CENTRO UNIVERSITÁRIO UNIFTEC

Relatório Técnico - Analisador de Código em C

Eduardo Staudt de Castilhos

Pedro Henrique Busnello Frizon

Professor Juliano Paulo Menzen

Sumário

1 INTRODUÇÃO 3
2 METODOLOGIA DE ANÁLISE 4
3 ESTRUTURAS DE DADOS UTILILIZADAS 5
3.1 TOKEN 5
3.2 INFOLOOP 5
3.3 INFORECURSAO 5
3.4 ANALISEFUNCAO 5
4 ALGORTIMOS IMPLEMENTADOS 6
4.1 ANÁLISE DE TOKENS 6
4.2 DETECÇÃO DE FUNÇÃO 6
4.3 DETECÇÃO DE LOOPS 6
4.4 DETECÇÃO DE RECURSÃO 6
4.5 DETECÇÃO DE CASO BASE 6
4.6 CÁLCULO DA COMPLEXIDADE 6
5 CASOS DE TESTE 7
6 LIMITAÇÕES IDENTIFICADAS 8
7 CONCLUSÃO 9

1 Introdução

Este relatório apresenta o funcionamento de um programa em C que desenvolvemos para analisar automaticamente arquivo **.c**. A ideia principal é ler o código e identificar alguns estruturas que são importantes, como:

- **Funções**
- **Loops (for e while)**
- **Recursão**
- **Funções recursivas**
- **Complexidade de algoritmo**

No final o programa gera um relatório com tidas as informações dentro do prompt de comando e também o mesmo será salvo em um formato .txt dentro da pasta origem.

2 Metodologia de Análise

Para fazer a análise do código, utilizamos uma abordagem estática, onde não executamos o programa que está sendo analisado. Apenas é lido linha a linha procurando pelos padrões. Esse processo tem como base 4 etapas:

- **Leitura por linha**
  - Ocorre a limpeza de espaços e é identificado palavras-chaves importantes.
- **Detecção de funções**
  - Quando encontramos uma linha que aparenta ser uma função, todas linhas dentro desse bloco da função são registradas.
- Análise do conteúdo da função
  - Dentro das linhas dessa função é procurado por loops, por funções recursivas, por casos base e também é verificado a profundidade dos blocos.
- Cálculo da complexidade
  - Com bases nos loops e tipo de recursão, estimamos a complexidade da função.

No final, todas essas funções são salvas no .txt para análise.

3 Estruturas de Dados Utililizadas

Criamos algumas structs para ajudar na organização dos dados.

3.1 Token

Guarda uma parte da linha que pode ser palavra, número, símbolo ou operador.

3.2 InfoLoop

Armazena tudo sobre cada loop encontrado:

- tipo (for ou while)
- linha onde aparece
- profundidade do aninhamento
- variável usada no loop (apenas quando conseguir identificar)
- se o loop depende do tamanho de entrada (ex: n)

3.3 InfoRecursao

Guarda dados sobre recursão

- quantidade de chamadas recursivas
- linhas onde elas aparecem
- sinalização de caso base
- linha do caso base

3.4 AnaliseFuncao

Essa é a struct mais importante. Ela guarda:

- nome da função
- linha inicial e final
- lista de loops
- profundidade máxima
- informações de recursão
- complexidade estimada

4 Algortimos Implementados

4.1 Análise de tokens

O programa percorre a linha e identifica números, palavras, símbolos e operadores. Não é um analisador léxico completo, mas funciona para o objetivo.

4.2 Detecção de função

A função **eh_declaracao_funcao()** usa algumas verificações simples: precisa ter (e) não pode estar dentro de um if, for ou while não pode ser comentário pega o último identificador antes do parêntese como nome da função.

4.3 Detecção de loops

A função **detectar_loop()** verifica a presença de "for" ou "while" e faz algumas análises sobre o loop: tipo do loop linha onde está profundidade (com base no nível de {}) se a condição parece depender do tamanho de entrada, usando palavras como n, size e length.

4.4 Detecção de recursão

A função verifica se dentro do corpo da função aparece uma chamada dela mesma. Se sim, ela registra: quantidade de chamadas linhas onde elas aparecem.

4.5 Detecção de caso base

Procuramos por return dentro de condicionais. Isso normalmente caracteriza um caso base em funções recursivas.

4.6 Cálculo da complexidade

Baseamos a lógica em regras simples:

- Recursão com mais de uma chamada → O(2^n)
- Recursão + loops → O(n log n)
- Recursão simples → O(n)
- Somente loops → depende da profundidade:

Profundidade	Complexidade
0	O(1)
1	O(n)
2	$O(n^2)$
3	$O(n^3)$
> 3	O(n^k)

5 Casos de Teste

Para testarmos que o validador funciona, criamos 6 casos de teste.

- Bubble sort
- Busca binária
- Busca linear
- Fibonacci
- Loops
- Merge

Para executar os casos de teste acima na sua máquina é preciso que o .exe do arquivo seja gerado, para isso abra o cmd ou git bash por exemplo, navegue até a pasta onde está o código em **.c** e digite o seguinte código

gcc -o analisador analise.c

Esse código acima irá gerar um arquivo **.exe** com o nome **analisador** (**analisador.exe**) do arquivo **analise.c**.

Após isso, basta executar esse programa para cada caso de teste da pasta.

Pelo cmd:

analisador.exe ct_fibonacci.c

Pelo git bash

./analisador.exe ct_fibonacci.c

E isso irá gerar os resultados tanto no visual quanto em um arquivo .txt na pasta.

6 Limitações Identificadas

Mesmo que o código tenha funcionado bem para os casos de teste escritos, ainda ficam algumas limitações, como loops dentro de um comentário, variáveis que contenham for ou while no meio, também o calculo da complexidade é bem básico e pode acontecer de alguns casos não serem identificados.

7 Conclusão

O programa desenvolvido cumpre bem o que foi solicitado, sendo capaz de identificar funções, loops, recursão e analisar a complexidade do código. Durante o desenvolvimento, deu para perceber que, quando trabalhamos com códigos pequenos e bem estruturados, como é o caso dessa função Fibonacci, o validador funciona muito bem e consegue identificar corretamente a estrutura. Esse tipo de código mais direto ajuda bastante na validação, porque segue exatamente o padrão esperado.