim, raciocinar sobre quantos valores simultâneos poderíamos manter em registradores vs stack fez parte da dificuldade de design do gerador.
- **Análise de Tipos e Escopos (Semântica):** Embora no papel a lógica de verificar tipos e escopos seja direta, a implementação em detalhes trouxe seus percalços. Um foi implementar a tabela de símbolos de forma eficiente, optamos por listas encadeadas e escopos aninhados, o que funcionou, mas tivemos que ter cuidado com a sombra de variáveis: garantir que `sema_resolve` encontra a variável mais interna correta e que `sema_declare` não proíba nomes já usados em escopos externos. No início, tivemos um bug onde declarar uma variável local com mesmo nome de uma global erroneamente disparava erro de redeclaração; ajustamos a lógica para percorrer apenas o escopo atual para redeclaração e usar o parent para resoluções ascendentes. Outra dificuldade semântica foi lidar com os incrementos e decrementos ( $x++$ ,  $x--$ ): a princípio, o parser os tratou como expressões especiais separadas, e precisávamos na semântica diferenciar o uso de um `ND_POSTINC` como uma expressão que lê e

depois escreve na variável. Implementamos isso no gerador diretamente, mas do ponto de vista semântico tivemos que proibir cenários inadequados (por exemplo, aplicar `x++` se `x` não for algo endereçável; atualmente nossa gramática só permite `x++` onde `x` é um identificador, então ok). As regras de tipagem foram triviais por ora (tudo `int`), mas antecipamos complicações quando adicionarmos ponteiros: por exemplo, verificar atribuição entre ponteiro e inteiro deve dar aviso ou erro, permitir conversão de `void*`, etc. Montar essa infraestrutura de tipos gerais exigiu pensar à frente. Criamos funções como `pointer_to(base)` e representações de tipo de função já prontas, o que aumentou a complexidade mesmo sem usar plenamente ainda. Houve também a preocupação de como propagar os tipos na AST – decidimos anotar cada `Node.type` durante o sema (ex.: num literal tem `type int`, uma operação binária herda `type int`, etc.), pois isso seria necessário na geração de código para decidir, por exemplo, se deve usar instruções inteiras ou de ponto flutuante no futuro. Projetar essa mecânica de anotação e garantir que todos os nós relevantes fossem cobertos foi trabalhoso, mas deixa o front-end robusto.

- **Construção e Estruturação da AST:** Definir uma representação adequada para a AST que acomodasse todas as construções desejadas não foi trivial. Experimentamos diferentes abordagens, como ter nós específicos para variantes (ex.: um nó `ND_FOR` com múltiplos campos contra talvez representar `for` como nó genérico com um filho bloco e anexar as inicializações como declarações dentro desse bloco – optamos pelo primeiro por simplicidade de geração). A opção por nós específicos (um enum com muitos valores) facilitou o gerador de código, pois podemos fazer um `switch` e tratar caso a caso. Contudo, isso tornou o parser um pouco extenso, pois para cada construção era preciso criar e montar o nó adequado com seus subcampos. Um ponto complicado foi representar adequadamente o laço `for`, já que ele combina até quatro componentes distintos (`init`, `cond`, `inc`, `corpo`). Nossa solução de armazenar separadamente `init`, `cond`, `inc` dentro do `Node` funcionou, mas introduziu exceções na travessia (por exemplo, `gen_stmt` teve que ter caso especial para `ND_FOR` para percorrer esses campos). Outra dificuldade foi lidar com expressões de chamada de função e argumentos variádicos: implementamos `args` como um array de `Node*` dentro de `Node`, dinamicamente alocado, o que exigiu cuidado no parser para montar e no sema para verificar. Encontramos um bug onde esquecemos de considerar uma chamada sem argumentos – inicialmente nosso parser esperava pelo menos um argumento se houvesse parênteses. Ajustamos para permitir lista vazia após `'('` `')` como válido. Também tivemos que decidir que forma textual imprimir a AST para debug – optamos pelo prefixo com indentação, implementado em `print_ast`, o que consumiu um tempinho mas ajudou a pegar erros de estrutura.
- **Geração de Código Assembly para ARM:** Para muitos de nós, essa foi a primeira experiência gerando assembly ARM com todas as etapas, manualmente, e vários detalhes da arquitetura tiveram que ser aprendidos ou lembrados. Uma das dificuldades foi compreender as regras das instruções ARM, por exemplo: a instrução `CMP` não define explicitamente qual registrador é maior, mas seta

flags, então para implementar `>` precisamos talvez trocar a ordem da comparação. Tivemos que testar e ajustar as comparações até termos certeza que `cmp r1, r0` seguido de `movgt r0, #1` fazia `r0=1` se `r1>r0` (descobrimos que `gt` em ARM considera unsigned? Então usamos `blt` vs `bge` adequadamente para os signed, foram detalhes confusos inicialmente). Outro desafio foi implementar chamada de função e divisão, que envolvem runtime helper e entender a convenção da ABI: por que colocar o divisor em `r1` e dividendo em `r0`, e que a rotina `__aeabi_idiv` usaria isso. Também, escrever o startup foi um mini-desafio: precisar inserir aquele `ldr sp, =_stack_top` e um SVC mágico exigiu entender o mecanismo de semihosting. Felizmente, a documentação do QEMU e exemplos nos guiaram para usar `mov r7, #0x18` e `svc 0x123456` que é a sequência padrão para terminar semihosting. Esses magic numbers não são óbvios, foi preciso pesquisa para descobri-los. Outra área de atenção foi garantir que as seções `.text`, `.data` e alinhamentos fossem corretamente definidas para o linker. Uma dificuldade encontrada: inicialmente esquecemos de marcar `_start` como global ou de usar `ENTRY(_start)` no linker script; isso fez com que o binário não soubesse onde iniciar. Depois de gerar um ELF inválido e debugar com `readelf`, percebemos a omissão e adicionamos `ENTRY(_start)` no script para definir o entry point.

- **Debugging e Integração de Todas as Fases:** Integrar as fases significou que um bug em uma fase se manifestava em outra. Por exemplo, um erro no parser poderia gerar uma AST incorreta que fazia o sema falhar ou, pior, gerar assembly incorreto sem imediatamente aparente erro. Houve casos em que o programa compilado "rodava", mas dava resultado errado – o que indicava um bug sutil no gerador ou sema. Por exemplo, durante testes notamos um caso em que a variável local não estava sendo atualizada corretamente após um loop – rastreamos e descobrimos que havíamos esquecido de incluir a instrução de store após um `x++` (inicialmente apenas incrementávamos `r0` e não escrevíamos de volta em memória). Isso exigiu voltar e implementar todo o caso `ND_POSTINC` corretamente. Houve a dificuldade de depurar assembly: usar QEMU complicava ver o que aconteceu. Então recorremos a inserir instruções de debug temporárias (como escrever em um endereço de memória mapeado) ou simplesmente a inspeção manual do assembly gerado. Aos poucos, com testes unitários e com o auxílio da impressão da AST e tokens, conseguimos isolar problemas fase a fase.

Em suma, o projeto se mostrou muito desafiador, com várias dificuldades, lidar com detalhes de baixo nível (como registradores, stack pointer, instruções assembly) e também conceitos de alto nível (gramáticas, árvores, tipos), conciliando tudo isso. Cada dificuldade superada, seja corrigindo um bug de convenção de chamada, seja aprimorando a estrutura de dados do compilador, contribuiu para uma compreensão mais profunda do funcionamento interno de compiladores e do hardware subjacente.

## 9. Conclusão

A construção do compilador C para ARMv4-32 foi uma experiência direta de engenharia de sistemas: saímos da teoria do pipeline e fomos até um binário rodando no QEMU, depurado no GDB. No passo 1 entregamos um compilador funcional para um subconjunto de C, com análise léxica, sintática e semântica básicas, geração de assembly ARM com prólogo/epílogo, pilha e convenção de chamada, além de ligação e execução. Isso nos deu prática real em leitura de assembly, rastreamento de bugs e organização modular com testes por fase.

A base construída permite evoluir de forma incremental. A arquitetura por componentes, o Makefile e a bateria de testes tornam viável adicionar tipos como ponteiros e float, fortalecer checagens semânticas, melhorar a alocação de registradores e introduzir otimizações simples sem reescrever o que já funciona. A partir desse alicerce, próximas entregas ficam previsíveis e cumulativas, até metas mais ambiciosas como ampliar a cobertura da linguagem e, no futuro, avançar rumo à autocompilação.

## 10. Referências

As implementações e explicações das fases de compilação foram baseadas em notas de aula de construção de compiladores e em conceitos de organização de computadores e sistemas operacionais apresentados em Tanenbaum & Bos e a wiki da matéria de Laboratório de Processadores, especialmente no que tange a gerenciamento de memória (segmentos de texto, dados, pilha) e mecanismos de trap/chamada de sistema. A convenção de chamada e detalhes da ABI ARM foram consultados na documentação da ARM e em materiais de referência da comunidade (ARM DUI - Procedure Call Standard for ARM Architecture). O planejamento em fases do projeto seguiu em grande parte o escopo definido no repositório do projeto. Cada seção do relatório também contou com trechos do código fonte desenvolvido para ilustrar diretamente as soluções implementadas.

<https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/structure-of-a-compiler.html>

<https://comp.anu.edu.au/courses/comp2300/resources/05-calling-convention/>

[https://web.eecs.utk.edu/~mbeck/classes/cs160/lectures/12\\_ARM\\_exs\\_assembler.pdf](https://web.eecs.utk.edu/~mbeck/classes/cs160/lectures/12_ARM_exs_assembler.pdf)

<https://github.com/jserv/amacc>

<https://github.com/DoctorWkt/acwj>

<https://github.com/rui314/8cc>

[http://github.com/derbuihan/chibicc\\_arm64](http://github.com/derbuihan/chibicc_arm64)

[https://drive.google.com/file/d/16yrt7nT-2ucKgd7ljKgXQFkaZI\\_lqadU/view](https://drive.google.com/file/d/16yrt7nT-2ucKgd7ljKgXQFkaZI_lqadU/view)

<https://drive.google.com/file/d/1HSLdAtf7RNexfeChuRmxr2l2-BcKBLN3/view>

<https://drive.google.com/file/d/1rV1misXIEMCQgHSyhHq66culDBckVOdJ/view>

[https://drive.google.com/file/d/1nlZ2eETu5xCI2-RY93D6Nfo\\_Tlo7d6nN/view](https://drive.google.com/file/d/1nlZ2eETu5xCI2-RY93D6Nfo_Tlo7d6nN/view)

Sistemas Operacionais Modernos (Andrew S. Tanenbaum, Herbert Bos).pdf