UNIVERSIDADE FEDERAL DE JATAÍ (UFJ)
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS (ICET)
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Gabriel Krishna de Assis Vieira

Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica: Manipulação de Strings

Jataí-Goiás

2024

Gabriel Krishna de Assis Vieira

Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica: Manipulação de Strings

Monografia apresentada ao curso de Ciência da Computação do Instituto de Ciências Exatas e Tecnológicas da Universidade Federal de Jataí (UFJ), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador(a):Prof. Dr. Thiago Borges de Oliveira

Jataí-Goiás

2024

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFJ.

Vieira, Gabriel Krishna de Assis
Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica : Manipulação de Strings / Gabriel Krishna de Assis Vieira. - 2024.
57 f.: il.

Orientador: Prof. Dr. Thiago Borges de Oliveira.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Jataí, Instituto de Ciências Exatas e Tecnológicas, Ciência da Computação, Jataí, 2024.
Bibliografia. Anexos.
Inclui siglas, abreviaturas, tabelas, lista de figuras, lista de tabelas.

1. Robcmp. 2. Manipulação de cadeias de caracteres. 3. Linguagem de programação. 4. Compiladores. 5. Microcontroladores. I. Oliveira, Thiago Borges de, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE JATAÍ
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

DECLARAÇÃO DE APROVAÇÃO DA VERSÃO FINAL

Declaro que o(a) discente Gabriel Krishna de Assis Vieira do curso de Bacharelado em Ciência da Computação foi aprovado(a) na defesa do Trabalho de Conclusão de Curso (TCC) com o título final Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica: Manipulação de Strings na data de 12/12/2024 e efetuou todas as correções pertinentes sugeridas pela banca examinadora, composta pelo seguintes membros:

Orientador(a)	Thiago Borges de Oliveira
Membro 1	Ariadne de Andrade Costa
Membro 2	Joslaine Cristina Jeske de Freitas

Declaro ainda que a versão final anexada a este processo está adequada para ser devidamente depositada em repositório institucional.

Observação

Esta declaração deve ser assinada pelo(a) orientador(a)

Documento assinado eletronicamente por **THIAGO BORGES DE OLIVEIRA**, **Professor do Magistério Superior**, em 18/12/2024, às 19:52, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufj.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0372167** e o código CRC **A46A2F7D**.

Dedico este trabalho a minha mãe, meu pai e minha irmã, pilares fundamentais na minha vida, cujo amor e apoio me impulsionaram até aqui. Dedico também à minha namorada, Isadora, por estar ao meu lado em cada passo desta jornada, com sua paciência, incentivo e carinho que me ajudaram a superar os momentos mais difíceis. A vocês, minha família, minha eterna gratidão e amor.

Agradecimentos

Agradeço primeiramente à minha mãe, meu pai, minha irmã e toda a minha família pelo apoio incondicional em todos os momentos e por sempre acreditarem no meu potencial. Um agradecimento especial à minha namorada, Isadora, cujo companheirismo e palavras de incentivo foram fundamentais para enfrentar os desafios deste trabalho, especialmente nos momentos mais difíceis. Ao meu orientador e professor, Thiago Borges, pela paciência, dedicação e pelas valiosas orientações que foram essenciais para o desenvolvimento deste trabalho. Aos meus amigos Hugo, Nicolas, João e Pedro, bem como a todos os colegas de curso, agradeço pelas boas experiências vividas ao longo desta jornada acadêmica, que tornaram o percurso mais leve e enriquecedor. Por fim, sou grato à instituição de ensino e aos professores, que contribuíram, direta ou indiretamente, para a construção do conhecimento que hoje carrego comigo.

"Nada é difícil se for dividido em pequenas partes."
(Henry Ford)

Resumo

Este trabalho apresenta o desenvolvimento de funções de manipulação de cadeias de caracteres na linguagem Robcmp, uma linguagem de programação projetada para microcontroladores no contexto da robótica e sistemas embarcados. A pesquisa teve como objetivos analisar abordagens existentes, projetar uma solução eficiente para representação e manipulação de strings em UTF-8, e implementar cinco funções essenciais: containsstr, lenstr, equalstr, strtouppercase, e strtolowercase. As funções foram validadas por meio de testes comparativos com implementações equivalentes em C, demonstrando eficiência no tamanho do código gerado, embora com um tempo de compilação ligeiramente superior. Os resultados contribuem para o avanço do Robcmp, garantindo maior expressividade e suporte para diferentes contextos linguísticos. Futuras expansões incluem a implementação de novas operações e a otimização adicional do compilador.

Palavras-chaves: *Robcmp; manipulação de cadeias de caracteres; linguagem de programação; compiladores; microcontroladores.*

Abstract

This work presents the development of string manipulation functions for the Robcmp programming language, designed for microcontrollers in the context of robotics and embedded systems. The research aimed to analyze existing approaches, design an efficient solution for UTF-8 string representation and manipulation, and implement five key functions: containsstr, lenstr, equalstr, strtouppercase, and strtolowercase. The functions were validated through comparative testing with equivalent C implementations, showing efficiency in the generated code size, albeit with slightly higher compilation times. The results contribute to advancing Robcmp, ensuring greater expressiveness and support for diverse linguistic contexts. Future expansions include implementing new operations and further compiler optimization.

Key-words: *Robcmp; string manipulation; programming language; compilers; microcontrollers.*

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

LLVM	Low Level Virtual Machine
ARM	Advanced RISC Machine
AVR	Alf and Vegard's RISC processor
UFJ	Universidade Federal de Jataí
ASCII	American Standard Code for Information Interchange
UTF-8	Unicode Transformation Format - 8-bit
CSV	Comma-Separated Values
IR	Intermediate Representation
CPU	Central Processing Unit
RAM	Random Access Memory
ROM	Read Only Memory

Sumário

1 Introdução

As *strings*, ou cadeias de caracteres, são uma parte essencial nas linguagens de programação por serem empregadas na representação de texto nos computadores. Conforme Mertz (2003, p. 2), o processamento de texto está presente em protocolos de comunicação da internet, sistemas de software empresarial, arquivos de configuração, de log, de dados em formato CSV, de mensagens de erros, documentação e o próprio código-fonte. Além disso, a manipulação de cadeias de caracteres permite aos programadores realizarem uma variedade de operações que podem ser necessárias em diversas situações, como combinar cadeias, dividi-las em partes, extrair partes específicas e convertê-las entre maiúsculas e minúsculas (FOUNDATION, 2024). O conteúdo das cadeias de caracteres também pode ser binário, não se limitando apenas a dados textuais. Essa versatilidade é crucial para diversas aplicações, como o processamento de arquivos binários, transmissão de dados em redes e manipulação de dados criptografados. A capacidade de manipular cadeias também é frequentemente necessária para a interpretação da linguagem natural, o que facilita o processamento de texto, a análise de documentos e a análise de dados.

As linguagens de programação tratam a manipulação de cadeias de caracteres de formas variadas de acordo com o seu nível de suporte a esse tipo de operação. Conforme Cooper e Torczon (2014, p. 315), o nível de suporte para cadeias de caracteres varia em relação ao nível de suporte oferecido pela lingaugem. No nível mais básico, como o da linguagem de programação C, as cadeias são apenas *arrays* de caracteres que podem ser alterados, caractere a caractere, e a manipulação – concatenação, conversão, dentre outros, acontece através de chamadas a funções da biblioteca padrão. No nível de suporte mais elevado, como o da linguagem Python, as cadeias são tratadas como instâncias de objetos que armazenam internamente a cadeia de caracteres e são imutáveis – a alteração sempre gera uma nova instância com o novo conteúdo, e as modificações acontecem a partir de métodos da própria classe (*find, encode, format, dentre outros*).

A representação das cadeias de caracteres na memória do computador varia e impacta no custo computacional, ou complexidade, dos algoritmos de manipulação (COOPER; TORCZON, 2014, pp. 317–318). Um tipo de representação é a cadeia de caracteres terminada em nulo, usada, por exemplo, na linguagem C. Nela, o caractere nulo (código zero) é adicionado no final da sequência de caracteres. Outra representação possível é a cadeia de caracteres com prefixo de comprimento, na qual um prefixo com o tamanho da cadeia é acrescentado antes do conteúdo da mesma, nas primeiras posições da memória. Dentre as operações que possuem a sua complexidade determinada por essa escolha estão a concatenação, a busca, e a modificação de sub-cadeias. Cadeias de caracteres terminadas em nulo, por exemplo, requerem a verificação do caractere nulo para determinar o seu comprimento (quantidade de caracteres), ou seja, é necessário percorrer toda a cadeia

de caracteres para o determinar, o que pode aumentar o tempo de processamento em operações de leitura e escrita. Por outro lado, cadeias de caracteres com prefixo de comprimento aceleram operações que dependem do conhecimento prévio do mesmo, mas podem tornar mais complexa a gestão da memória e as operações que alteram o comprimento (COOPER; TORCZON, 2014, pp. 317–318).

Nas cadeias de caracteres textuais, a representação dos caracteres é feita seguindo uma codificação que determina quais números correspondem a quais letras. Inicialmente, a codificação mais utilizada era o ASCII (STAFF, 1980, p. 1), que utiliza 7 bits para representar 128 caracteres diferentes, incluindo letras, números e símbolos básicos. Embora o ASCII tenha sido amplamente adotado e seja eficiente para textos em inglês, ele é insuficiente para representar a diversidade de caracteres usados em outros idiomas e símbolos técnicos. Para superar essas limitações, nas últimas décadas foi adotado o padrão UTF-8, que é uma codificação de comprimento variável capaz de representar todos os caracteres do conjunto Unicode (BECKER; COLLINS; DAVIS, 2011, pp. 27–29). O UTF-8 usa de um a quatro *bytes* para codificar caracteres, mantendo a compatibilidade com ASCII para os primeiros 128 caracteres enquanto permite a inclusão de milhares de outros caracteres. Esta flexibilidade e compatibilidade tornaram o UTF-8 a escolha preferida para a codificação de caracteres em aplicações modernas, facilitando a internacionalização e o suporte a múltiplos idiomas. No entanto, a maior possibilidade de representação de símbolos causa um maior uso de memória e introduz dificuldades para a contagem da quantidade de caracteres devido a representação de tamanho variável.

A escolha destes aspectos (nível de suporte, representação e codificação) ocorre durante o projeto da linguagem e raramente muda com o passar do tempo. Frequentemente, tais aspectos são determinados com base no fato de a linguagem ser de propósito geral, ou seja, não atrelada a nenhum domínio específico de utilização. Ou seja, projeta-se uma solução de compromisso geral, de forma a atender uma ampla gama de possibilidades de utilização. Esse é o caso, por exemplo, das linguagens C, C++ e Python (especificamente a implementação MicroPython), que são frequentemente usadas para a criação de *firmware* para microcontroladores. Por um lado, C e C++ são valorizadas por seu controle preciso do hardware e eficiência, enquanto MicroPython oferece uma sintaxe simples e rápida para prototipagem. Ambas variam no nível de suporte nativo a cadeias de caracteres, forma de representação e codificações possíveis.

Nesse sentido, o projeto Robotics Language: Uma Linguagem de Programação de Propósito Específico para Microcontroladores, PI05974-2024 – e seu predecessor, Especificação e Construção de Protótipos Funcionais de Kits Robóticos de Baixo Custo para uso em Processos de Ensino-Aprendizagem (PI02361-2018) – atuam no desenvolvimento de uma linguagem específica para o domínio da robótica e de microcontroladores, denominada The Robotics Language, e de seu compilador, Robcmp.

O objetivo da Robotics Language é isolar as especificidades dos microcontroladores, provendo uma camada de abstração de hardware dentro do próprio compilador e em sua biblioteca padrão, ao invés de depender de *frameworks* ou bibliotecas que implementem tal camada. Desta forma, os desenvolvedores podem escrever uma só vez o código de suas aplicações (*firmware* para projetos de automação, equipamentos de tecnologia embarcada, dentre outros) sem a necessidade de ter, dentro da aplicação, partes condicionadas e específicas para os equipamentos que a aplicação suporta. Disponibilizando palavras reservadas e mecanismos específicos deste domínio, a análise semântica do compilador consegue encontrar erros comuns de programação que só podem ser encontrados em tempo de execução se as mesmas aplicações fossem desenvolvidas em linguagens de propósito geral, durante o teste da aplicação já no *hardware* embarcado (SUBHI, 2019).

Atualmente, a Robotics Language oferece suporte a *array* de *bytes*, mas ainda não inclui funcionalidades específicas para manipulação e representação de cadeias de caracteres. Portanto, o objetivo desse trabalho é projetar e implementar esse suporte na biblioteca padrão da linguagem, incluindo a forma de representação, a codificação suportada e as funções de manipulação (nível de suporte). Os objetivos específicos são:

- Analisar as diferentes formas de representação, codificação e nível de suporte das cadeias de caracteres nas linguagens de programação usadas em microcontroladores;
- Definir as necessidades específicas para a linguagem Robcmp, conforme o seu domínio específico;
- Projetar a representação, codificação e nível de suporte com base no levantamento de necessidades;
- Implementar no compilador e na biblioteca padrão o suporte projetado; e
- Criar casos de testes para validação da implementação.

2 Referencial Teórico

Este capítulo detalha os conceitos necessários à execução e entendimento do presente projeto. Inicialmente, são explorados os principais componentes de um compilador, incluindo o analisador léxico, analisador sintático, analisador semântico e o gerador de código intermediário, detalhando suas funções e desafios. Em seguida, são discutidos aspectos essenciais da representação e codificação de cadeias de caracteres, focando no controle de memória e no impacto no tempo de execução. Esta análise fornece uma base sólida para o entendimento das técnicas e desafios envolvidos na construção de compiladores e na manipulação eficiente de cadeias de caracteres.

2.1 Compilador

Segundo Cooper e Torczon (2014, p. 1), a função dos computadores em conjunto com o software presente neles vem crescendo a cada ano, e fornecem comunicação, entretenimento, notícias e segurança. Quase todo software é traduzido por uma ferramenta chamada compilador, sendo ele um programa que tem função de traduzir outros programas com intuito de prepará-los para execução.

Os compiladores se destacam como instrumentos cruciais para a transformação da linguagem fonte em uma linguagem alvo, uma etapa indispensável para a execução de software em dispositivos eletrônicos (AHO; ULLMAN, 1995, p. 1). A tradução que acontece possibilita que praticamente todos os usuários de computadores possam ignorar os detalhes da linguagem de máquina, pois a tradução transforma linguagens de programação orientadas para humanos em linguagens de máquinas orientadas para computadores. Nesse sentido, os compiladores permitem que programas sejam portáveis em uma ampla variedade de computadores (FISCHER RON K. CYTRON, 2009, p. 2).

O compilador é separado em duas partes principais, o *frontend* e o *backend* (COOPER; TORCZON, 2014, p. 5). O papel de ambos é bem distinto, com o *frontend* na função de compreender o programa na linguagem-fonte e codificar seu conhecimento em uma estrutura que será utilizada pelo *backend*, posteriormente. A estrutura em questão tem o nome de IR (*Intermediate Representation*), uma representação do código que é processado. O *backend* por sua vez, tem a função de mapear o programa em IR para a máquina-alvo. A implementação do *backend* pode ser feita através de *frameworks*, como é o caso do LLVM(Low-Level Virtual Machine), projetado para suportar a análise e a transformação de programas, fornecendo ao compilador informações detalhadas de alto nível sobre os programas (LATTNER; ADVE, 2004).

Por sua vez, o *frontend* do compilador se divide em etapas distintas, cada uma

desempenhando uma função específica na tradução do código-fonte. A primeira etapa é a análise léxica, seguida da análise sintática e análise semântica, como descrito por Cooper e Torczon (2014, p. 4), e por último o gerador de código intermediário (AHO; ULLMAN, 1995, p. 200). Essas etapas são descritas a seguir.

2.1.1 Analisador Léxico

O papel da análise léxica é traduzir um fluxo de caracteres (código fonte) em um fluxo de *tokens*, onde cada *token* representa uma instância de algum símbolo terminal (FISCHER RON K. CYTRON, 2009, p. 38). O analisador léxico, ou *scanner*, forma palavras agregando caracteres, reconhecendo-as ou não como palavras válidas da linguagem-fonte (COOPER; TORCZON, 2014, p. 19).

Autômatos finitos trazem formalismo para os reconhecedores. Estes possuem um conjunto finito de estados, com estado inicial, um ou mais estados de aceitação, um alfabeto e uma função de transição (COOPER; TORCZON, 2014, p. 23). A linguagem que um autômato finito reconhece é conhecida como linguagem regular. Tal linguagem pode também ser descrita por uma expressão regular, que é uma notação abreviada de um autômato finito. Expressões regulares, apesar de mais simples, são um formalismo equivalente a autômatos finitos (COOPER; TORCZON, 2014, p. 26) e são usadas na construção dos analisadores léxicos dos compiladores modernos.

A Figura 1 ilustra a implementação de um fragmento de autômato que reconhece a palavra ‘new’. Como sugerido por Cooper e Torczon (2014, pp. 21–22), o código testa ‘n’, em seguida ‘e’ e por último ‘w’. A falha ao reconhecer o caractere ocasiona na rejeição e resulta em ‘tente outra coisa’. Frequentemente, o uso de expressões regulares é empregado e reduz a necessidade e tamanho do código de reconhecimento dos terminais da linguagem.

2.1.2 Analisador Sintático

A análise sintática tem como função receber a cadeia de *tokens* provenientes da análise léxica e determinar se a mesma obedece a sintaxe da linguagem. A sintaxe da linguagem, por sua vez, pode ser descrita por uma gramática livre de contexto, que é um conjunto de regras que definem como formar sentenças, sendo essa coleção de sentenças uma linguagem linguagem livre de contexto (COOPER; TORCZON, 2014, pp. 72–73).

O analisador sintático, ou parser, determina se o fluxo de *tokens* forma sentenças válidas dentro da linguagem de programação (COOPER; TORCZON, 2014, p. 69). A Figura 2 ilustra como se dá esse processo. O analisador sintático solicita um *token* ao analisador léxico, que o gera a partir do código fonte. O *token* recebido é avaliado dentro

```
c ← NextChar();
if (c = 'n')
  then begin;
    c ← NextChar();
    if (c = 'e')
      then begin;
        c ← NextChar();
        if (c = 'w')
          then informe sucesso;
          else tente outra coisa;
      end;
      else tente outra coisa;
  end;
  else tente outra coisa;
```

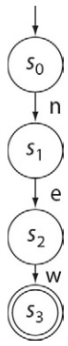


Figura 1 – Fragmento de código para reconhecer 'new' - (COOPER; TORCZON, 2014, pp. 21–22).

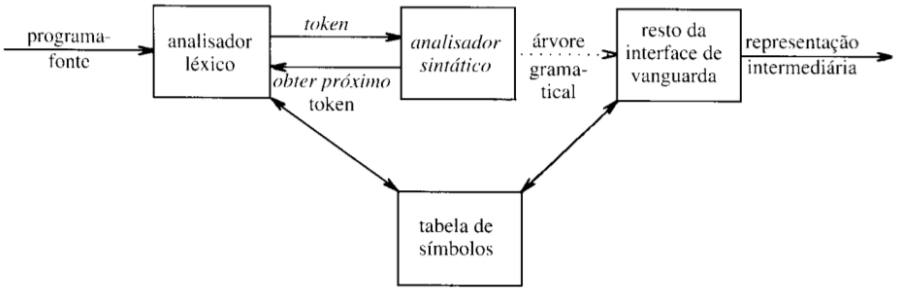


Figura 2 – Posição de um Analisador Sintático num Modelo de Compilador - (AHO; ULLMAN, 1995, p. 72).

das regras sintáticas e eventualmente passa a compor a tabela de símbolos. Quando aceitas, as sentenças passam a compor a árvore gramatical ou árvore sintática. No caso de não formarem, o analisador sintático reporta os erros com possível indicação do motivo (AHO; ULLMAN, 1995, p. 72).

A *abstract syntax tree* (AST) ou árvore de sintaxe abstrata é o principal artefato resultante da análise sintática, sendo ela a estrutura de dados central para todas as atividades subsequentes. A AST é transpassada durante a análise semântica e a geração de código (FISCHER RON K. CYTRON, 2009, p. 250).

A Figura 3 ilustra um exemplo de AST para a expressão $a \times 2 + a \times 2 \times b$. Ela captura a hierarquia e a precedência das operações aritméticas, e inclui apenas os detalhes mais relevantes da sintaxe. Na raiz da árvore temos a operação de adição *(+)*, que possui

dois ramos. O ramo esquerdo representa a multiplicação $a \times 2a$, enquanto o ramo direito representa a multiplicação composta $(a \times 2) \times b$. Isso ilustra que primeiro realizamos as multiplicações, seguindo a precedência, antes de realizar a adição final. Essa precedência é determinada durante a construção da gramática.

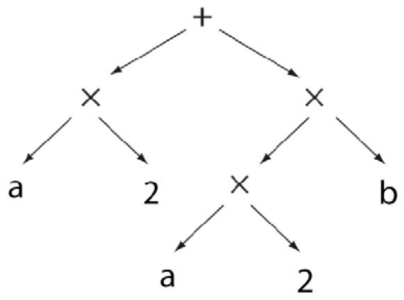


Figura 3 – Árvore sintática abstrata para a expressão $a \times 2 + a \times 2 \times b$ - (COOPER; TORCZON, 2014, p. 197).

2.1.3 Analisador Semântico

A análise semântica verifica se as construções gramaticais fazem sentido lógico dentro do contexto do programa, capturando erros de semântica no programa fonte. A verificação de tipos é uma etapa importante na compilação, e ela acontece na análise semântica. Nela o compilador verifica se os operadores recebem corretamente os operandos permitidos pela especificação da linguagem (AHO; ULLMAN, 1995, p. 4).

O seguinte trecho de código escrito em Linguagem C possui um erro semântico na linha 10. A função *square* é definida para calcular o quadrado de um número inteiro. No entanto, a função *main*, ao chamar *square*, passa um argumento de tipo *double*. A saída do programa é: '*Square of 2.500000 is 4*'. O erro não foi acusado, porém o resultado não foi o esperado.

```
1  #include <stdio.h>
2
3  int square(int num) {
4      return num * num;
5  }
6
7  int main() {
8      int result;
9      double x = 2.5;
10     result = square(x);
11     printf("Square of %f is %d\n", x, result);
12     return 0;
13 }
```

A seguir, o trecho de código demonstra um erro semântico envolvendo cadeias de caracteres. A função *concatenate* concatena a cadeia src ao final da cadeia dest. Embora o código pareça correto, há um defeito semântico. O tamanho do *buffer* str1 não é grande o suficiente para conter a cadeia resultante da concatenação de 'Hello' e 'World', que precisa de pelo menos 11 caracteres (5 de 'Hello', 5 de 'World' e 1 para o terminador nulo). A saída do programa é: '*str1 = HelloWorld*'. Recebemos o resultado esperado, porém a memória adjacente pode ser subscrita, causando potenciais erros no programa.

```c
#include <stdio.h>
#include <string.h>

void concatenate(char *dest, const char *src) {
    while (*dest != '\0') {
        dest++;
    }
    while (*src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
}

int main() {
    char str1[10] = "Hello";
    char str2[] = "World";
    concatenate(str1, str2);
    printf("str1 = %s\n", str1);
    return 0;
}
```

Durante a análise semântica, o compilador verifica a correção dos tipos de dados e a validade das operações definidas na AST, garantindo que as variáveis e operadores são utilizados corretamente, evitando erros de tipo e assegurando o comportamento esperado das expressões.

2.1.4 Gerador de Código Intermediário

A geração do código intermediário vem depois de todas as análises e não é uma etapa presente em todos os compiladores, mas é uma etapa que possui propriedades importantes: ser fácil de traduzir e produzir o programa alvo (AHO; ULLMAN, 1995, p. 7).

No Robcmp, a geração de código intermediário é implementada usando a linguagem

LLVM IR, e isso permite integrar o *frontend* do compilador com o *backend* da suíte LLVM. O *backend*, por sua vez, provê o suporte de tradução de LLVM IR para código alvo em várias plataformas, incluindo os microcontroladores usados nesse projeto. Exemplos são os microcontroladores AVR e ARM (INFRASTRUCTURE, 2024).

A Figura 4 apresenta um fragmento da LLVM IR, em que uma cadeia de caracteres com o texto 'hello world' é definida. Em seguida, é mencionada uma função externa *puts* que imprime texto na tela. A função principal do programa é então definida, que por sua vez chama a função *puts* para imprimir a cadeia previamente definida na tela.

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(ptr nocapture) nounwind

; Definition of main function
define i32 @main() {
  ; Call puts function to write out the string to stdout.
  call i32 @puts(ptr @.str)
  ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

Figura 4 – Fragmento da LLVM IR - (INFRASTRUCTURE, 2024).

2.2 Representação de Cadeias de Caracteres

Exitem diferentes formas de representar cadeias de caracteres em linguagens de programação, impactando no custo computacional, ou complexidade, dos algoritmos de manipulação (COOPER; TORCZON, 2014, pp. 317–318). Um tipo de representação é a cadeia de caracteres terminada em nulo, usada, por exemplo, na linguagem C. Nela, o caractere nulo (código zero) é adicionado no final da sequência de caracteres. Outra representação possível é a cadeia de caracteres com prefixo de comprimento, na qual o mesmo é acrescentado logo antes do conteúdo da mesma, nas primeiras posições da memória.

A escolha da representação de cadeias de caracteres influencia diretamente o desempenho e a eficiência das operações de manipulação, como concatenação, busca, e modificação de sub-cadeias. Cadeias de caracteres terminadas em nulo, por exemplo, requerem a verificação do caractere nulo para determinar o seu comprimento (quantidade de caracteres), ou seja, é necessário percorrer toda a cadeia de caracteres para o determinar, o que pode aumentar o tempo de processamento em operações de leitura e escrita. Por outro lado, cadeias de caracteres com prefixo de comprimento aceleram operações que

dependem do conhecimento prévio do mesmo, mas podem tornar mais complexa a gestão da memória e as operações que alteram o comprimento (COOPER; TORCZON, 2014, pp. 317–318).

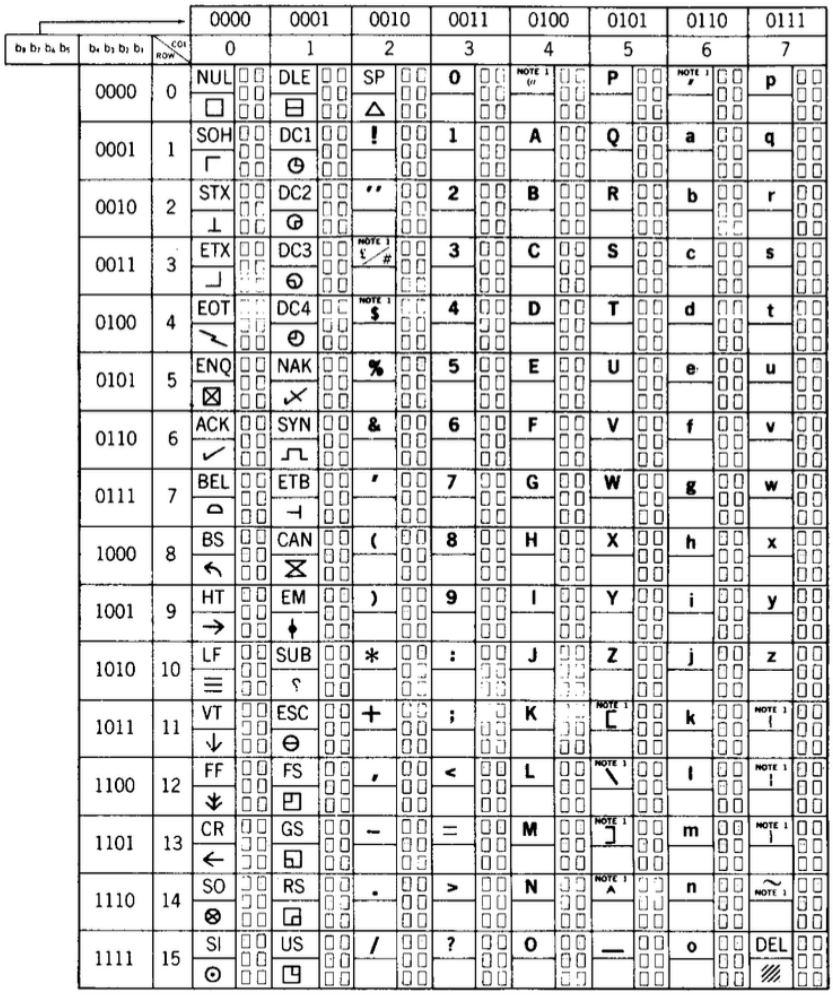
Existem vantagens distintas para cada uma das representações de cadeias de caracteres em linguagens de programação. As cadeias de caracteres terminadas em nulo, oferecem uma gestão simplificada do comprimento, uma vez que basta encontrar o caractere nulo para determinar o fim da cadeia. Isso facilita operações simples como leitura e escrita, especialmente em ambientes onde o tamanho da cadeia não é pré-definido. Por outro lado, as cadeias de caracteres com prefixo de comprimento proporcionam um acesso rápido ao tamanho da cadeia, o que otimiza operações que dependem dessa informação, como concatenação e busca. Essa abordagem pode reduzir o tempo de processamento ao eliminar a necessidade de percorrer toda a cadeia para determinar seu tamanho, em troca de um espaço de memória dedicado.

2.3 Codificação de Cadeias de Caracteres

O ASCII é um tipo de codificação extremamente utilizado ao redor do mundo, principalmente na década de 1980. Foi incorporado em diversos sistemas e equipamentos. A codificação é de 7 bits, por isso consegue representar 128 caracteres, incluindo letras, dígitos, sinais de pontuação e controles básicos (STAFF, 1980, p. 1).

A Figura 5 ilustra a tabela ASCII, com seus 128 caracteres, representados por números de 0 a 127. Os caracteres de 0 a 31, mais o caractere 127, são caracteres de controle, como por exemplo: NUL (*Null*, Código 0); BS (*Backspace*, Código 8); ESC (*Escape*, Código 27); DEL (*Delete*, Código 127). Os caracteres imprimíveis depois, como exemplo: dígitos de ’0’ a ’9’ com códigos 48 a 57; letras maiúsculas de ‘A’ a ‘Z’ com códigos de 65 a 90; letras minúsculas de ‘a’ a ‘z’ com códigos de 97 a 122.

Outra forma de codificação de caracteres é o padrão Unicode, que pode ser representado por várias codificações, sendo uma das mais importantes o UTF-8. O UTF-8 é uma codificação de comprimento variável que usa unidades de 8 *bits* para representar caracteres. O comprimento varia de um a quatro *bytes* para representar cada caractere, permitindo a codificação eficiente de todos os caracteres do Unicode, que é composto pelos caracteres convencionais e ainda abrange escritas do Leste Asiático, como o Chinês, Japonês, Coreano e outras. Uma das principais vantagens do UTF-8 é sua compatibilidade com o ASCII, já que os primeiros 128 caracteres do UTF-8 são idênticos ao ASCII, facilitando a integração com sistemas legados. Além disso, o UTF-8 é amplamente adotado na Web, como no protocolo HTML, devido à sua eficiência e capacidade de representar texto multilíngue de maneira compacta (BECKER; COLLINS; DAVIS, 2011, pp. 27-29).



b4 b3 b2 b1 \ b7 b6 b5	0000	0001	0010	0011	0100	0101	0110	0111
	0	1	2	3	4	5	6	7
0000	NUL	DLE	SP	0	NOTE 1 (@)	P	NOTE 1 (`)	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	NOTE 1 £ #	3	C	S	c	s
0100	EOT	DC4	NOTE 1 $	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	NOTE 1 [	k	NOTE 1 {
1100	FF	FS	,	<	L	NOTE 1 \	l	NOTE 1 |
1101	CR	GS	-	=	M	NOTE 1]	m	NOTE 1 }
1110	SO	RS	.	>	N	NOTE 1 ^	n	NOTE 1 ~
1111	SI	US	/	?	O	_	o	DEL

Figura 5 – Tabela ASCII completa - (STAFF, 1980, p. 2).

A Figura 6 ilustra quatro caracteres codificados em UTF-8, demonstrando a variação do comprimento em *bytes*. O caractere 'A' possui o código 41 em hexadecimal, equivalente ao código decimal 65 do mesmo caractere em ASCII. Caracteres de quatro *bytes* seguem um formato na disposição de *bits*: o primeiro *byte* é '11110xxx', onde o primeiro '1' indica a presença de mais *bytes* além do primeiro. Os seguintes '111' indicam que que há três *bytes* além do primeiro, todos seguindo o formato '10xxxxxx'. Os 'x's representam os *bits* do código Unicode. Em cadeias de caracteres, esse formato permite delimitar os *bytes* em blocos de um a quatro, pois o primeiro *byte* do caractere sempre vai indicar quantos outros *bytes* à frente ainda fazem parte do mesmo bloco.

O caractere de quatro *bytes* apresentado possui o ponto de código Unicode U+10384, e sua representação binária no formato mencionado resulta em quatro *bytes*: '11110000 10010000 10001110 10000100'. Em hexadecimal, esses quatro *bytes* são representados como 'F0 90 8E 84', que é o resultado final da codificação em UTF-8.

Figura 6 – Exemplo de caracteres de 1 a 4 *bytes* em UTF-8 (BECKER; COLLINS; DAVIS, 2011, p. 26).

2.3.1 Controle de Memória

A alocação de memória pode acontecer de formas distintas. De acordo com Aho e Ullman (1995, pp. 173–177), os principais métodos de alocação de memória são a alocação de memória de pilha e a alocação de memória *heap*, métodos esses que possuem vantagens e desvantagens em relação um ao outro. A alocação de memória em pilha tem como característica os registros de ativação serem empilhados e desempilhados a medida que as ativações (chamadas de funções) começam e terminam, respectivamente. O processo se dá da seguinte forma: os dados locais estão vinculados à memória nova em cada ativação devido à criação de um novo registro de ativação durante cada chamada. Esses dados locais são perdidos quando a ativação é encerrada, pois a memória associada a eles desaparece ao desempilhar o registro de ativação. Em relação à alocação de memória *heap*, blocos de memória contíguos são fornecidos a medida do necessitado para registros de ativação, fazendo com que seja possível liberar áreas da memória em qualquer ordem.

Tipos de dados em linguagens de programação podem ser do tipo imutável, ou seja, ao ser criada uma variável do tipo imutável, ela não poderá ser alterada. De acordo com a documentação do Python (FOUNDATION, 2024), as cadeias de caracteres da linguagem são imutáveis. As operações de manipulação de cadeias alteram as cadeias de variadas formas, e para isso ser possível com um tipo imutável, novas cadeias são criadas sempre que sofrem alterações. Em ambientes com *garbage collection*, cadeias imutáveis facilitam a identificação e limpeza de memória não mais usada, pois não há necessidade de rastrear alterações em objetos existentes.

2.3.2 Tempo de Execução

As diferentes abordagens de representação de cadeias de caracteres têm impacto no tempo de execução de formas diferentes. Ao realizar uma operação de concatenação de cadeias, saber o tamanho das cadeias a serem unidas é o primeiro passo a ser feito. Na abordagem de cadeias de caracteres terminadas por um caractere nulo, para calcular o comprimento resultante, é necessário percorrer as duas cadeias por completo até encontrar o caractere nulo, o que leva a um tempo proporcional ao comprimento da mesma. Isso

deve ser feito antes da concatenação em si, já que o espaço da memória deve ser reservado previamente.

Na abordagem de representação com prefixo de comprimento, o mesmo já está registrado na memória e é preciso apenas referenciar um endereço de memória, o que leva um tempo constante (COOPER; TORCZON, 2014, pp. 317–318).

A operações de busca de sub-cadeias e de comparação de cadeias também podem possuir tempos de execução menores na representação com prefixo de comprimento. A busca de sub-cadeias na abordagem de cadeias terminadas por um caractere nulo, requer a verificação de cada posição inicial possível na cadeia principal, comparando a sub-cadeia com a sequência de caracteres subsequente até encontrar uma correspondência ou alcançar o caractere nulo. Com o comprimento da cadeia principal e da sub-cadeia sendo conhecidos na abordagem de cadeias com prefixo de comprimento, o algoritmo pode descartar imediatamente posições onde a sub-cadeia não poderia caber.

Na comparação de cadeias, o processo acontece comparando caractere por caractere, até o ponto onde o caractere nulo é encontrado, ou caracteres diferentes aparecem. Na abordagem de cadeias de caracteres com prefixo de tamanho, o processo pode começar com a verificação dos comprimentos das cadeias. Se os comprimentos forem diferentes, a comparação pode ser concluída instantaneamente, evitando a necessidade de uma verificação caractere por caractere. Se os comprimentos forem iguais, a comparação prossegue como de costume.

2.3.3 Operações de Manipulação de Cadeias de Caracteres

Segundo Cooper e Torczon (2014, p. 315), o nível de suporte a cadeias de caracteres varia de uma linguagem de programação para outra. A linguagem Python, por exemplo, apresenta métodos que permitem realizar diversas ações em cima de cadeias de caracteres (FOUNDATION, 2024). Exitem mais de 40 métodos que têm objetivo de manipular cadeias na linguagem Python; alguns deles modificam a capitalização, como transformar todos os caracteres da cadeia em maiúsculo ou minúsculo, converter a primeira letra em maiúscula e o restante todo em minúsculo e também deixar apenas as primeiras letras de cada palavra como maiúscula. Outros métodos vão retornar verdadeiro ou falso se todos os caracteres da cadeia forem dígitos, decimais, maiúsculos, minúsculos, dentre outros. Métodos importantes como o de concatenação de cadeias, encontro de uma sub-cadeira dentro de uma cadeia de caracteres, substituição de ocorrências de sub-cadeias por outras, divisão em sub-cadeias, também estão presentes na linguagem.

Uma lista não exaustiva das operações é apresentada na Tabela 1, incluindo a sua descrição e exemplos. Um subconjunto desta lista será implementado na biblioteca padrão

da Robotics Language.

Tabela 1 – Operações de manipulação de cadeias de caracteres mais frequentes nas linguagens de programação.

Operação	**Descrição**	**Exemplos**
Concatenação	Concatena duas cadeias, a e b, retornando uma nova cadeia combinada.	C: strcat(a, b) Python: a + b
Contém	Retorna se a cadeia contém a sub-cadeia buscada como uma sub-cadeia.	C: strstr(str, substr) != NULL; Python: substr in str
Contém Caractere	Retorna um caractere específico de uma cadeia.	C: char ch = str[index] Python: char = str[index]
Comprimento	Retorna o número de caracteres em uma cadeia.	C: strlen(str); Python: len(str)
Substituição	Substitui todas as ocorrências de uma sub-cadeia por outra.	Python: str.replace(old, new)
Maiúsculas	Converte todos os caracteres de uma cadeia em maiúsculos.	Python: str.upper()
Minúsculas	Converte todos os caracteres de uma cadeia para minúsculos.	Python: str.lower()
Remover Espaços	Remove espaços em branco do início e do fim de uma cadeia.	Python: str.strip()
Divisão	Divide uma cadeia em uma lista de sub-cadeia, com base em um delimitador.	C: strtok(str, delim); Python: str.split(delim)
Igualdade	Testa se duas cadeias são iguais.	C: strcmp(str1, str2); Python: str1 == str2
Diferença	Testa se duas cadeias são diferentes.	Python: str1 != str2
Unir	Junta uma lista de cadeias em uma só.	Python: delimiter.join(strlist)
Esquerda	Retorna os primeiros n caracteres de uma cadeia.	C: strncpy(dest, str, n) Python: str[:n]
Direita	Retorna os últimos n caracteres de uma cadeia.	C: strncpy(dest, str + strlen(str) - n, n) Python: str[-n:]

2.4 Microcontroladores

Como descrito por Hussian et al. (2016, p. 21), microcontroladores são computadores em um único circuito integrado que incorporam uma CPU, RAM, ROM e portas de entrada e saída (E/S). Eles são projetados para executar tarefas específicas e são frequentemente

programados para controlar produtos automatizados, como sistemas de controle de motores, controles remotos, ferramentas elétricas, brinquedos e equipamentos de escritório. A quantidade de memória e interfaces que cabem em um único *chip* é limitada, por isso os microcontroladores tendem a ser usados em sistemas menores.

O termo 'micro' refere-se ao tamanho reduzido do dispositivo, enquanto 'controlador' indica sua função de controlar objetos, processos ou eventos. Um termo alternativo para microcontrolador é 'controlador embarcado', pois o microcontrolador e seus circuitos de suporte são frequentemente integrados nos dispositivos que controlam. O uso desses dispositivos varia desde o controle de motores em automóveis até teclados e impressoras para computadores.

Dois dos microcontroladores mais comuns são:

- ARM: Uma arquitetura de processadores que possui baixo consumo de energia, pois conta com um conjunto de instruções reduzido. Processadores ARM entregam uma performance relativamente alta, além de uma boa flexibilidade, devido ao seu conjunto de instruções (ADUSUMILLI, 2002).
- AVR: Microcontrolador desenvolvido pela Atmel e tem uso em eletrodomésticos diversos, carros e está presente no hardware do Arduino. A arquitetura dele é uma modificação da arquitetura Harvard. (SILVA, 2014).

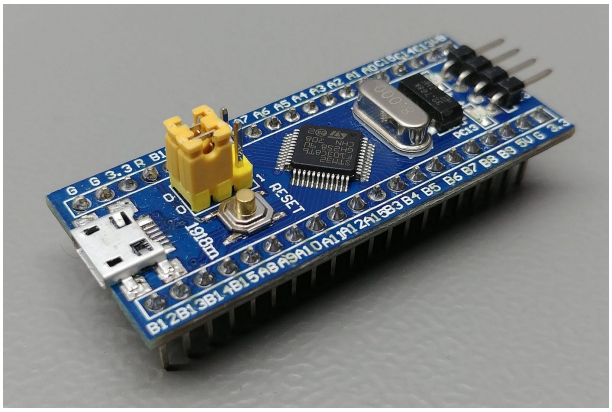


Figura 7 – Microcontrolador STM32F103 de arquitetura ARM fabricado pela STMicroeletronics. Pode ter de 64 a 128 *Kbytes* de memória *flash* (armazenamento), 20 *Kbytes* de RAM e até 80 pinos de E/S.

A quantidade limitada de memória *flash* (armazenamento) e RAM nos microcontroladores destaca a importância do controle eficiente de memória no desenvolvimento de linguagens de programação, como a Robotics Language. Devido às restrições de recursos, é crucial otimizar a utilização da memória. A implementação de cadeias de caracteres e a manipulação das mesmas precisa considerar todos esses fatores, para evitar problemas futuros.

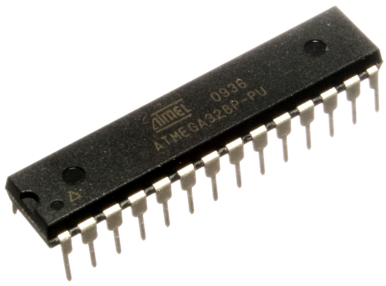


Figura 8 – Microcontrolador ATmega328P de arquitetura AVR fabricado pela Atmel. Possui 32kb de memória *flash* (armazenamento), 2kb de RAM e 28 pinos.

2.5 Robotics Language and Compiler

O Robcmp é um compilador projetado para microcontroladores usados em robótica e Internet das Coisas. A linguagem associada ao Robcmp visa isolar as particularidades dos microcontroladores, fornecendo uma camada de abstração de hardware diretamente no compilador e em sua biblioteca padrão, ao invés de depender de *frameworks* e bibliotecas. Essa abordagem permite que os programadores escrevam um único código, sem a necessidade de diretivas de compilação. Além disso, aumentando o poder da análise semântica com estas especificidades é possível que se previna os erros comuns encontrados quando do desenvolvimento de *firmware* em linguagens de propósito geral já na etapa de compilação.

O compilador Robcmp foi desenvolvido utilizando a linguagem C++, com o auxílio das ferramentas Flex[1] (versão 2.6.4) e Bison[2] (versão 3.8.2) para análise léxica e sintática, respectivamente. Para a geração do código intermediário, foi empregado o LLVM[3] (versão 18), um *framework* de compilação modular e reutilizável que facilita a otimização e a geração de código para múltiplas plataformas.

A IDE ideal para o desenvolvimento do Robcmp é o Visual Studio Code, devido ao seu suporte extensivo para C++ e à flexibilidade proporcionada por suas extensões. A extensão PlatformIO foi integrada ao ambiente de desenvolvimento para facilitar a depuração do *firmware* para microcontroladores. Além disso, foi criado suporte para o realce de sintaxe da Robotics Language, o que simplifica o processo de desenvolvimento.

A Figura 9 apresenta um trecho de código que implementa a função `itoa` na Robotics Language. Essa função converte um número inteiro (`int16 value`) em uma cadeia de caracteres na base especificada (`int8 base`) e armazena o resultado na variável `char[] result`. Ao final, linhas 39 a 48, a função reverte a cadeia no *array result* para

[1] https://www.gnu.org/software/flex
[2] https://www.gnu.org/software/bison
[3] https://www.llvm.org

garantir a ordem correta dos dígitos. Além disso é possível observar o realce de sintaxe criado para o Visual Studio Code.

```
6  const itoa_chars = "0123456789abcdef";
7
8  void itoa(int16 value, int8 base, char[] result) {
9      if base < 2 or base > 16 {
10         result[0] = 0;
11         return;
12     }
13
14     negative = value < 0;
15     // abs without branch
16     mask = value >> 15;
17     value = (value ^ mask) - mask;
18
19     i = 0;
20     if value == 0 {
21         result[i] = '0';
22         i++;
23     } else {
24         while value != 0 {
25             temp = value;
26             value /= base;
27             result[i] = itoa_chars[temp - value * base];
28             i++;
29         }
30         if negative {
31             result[i] = '-';
32             i++;
33         }
34     }
35
36     result[i] = 0;
37
38     // reverse result
39     j = 0;
40     i--;
41     while j < i {
42         temp = result[j];
43         result[j] = result[i];
44         result[i] = temp;
45         j++;
46         i--;
47     }
48 }
```

Figura 9 – Trecho de código em Robotics Language com o código da função `itoa`, que converte números inteiros em *strings*.

3 Trabalhos relacionados

3.1 Introdução

Para realizar o levantamento de trabalhos relacionados, foi adotada uma revisão narrativa da literatura, que permitiu, observando os principais catálogos de *softwares* para microcontroladores, estabelecer um conjunto dos trabalhos, *frameworks* e bibliotecas correlatos. O ponto inicial de busca dos trabalhos relacionados foi o conjunto de monografias já defendidas no contexto do projeto de pesquisa ao qual este trabalho também se insere (ex. (SUBHI, 2019; RODRIGUES, 2018)), seguido pela busca de trabalhos correlatos no PlatformIO Registry, que é um catálogo de mais de dez mil bibliotecas e ferramentas para *software* embutido (PlatformIO Labs OÜ, 2024).

3.2 Metodologia de análise

Para a análise dos trabalhos relacionados, foram estabelecidos critérios técnicos que permitem avaliar e comparar os métodos de manipulação de cadeias de caracteres.

Os critérios nas sub-seções a seguir foram considerados na análise.

3.2.1 Nível de suporte (SUP)

As linguagens de programação apresentam diferentes níveis de suporte à manipulação de cadeias de caracteres. Esse critério avalia o quão desenvolvida e integrada é a manipulação de cadeias na linguagem, podendo ser caracterizadas como baixo, médio ou alto nível de suporte.

3.2.2 Representação (REP)

Uma característica presente nas cadeias de caracteres é a representação. Esse critério avalia como as cadeias são internamente armazenadas na linguagem, sendo com terminador nulo ou com prefixo de comprimento.

3.2.3 Codificação (COD)

Existem diferentes formas de codificar caracteres dentro de uma linguagem de programação. As duas principais codificações são o ASCII e o UTF-8 do padrão Unicode.

3.3 Trabalhos analisados

Foram considerados como correlatos aqueles estudos que, de maneira direta ou indireta, abordam a criação e aplicação de linguagens voltadas para domínios específicos, alinhando-se aos objetivos deste trabalho. A seguir, são descritos três estudos analisados, com destaque para suas contribuições e limitações no contexto deste projeto.

3.3.1 Majestic: Uma Ampliação de uma Linguagem de Programação para Robótica Educacional

O trabalho de Subhi (2019) ampliou a linguagem de programação denominada Majestic, desenvolvida na UFG, Regional Jataí, para robótica educacional, com o objetivo de facilitar o ensino de programação. A nova linguagem foi projetada para melhorar a expressividade no ensino de robótica, adicionando suporte a funções com parâmetros, vetores, e matrizes estáticas, todos alocados em pilha. Majestic diferencia-se das linguagens tradicionais usadas na robótica educacional por ser específica de domínio, o que simplifica o aprendizado ao reduzir a complexidade sintática.

As linguagens de propósito específico, como Majestic, possuem características que facilitam a utilização no contexto educacional, proporcionando benefícios como maior produtividade e manutenibilidade, porém, com desafios de custo e complexidade de implementação. Além disso, o desenvolvimento de Majestic também se deu com a integração ao framework LLVM, possibilitando a geração de código compatível com os processadores AVR e ARM usados em robôs de baixo custo.

A pesquisa de Subhi (2019) contribuiu para a ampliação da linguagem de programação e, os avanços obtidos foram posteriormente incorporados no Robcmp, linguagem subjacente a esta pesquisa, que amplia suas capacidades.

3.3.2 Avr-Rust

O AVR-Rust (2024) é uma extensão do compilador Rust, projetada para suportar microcontroladores AVR, amplamente utilizados em sistemas embarcados, como o AT-

mega328p. O principal objetivo é oferecer uma alternativa segura e eficiente ao tradicional uso de C/C++ nesses dispositivos.

A iniciativa Avr-Rust visa trazer as vantagens do ecossistema Rust para o desenvolvimento de sistemas embarcados, permitindo que desenvolvedores explorem a expressividade da linguagem ao mesmo tempo em que mantêm o controle detalhado sobre o hardware. O projeto integra recursos avançados de Rust, como tipos seguros, gerenciamento de memória sem coleta de lixo (garbage collection), e abstrações de alto nível, com a possibilidade de uso em dispositivos com recursos limitados.

Como é mostrado na documentação da Linguagem Rust (KLABNIK; NICHOLS, 2024), Rust adota uma abordagem poderosa se tratando de manipulação de cadeias de caracteres, tratando-as como objetos imutáveis da classe *String*, que armazenam a cadeia internamente e oferecem métodos integrados para manipulação. Essa abstração eleva o nível de suporte para cadeias, permitindo operações como concatenação, substituição e busca de forma simplificada e eficiente.

Internamente, a representação das cadeias em Rust utiliza um prefixo de tamanho, que armazena a quantidade de *bytes* ocupados pela cadeia, evitando a necessidade de caracteres terminadores. Além disso, Rust utiliza codificação UTF-8, o que garante compatibilidade com caracteres de diversos idiomas e padrões Unicode.

3.3.3 Arduino

Arduino (ARDUINO, 2024) é uma plataforma de hardware e software amplamente utilizada em sistemas embarcados, conhecida por sua simplicidade e acessibilidade. Inicialmente projetado para introduzir iniciantes ao mundo da eletrônica e programação, o Arduino tornou-se uma ferramenta versátil, empregada em protótipos, projetos educacionais e aplicações industriais de baixo custo.

O ecossistema Arduino consiste em placas baseadas em microcontroladores AVR (e outros), acompanhadas pela Arduino IDE, um ambiente de desenvolvimento simplificado que utiliza a linguagem C++ juntamente com um conjunto de bibliotecas e funções que facilitam a interação com os microcontroladores. A escolha do C++ como base proporciona flexibilidade para programadores avançados, enquanto bibliotecas integradas e uma comunidade ativa oferecem suporte para iniciantes.

No contexto da manipulação de cadeias de caracteres, o Arduino adota uma abordagem básica, com cadeias como *arrays* de caracteres. Além disso, a API do Arduino inclui uma classe *String*, que oferece métodos para operações como concatenação, busca e substituição. Contudo, a implementação da classe *String* possui limitações, como maior uso de memória dinâmica, o que pode ser problemático em microcontroladores com recursos

restritos.

A representação de cadeias no Arduino é baseada na terminação em nulo, seguindo o padrão tradicional do C. A codificação das cadeias é o ASCII, com as funções de manipulação tratando esses caracteres.

3.4 Resumo Comparativo

Esta seção tem como objetivo apresentar uma análise comparativa dos trabalhos relacionados, destacando a utilização das características estudadas nesta pesquisa. A seguir, serão detalhadas as principais diferenças e semelhanças entre os trabalhos com base em critérios específicos que orientam esta pesquisa.

A Tabela 2 resume essa comparação, indicando como cada trabalho aborda os critérios estabelecidos, como nível de suporte à manipulação de cadeias de caracteres, forma de representação e tipo de codificação adotada.

Tabela 2 – Comparativo entre trabalhos relacionados.

Trabalhos	SUP	REP	COD
Majestic Language	Baixo nível	Sem terminação nula	ASCII
AVR-Rust	Alto nível	Prefixo de comprimento	UTF-8
Arduino	Médio nível	Terminação nula	ASCII
Proposta desta Pesquisa	Médio nível	Sem terminação nula	UTF-8

Em suma, o trabalho de Subhi (2019) permitiu ampliar a linguagem de programação suportada pelo Robcmp, mas sem oferecer suporte à manipulação de cadeias de caracteres. O Avr-Rust apresenta o nível mais avançado de suporte a cadeias de caracteres, enquanto o Arduino fornece o mesmo suporte de cadeia de caracteres da linguagem C, ampliado por uma classe String (em C++) que facilita a programação mais exige mais recursos do microcontrolador. Este trabalho expande as capacidades do Robcmp ao adicionar suporte à manipulação de cadeias de caracteres em UTF-8.

4 Implementação ou Construção

4.1 Introdução

Este capítulo especifica os materiais utilizados e apresenta o desenvolvimento das principais funções de manipulação de cadeias de caracteres para a linguagem Robcmp. As funções implementadas foram selecionadas por sua relevância na programação e por permitirem uma análise direta de sua eficácia dentro do compilador. O desenvolvimento dessas funções considerou tanto a simplicidade de uso quanto a performance, levando em conta o comportamento esperado em diversas situações. Cada função será detalhada, abordando suas estruturas e os testes realizados para garantir a integridade do comportamento no compilador.

4.2 Materiais

Nesta seção serão especificados os materiais e métodos que foram utilizados para o desenvolvimento dos novos recursos na linguagem.

4.2.1 Software

Os novos recursos no desenvolvimento do compilador foram escritos em C++, utilizando o auxílio dos programas Flex[1] na versão 2.6.4 e Bison[2] na versão 3.8.2 para gerar as alterações necessárias no analisador léxico e sintático da Robotics Language. As novas validações semânticas, para verificação de possíveis erros semânticos com as cadeias de caracteres, também foram implementada em C++. Utilizando o LLVM[3], versão 18, foi gerado o código intermediário, aproveitando suas modernas e otimizadas ferramentas de compilação. Todo o desenvolvimento e depuração foram realizados no editor de texto da IDE Visual Studio Code, que oferece suporte extensivo para C++, a extensão PlatformIO, que tem como papel a depuração do *firmware* para microcontroladores, além também do suporte criado para o realce de sintaxe para a Robotics Language, facilitando o processo de desenvolvimento.

[1] https://www.gnu.org/software/flex
[2] https://www.gnu.org/software/bison
[3] https://www.llvm.org

4.2.2 Hardware

Os experimentos foram conduzidos em um notebook MacBook Air M1 (2020), equipado com o sistema operacional macOS Sequoia 15.1, chip M1, 8 GB de memória RAM e 256 GB de armazenamento. O compilador atualizado foi testado no simulador AVR, incluindo o AVR modelo ATmega328P e ATmega1284P.

4.3 Validação e Experimentos

Com a finalidade de garantir a qualidade e correção das manipulações de cadeias implementadas, foram escritos testes unitários para cada operação nas cadeias de caracteres na própria Robotics Language e foram executados automaticamente. O formato foi o mesmo adotado pelas outras funcionalidades do compilador, conforme pode-se visualizar em <https://github.com/thborges/robcmp/tree/llvm18/test/general>.

Para a realização dos experimentos, os programas desenvolvidos para validação foram executados em simulador AVR. Devido ao limite de tempo para desenvolvimento da pesquisa, experimentos com a arquitetura STM32 não puderam ser realizados e foram deixados como trabalhos futuros. Foram consideradas as seguintes variáveis nos experimentos:

- Tempo de execução, como testes de desempenho, no estilo *benchmark*, a partir de comparações de um código *baseline* escrito em C e o seu correspondente, escrito na Robotics Language;
- Uso de memória SRAM: Verificação do tamanho do código gerado depois de compilado.

4.4 Funções de manipulação de cadeias de caracteres

A seguir, detalhamos cada uma das funções implementadas. O código da implementação foi disponibilizado no Anexo A.

4.4.1 *Função* containsstr *(Função contém)*

A função *containsstr* foi desenvolvida para verificar se uma sub-cadeia está presente em uma cadeia maior e, em caso afirmativo, retornar a posição inicial da primeira ocorrência. Caso a sub-cadeia não seja encontrada, a função retorna -1.

O funcionamento da função começa com a inicialização de índices de controle para percorrer a cadeia principal e comparar os caracteres com os da sub-cadeia. Um laço externo percorre a cadeia principal (str), enquanto um laço interno compara os caracteres consecutivos da sub-cadeia (substr) com os da posição correspondente na cadeia principal. Essa verificação ocorre até que uma discrepância seja encontrada ou todos os caracteres da sub-cadeia sejam comparados com sucesso. Quando a comparação é bem-sucedida, a função retorna o índice inicial da ocorrência. Se o laço externo é concluído sem que a sub-cadeia seja encontrada, a função retorna -1.

```
containsstr("Exemplo de string", "string"); // Retorna 11
```

4.4.2 *Função* lenstr *(Função comprimento, em caracteres)*

A função *lenstr* calcula o comprimento de uma cadeia codificada em UTF-8, considerando caracteres em vez de *bytes*.

O algoritmo percorre a cadeia *byte* a *byte*, utilizando uma máscara de *bits* para identificar os *bytes* iniciais de cada caractere. Apenas os *bytes* cujo padrão nos dois *bits* mais significativos não é 10 são considerados válidos como início de caracteres. Esses padrões indicam se o *byte* é inicial (0x, 110x, 1110x ou 11110x) ou de continuidade (10xxxxxx). Assim, a função incrementa o contador sempre que encontra um *byte* inicial, garantindo a contagem correta de caracteres, independentemente do número de *bytes* usados para codificá-los. A cada detecção de um caractere, o comprimento total é incrementado. Ao final da execução, a função retorna o número total de caracteres presentes na cadeia.

```
lenstr("Comprimento"); // Retorna 11
```

4.4.3 *Função* equalstr *(Função igual)*

A função *equalstr* verifica se duas cadeias são equivalentes, tanto em seu conteúdo quanto em seu comprimento.

Inicialmente, a função realiza uma comparação direta entre os *bytes* das duas cadeias, interrompendo o processo caso uma discrepância seja encontrada. Caso todos os *bytes* sejam iguais até o final da cadeia mais curta, a função utiliza *lenstr* para verificar o comprimento em caracteres das duas cadeias. Apenas se ambas tiverem o mesmo comprimento, a função retorna *true*, indicando equivalência.

```
equalstr("Texto igual", "Texto igual") // Retorna true
equalstr("Texto diferente", "TEXTO DIFERENTE"); // Retorna false
```

4.4.4 *Funções* strtouppercase *e* strtolowercase *(Funções maiúsculo e minúsculo)*

As funções *strtouppercase* e *strtolowercase* foram projetadas para transformar uma cadeia UTF-8 em sua versão maiúscula ou minúscula, respectivamente.

Ambas utilizam um mapeamento abrangente de caracteres Unicode que cobre todos os caracteres com versões maiúsculas e minúsculas, incluindo caracteres latinos e de outras línguas.

Para realizar o mapeamento de caracteres maiúsculos e minúsculos, utilizou-se uma tabela de conversão fornecida pela IBM, disponível em: <https://www.ibm.com/docs/en/i/7.3?topic=tables-unicode-lowercase-uppercase-conversion-mapping-table>. Essa tabela contém quatro colunas: o ponto de código do caractere minúsculo, o ponto de código do caractere maiúsculo, a descrição do caractere minúsculo e a descrição do caractere maiúsculo. A tabela possui 666 linhas, cobrindo uma ampla gama de caracteres Unicode com versões maiúsculas e minúsculas.

Para automatizar o processo de extração e formatação dos dados, escreveu-se um *script* em Python. O *script* foi desenvolvido para processar a tabela e gerar automaticamente um formato adequado ao código, extraindo apenas as duas primeiras colunas (pontos de código minúsculo e maiúsculo) e formatando-os da seguinte maneira: "0x0061, 0x0041,".

O funcionamento das funções principais é baseado em etapas realizadas por funções auxiliares desenvolvidas especificamente para o propósito da transformação de caracteres. Inicialmente, cada caractere da cadeia é decodificado utilizando a função *utf8decode*, que identifica o ponto de código correspondente. Em seguida, esse ponto de código é passado para *touppercase* ou *tolowercase*, que aplicam o mapeamento Unicode necessário para realizar a transformação. O ponto de código transformado é então recodificado em UTF-8 por meio da função *utf8encode* e sobrescrito na cadeia original.

```
1 strtouppercase("texto em caixa baixa"); // Retorna "TEXTO EM CAIXA BAIXA"
2 strtolowercase("TEXTO EM CAIXA ALTA"); // Retorna "texto em caixa alta"
```

5 Experimentos

Neste capítulo, apresenta-se a metodologia usada na validação do código seguida do resultado dos experimentos.

Para a execução dos experimentos, foram desenvolvidos quatro algoritmos com objetivos específicos, implementados tanto na Robotics Languagem quanto em C. Na compilação, foi empregada a *flag* de otimização '-Os', que é o melhor nível de otimização para tamanho em C, e '-Oz' para Robcmp, que é o melhor nível no *backend* LLVM, um compromisso entre tamanho e velocidade de execução do código. Cada algoritmo foi projetado para ser funcionalmente equivalente em ambas as linguagens. Cada algoritmo foi executado cinco vezes utilizando os mesmos parâmetros.

5.1 Algoritmos dos Experimentos

Os algoritmos desenvolvidos para os experimentos realizados foram estruturados de maneira a avaliar aspectos específicos da manipulação de cadeias de caracteres e funções básicas de comparação, medição e transformação. A seguir, são apresentados os detalhes de cada um dos quatro algoritmos utilizados. O código dos mesmos está disponível no Anexo B.

Para os 3 primeiros algoritmos, as cadeias de caracteres foram geradas aleatoriamente através da ferramenta: <https://onlinetools.com/utf8/generate-random-utf8>

5.1.1 Função containsstr

O primeiro experimento tem como objetivo testar a capacidade de identificação de sub-cadeia dentro de uma cadeia maior. Para isso, o código utiliza três cadeias com caracteres UTF-8: uma contendo 500 caracteres, e duas outras contendo 250 caracteres cada. O teste consiste em usar a função *containsstr* para verificar se a maior cadeia contém cada uma das menores, sendo que uma delas é uma sub-cadeia da maior e a outra não. O algoritmo valida se a comparação é feita corretamente, retornando 0 caso o comportamento esperado seja cumprido (a primeira sub-cadeia é encontrada, e a segunda não é).

5.1.2 Função lenstr

O segundo experimento avalia a precisão da função de cálculo do comprimento de uma cadeia de caracteres. Para isso, foi utilizada uma cadeia de 1000 caracteres, gerada aleatoriamente. O teste verifica se o comprimento retornado pela função *lenstr* é o esperado (1000 caracteres). O algoritmo compara o valor retornado pela função com o valor esperado e retorna 0 caso a verificação seja bem-sucedida.

5.1.3 Função equalstr

Este experimento testa a comparação de igualdade entre cadeias. Foram utilizadas três cadeias, cada uma com 500 caracteres. A função *equalstr* compara a primeira cadeia com a segunda e deve ser igual, enquanto a comparação com a terceira deve ser diferente. O algoritmo valida se as comparações são feitas corretamente e retorna 0 caso as condições sejam atendidas.

5.1.4 Funções strtouppercase e strtolowercase

O último experimento foca na transformação de caracteres para maiúsculas e minúsculas utilizando as funções *strtouppercase* e *strtolowercase*. O mapeamento dessas funções contém 666 linhas, cada uma associando um caractere minúsculo à sua versão maiúscula correspondente. Para garantir a execução eficiente dos testes, foi desenvolvido um *script* em Python que gera arquivos de teste baseados nesse mapeamento.

Ao todo, foram criados 222 arquivos de teste, cada um contendo 3 pares de caracteres minúsculos e suas respectivas versões maiúsculas, totalizando os 666 pares mapeados. Cada arquivo verifica a correta transformação de caracteres de minúsculas para maiúsculas e vice-versa, assegurando que as funções de conversão operam de forma consistente e precisa para todos os caracteres contemplados no mapeamento.

5.2 Experimentos com Tempo de Execução

Para avaliar o tempo de execução dos algoritmos de experimentos, foi utilizado o comando time, nativo do sistema operacional, que fornece o tempo gasto na execução de um processo.

A Tabela 3 apresenta a média dos tempos de execução, em milissegundos, obtidos nos experimentos realizados com os algoritmos escritos e compilados com o Robcmp,

comparados à média dos tempos de execução, também em milissegundos, dos algoritmos equivalentes em linguagem C.

O quarto experimento foi realizado através da execução, em sequência, dos 222 casos que compõem o teste, com o tempo registrado na tabela representando a soma total do tempo de execução de todos os testes. Assim como nos experimentos anteriores, o tempo apresentado é a média dos tempos registrados durante a execução de cada conjunto de testes.

Durante os testes, apenas o processo de compilação e o sistema operacional estavam ativos na máquina, garantindo que os resultados não fossem influenciados por outras tarefas.

Com base nos resultados, observa-se que o compilador Robcmp apresentou um desempenho aproximadamente 0,32% mais lento em relação ao código equivalente escrito em C e compilado com o AVR-GCC quando consideradas todas as funções. Contudo, ao analisar apenas as três primeiras funções, o desempenho do Robcmp foi cerca de 29,81% mais lento.

Tabela 3 – Tempo de execução, em milissegundos.

Função	**Robcmp**	**C(AVR-GCC)**
containsstr	3.000	2.300
lenstr	3.000	2.400
equalstr	3.000	2.233
strtouppercase e strtolowercase	669.533	669.433

5.3 Experimentos com o uso de memória SRAM (Tamanho de Código Objeto)

No que diz respeito ao uso de memória SRAM, ou seja, o tamanho do código compilado, destaca-se a importância de um código otimizado, considerando a limitada capacidade de memória disponível em microcontroladores para armazenar o programa objeto.

Nestes experimentos foram utilizados os mesmos quatro algoritmos desenvolvidos para a análise de tempo de execução. Como o tamanho do arquivo gerado é determinístico, a criação do arquivo objeto (.o) foi realizada apenas uma vez para cada algoritmo.

No quarto experimento, foi realizada a soma dos tamanhos dos arquivos gerados e, em seguida, a divisão pelo número total de testes, resultando em uma média do tamanho dos arquivos. Como todos os arquivos continham a mesma lógica, a variação no tamanho se dá apenas devido ao tamanho dos caracteres UTF-8, que possuem diferentes quantidades

de *bytes*. Os tamanhos obtidos nos testes foram: 4702, 4716, 4725 e 4710 *bytes*.

A Tabela 4 apresenta a comparação dos tamanhos dos arquivos gerados pelos algoritmos, compilados tanto pelo compilador Robcmp quanto pelo compilador AVR-GCC, na linguagem C.

Com base nos resultados, observa-se que o compilador Robcmp produziu arquivos finais aproximadamente 16% menores em relação aos gerados pelo AVR-GCC, considerando códigos equivalentes em linguagem C.

Tabela 4 – Tamanho do código objeto em *bytes*.

Função	**Robcmp**	**C(gcc)**
containsstr	2700	2778
lenstr	2635	2808
equalstr	2862	2840
strtouppercase e strtolowercase	4713.5	6934.4

5.4 Considerações Finais

Foi possível verificar que no experimento de tempo de execução, o compilador Robcmp teve um desempenho aproximadamente 0.32% inferior. No entanto, ao analisar apenas as três primeiras funções, a diferença de desempenho foi mais significativa, com o Robcmp apresentando uma lentidão de aproximadamente 29,81%. O tamanho dos experimentos que se executam muito rapidamente, pode ter comprometido esta avaliação. Em trabalhos futuros, experimentos com programas maiores devem ser elaborados.

Em relação ao uso de memória SRAM, observou-se uma vantagem aproximada de 16% no tamanho do código objeto para o Robcmp, comparado ao código equivalente escrito em C e compilado no compilador AVR-GCC.

6 Conclusões e Trabalhos Futuros

O presente trabalho implementou a base para o processamento de cadeia de caracteres na Robotics Language e cinco funções de manipulação de cadeias representadas em UTF-8, sendo elas: *containsstr*, *lenstr*, *equalstr*, *strtouppercase* e *strtolowercase*. As mesmas foram desenvolvidas com foco na eficiência, precisão e adequação ao contexto do Robcmp.

A adoção do padrão de codificação UTF-8 mostrou-se uma escolha estratégica para garantir a acessibilidade do compilador a diferentes nacionalidades. Por sua ampla compatibilidade e suporte a diversos sistemas de escrita, o UTF-8 torna o Robcmp mais inclusivo e adaptável a contextos globais.

Através do experimentos foi possível concluir que o Robcmp teve um tamanho de arquivo gerado menor que o compilador AVR-GCC, porém com um tempo de execução maior.

Como possíveis direções para aprimorar o Robcmp e ampliar seu escopo de aplicação, sugere-se os seguintes trabalhos futuros:

- Expansão do conjunto de funções de manipulação de cadeias de caracteres: incluir operações como substituição de sub-cadeias, divisão de cadeias em base a delimitadores e formatação de cadeias para cenários específicos.
- Expansão dos experimentos de tempo de execução, para ocuparem mais tempo de CPU e permitirem uma melhor avaliação desta variável nos experimentos.

Com base nas contribuições deste trabalho e nas possibilidades de melhorias futuras, espera-se que o Robcmp continue evoluindo, atendendo às necessidades específicas de sistemas embarcados e consolidando sua posição como uma linguagem de programação eficiente e inovadora para esse domínio.

Referências

ADUSUMILLI, S. System and method to reduce power consumption in advanced risc machine(arm) based systems. *uspto*, p. 5, 2002. Citado na página 28.

AHO, R. S. A. V.; ULLMAN, J. D. *Compiladores. Princípios, Técnicas e Ferramentas*. Rio de Janeiro, RJ: Addison Wesley, 1995. 344 p. ISBN 10. 8588639246. Citado 7 vezes nas páginas 9, 17, 18, 19, 20, 21 e 25.

ARDUINO. *Arduino Documentation*. 2024. Https://docs.arduino.cc/. Acessado em: 2024-11-27. Citado na página 33.

AVR-RUST. *Book AVR-Rust*. 2024. Https://github.com/avr-rust/book.avr-rust.com/tree/master. Acessado em: 2024-11-10. Citado na página 32.

BECKER, J.; COLLINS, L.; DAVIS, M. *The Unicode Standard Version 6.0 – Core Specification*. Mountain View, CA: Unicode Consortium, 2011. 638 p. ISBN 978-1-936213-01-6. Citado 4 vezes nas páginas 9, 15, 23 e 25.

COOPER, K. D.; TORCZON, L. *Construindo Compiladores*. Rio de Janeiro, RJ: Morgan Kaufmann, 2014. 656 p. ISBN 978-85-352-5564-5. Citado 10 vezes nas páginas 9, 14, 15, 17, 18, 19, 20, 22, 23 e 26.

FISCHER RON K. CYTRON, R. J. L. J. C. N. *Crafting a Compiler*. Boston, Massachusetts: Addison Wesley, 2009. 683 p. ISBN 10: 0-13-606705-0. Citado 3 vezes nas páginas 17, 18 e 19.

FOUNDATION, P. software. *Python 3.12.4 documentation*. 2024. Https://docs.python.org/3/. Acessado em: 2024-07-02. Citado 3 vezes nas páginas 14, 25 e 26.

HUSSIAN, A. et al. Programming a microcontroller. *ijca*, v. 155, n. 5, p. 309–318, 2016. Citado na página 27.

INFRASTRUCTURE, L. C. *Getting Started with the LLVM System*. 2024. Https://www.llvm.org/docs/GettingStarted.html. Acessado em: 2024-07-08. Citado 2 vezes nas páginas 9 e 22.

KLABNIK, S.; NICHOLS, C. *The Rust Programming Language*. 2024. Https://doc.rust-lang.org/book/title-page.html. Acessado em: 2024-11-27. Citado na página 33.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California: [s.n.], 2004. Citado na página 17.

MERTZ, D. *Text Processing in Python*. Boston, MA: Addison Wesley, 2003. 544 p. ISBN 0-321-11254-7. Citado na página 14.

PlatformIO Labs OÜ. *PlatformIO Registry*. 2024. Https://registry.platformio.org/. Acessado em: 2024-11-13. Citado na página 31.

RODRIGUES, D. M. *Elaboração de uma Linguagem de Programação Específica para Robótica Educacional*. 65 p. Monografia — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2018. Citado na página 31.

SILVA, A. J. B. da. *Um Modelo de Baixo Custo Para Aulas de Robótica Educativa Usando a Interface Arduino*. Dissertação (Mestrado) — Universidade Federal de Alagoas, Maceió, AL, Brasil, 2014. Citado na página 28.

STAFF, T. I. A. *Best of Interface Age - Volume 2 General purpose software*. Portland, Oregon: Dilithium Press, 1980. 204 p. ISBN 0-918398-37-1. Citado 4 vezes nas páginas 9, 15, 23 e 24.

SUBHI, V. A. *Majestic: Uma Ampliação de uma Linguagem de Programação para Robótica Educacional*. 57 p. Monografia — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2019. Citado 4 vezes nas páginas 16, 31, 32 e 34.

Anexos

ANEXO A – Funções em Robcmp

```
int16 containsstr(char[] str, char[] substr) {
    i = uint16 (0);

    while i < str.size {
        j = uint16 (0);

        while (str[i + j] == substr[j]) {
            j++;
        }

        if j == substr.size {
            return i;
        }
        i++;
    }
    return -1;
}
```

```
use string.lenstr;

bool equalstr(char[] s1, char[] s2) {

    i = int16 (0);
    while (i < s1.size and i < s2.size) {
        if (s1[i] != s2[i]) {
            return false;
        }
        i++;
    }

    str1 = lenstr(s1);
    str2 = lenstr(s2);
    if str1 == str2 {
        return true;
    } else {
        return false;
    }
}
```

```
int16 lenstr(char[] s) {
    length = int16 (0);
    i = int16 (0);

    while i < s.size {
```

```
        if ((s[i] & 0xC0) != 0x80) {
            length++;
        }
        i++;
    }
    return length;
}
```

```
use string.equalstr;

mapping = {
    0x0061, 0x0041 ,
    0x0062, 0x0042 ,
    0x0063, 0x0043 ,
    0x0064, 0x0044 ,
    0x0065, 0x0045 ,
    0x0066, 0x0046 ,

    //Restante do mapeamento

    0xFF55, 0xFF35 ,
    0xFF56, 0xFF36 ,
    0xFF57, 0xFF37 ,
    0xFF58, 0xFF38 ,
    0xFF59, 0xFF39 ,
    0xFF5A, 0xFF3A
};

    type Codepoint {
        code = uint32(0);
    }

    int32 utf8_decode(char[] s, uint16 i, Codepoint c) {
        if (s[i] < 0x80) {
            c.code = s[i];
            return 1; // 1 byte
        } else if ((s[i] & 0xE0) == 0xC0) {
            c.code = ((s[i] & 0x1F) << 6) | (s[i+1] & 0x3F);
            return 2; // 2 bytes
        } else if ((s[i] & 0xF0) == 0xE0) {
            c.code = ((s[i] & 0x0F) << 12) | ((s[i+1] & 0x3F) << 6) | (s[i+2] & 0x3F);
            return 3; // 3 bytes
        } else if ((s[i] & 0xF8) == 0xF0) {
            c.code = ((s[i] & 0x07) << 18) | ((s[i+1] & 0x3F) << 12) | ((s[i+2] & 0x3F
    ) << 6) | (s[i+3] & 0x3F);
            return 4; // 4 bytes
        }
```

```
        return -1;
    }

    int32 utf8_encode(uint32 codepoint, char[] output) {
        if (codepoint <= 0x7F) {
            output[0] = codepoint & 0x7F;
            return 1; // 1 byte
        } else if (codepoint <= 0x7FF) {
            output[0] = 0xC0 | ((codepoint >> 6) & 0x1F);
            output[1] = 0x80 | (codepoint & 0x3F);
            return 2; // 2 bytes
        } else if (codepoint <= 0xFFFF) {
            output[0] = 0xE0 | ((codepoint >> 12) & 0x0F);
            output[1] = 0x80 | ((codepoint >> 6) & 0x3F);
            output[2] = 0x80 | (codepoint & 0x3F);
            return 3; // 3 bytes
        } else if (codepoint <= 0x10FFFF) {
            output[0] = 0xF0 | ((codepoint >> 18) & 0x07);
            output[1] = 0x80 | ((codepoint >> 12) & 0x3F);
            output[2] = 0x80 | ((codepoint >> 6) & 0x3F);
            output[3] = 0x80 | (codepoint & 0x3F);
            return 4; // 4 bytes
        }
        return -1;
    }

    int32 to_uppercase(uint32 codepoint) {
        i = uint16(0);
        while i < mapping.size {
            if mapping[i] == codepoint {
                return mapping[i + 1];
            }
            i += 2;
        }

        if (codepoint >= 0x61 and codepoint <= 0x7A) {
            return codepoint & 0x5F;
        }
        return codepoint;
    }

    int32 to_lowercase(uint32 codepoint) {
        i = uint16(0);
        while i < mapping.size {
            if mapping[i + 1] == codepoint {
                return mapping[i];
            }
```

```
            i += 2;
        }

        if (codepoint >= 0x41 and codepoint <= 0x5A) {
            return codepoint | 0x20;
        }
        return codepoint;
    }

    void strcpy(char[] src, uint16 spos, char[] dst, uint16 dpos, uint16 len) {
        i = 0;
        if (spos + len > src.size) or (dpos + len > dst.size) {
            return;
        }
        while (i < len) {
            dst[dpos + i] = src[spos + i];
            i++;
        }
    }


    void strtouppercase(char[] str) {
        p = uint16 (0);
        w = uint16 (0);
        output = {0:4};

        while p < str.size {
            codepoint = Codepoint();
            len = utf8_decode(str, p, codepoint);
            if (len < 1) {
                p++;
            } else {
                uppercase_codepoint = to_uppercase(codepoint.code);
                out_len = utf8_encode(uppercase_codepoint, output);

                if (out_len > 0) {
                    strcpy(output, 0, str, w, out_len);
                    w += out_len;
                }

                p += len;
            }

        }
    }

    void strtolowercase(char[] str) {
```

```
        p = uint16 (0);
        w = uint16 (0);
        output = {0:4};

        while p < str.size {
            codepoint = Codepoint();
            len = utf8_decode(str, p, codepoint);
            if (len < 1) {
                p++;
            } else {
                lowercase_codepoint = to_lowercase(codepoint.code);
                out_len = utf8_encode(lowercase_codepoint, output);

                if (out_len > 0) {
                    strcpy(output, 0, str, w, out_len);
                    w += out_len;
                }

                p += len;
            }

        }
    }
```

ANEXO B – Experimentos em Robcmp

```
use string.containsstr;

const str = " //Cadeia com 500 caracteres UTF-8 ";

const substr1 = " //Cadeia com 250 caracteres UTF-8 contidas em str ";

const substr2 = " //Cadeia com 250 caracteres UTF-8 nao contidas em str ";

int8 main() {
    r = containsstr(str, substr1);
    s = containsstr(str, substr2);
    if r != -1 and s == -1 {
        return 0;
    } else {
        return 1;
    }
}
```

```
use string.lenstr;

const str = " //Cadeia com 1000 caracteres ";

int8 main() {
    r = lenstr(str);
    if r == 1000 {
        return 0;
    } else {
        return 1;
    }
}
```

```
use string.equalstr;

const str = " //Cadeia com 500 caracteres UTF-8 ";

const str_equal = " //Cadeia com 500 caracteres UTF-8, igual str ";

const str_diff = " //Cadeia com 500 caracteres UTF-8, diferente de str ";

int8 main() {
    if equalstr(str, str_equal) and !equalstr(str, str_diff) {
        return 0;
    } else {
        return 1;
```

```
    }
}
```

```
use string.utf8lenstr;
use string.upperlowerstr;
int32 main() {
    ol_str = " //Cadeia com 3 caracteres minusculos do mapeamento ";
    ou_str = " //Cadeia com os 3 caracteres em versao mauscula ";
    l_str = " //Cadeia com os mesmos 3 caracteres minusculos do mapeamento ";
    u_str = " //Cadeia com os mesmos 3 caracteres em versao mauscula ";
    strtouppercase(l_str);
    strtolowercase(u_str);
    if equalstr (ol_str, u_str) and equalstr(ou_str, l_str) {
        return 0;
    } else {
        return 1;
    }
}
```

ANEXO C – Scripts em python

```
string = (’’’
0061  0041  LATIN SMALL LETTER A  LATIN CAPITAL LETTER A
0062  0042  LATIN SMALL LETTER B  LATIN CAPITAL LETTER B
0063  0043  LATIN SMALL LETTER C  LATIN CAPITAL LETTER C
0064  0044  LATIN SMALL LETTER D  LATIN CAPITAL LETTER D
0065  0045  LATIN SMALL LETTER E  LATIN CAPITAL LETTER E
0066  0046  LATIN SMALL LETTER F  LATIN CAPITAL LETTER F
0067  0047  LATIN SMALL LETTER G  LATIN CAPITAL LETTER G
0068  0048  LATIN SMALL LETTER H  LATIN CAPITAL LETTER H
0069  0049  LATIN SMALL LETTER I  LATIN CAPITAL LETTER I
006A  004A  LATIN SMALL LETTER J  LATIN CAPITAL LETTER J
006B  004B  LATIN SMALL LETTER K  LATIN CAPITAL LETTER K
006C  004C  LATIN SMALL LETTER L  LATIN CAPITAL LETTER L
006D  004D  LATIN SMALL LETTER M  LATIN CAPITAL LETTER M
006E  004E  LATIN SMALL LETTER N  LATIN CAPITAL LETTER N
006F  004F  LATIN SMALL LETTER O  LATIN CAPITAL LETTER O
0070  0050  LATIN SMALL LETTER P  LATIN CAPITAL LETTER P
0071  0051  LATIN SMALL LETTER Q  LATIN CAPITAL LETTER Q
0072  0052  LATIN SMALL LETTER R  LATIN CAPITAL LETTER R
0073  0053  LATIN SMALL LETTER S  LATIN CAPITAL LETTER S
0074  0054  LATIN SMALL LETTER T  LATIN CAPITAL LETTER T
0075  0055  LATIN SMALL LETTER U  LATIN CAPITAL LETTER U
0076  0056  LATIN SMALL LETTER V  LATIN CAPITAL LETTER V
0077  0057  LATIN SMALL LETTER W  LATIN CAPITAL LETTER W
0078  0058  LATIN SMALL LETTER X  LATIN CAPITAL LETTER X
0079  0059  LATIN SMALL LETTER Y  LATIN CAPITAL LETTER Y
007A  005A  LATIN SMALL LETTER Z  LATIN CAPITAL LETTER Z

//Restante da tabela
’’’)

result = []
for line in string.strip().split(’\n’):
    parts = line.split()
    utf8_1 = f"0x{parts[0]}"
    utf8_2 = f"0x{parts[1]}"
    result.append(f" {utf8_1}: {utf8_2},")

print(’\n’.join(result))
```

```
import os

mapping = {
0x0061: 0x0041,
```

```
 0x0062: 0x0042,
 0x0063: 0x0043,
 0x0064: 0x0044,
 0x0065: 0x0045,
 0x0066: 0x0046,

//Restante do mapeamento

 0xFF55: 0xFF35,
 0xFF56: 0xFF36,
 0xFF57: 0xFF37,
 0xFF58: 0xFF38,
 0xFF59: 0xFF39,
 0xFF5A: 0xFF3A,
}

def generate_python_files(mapping, pairs_per_file=3, output_dir="output"):

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    sorted_mapping = list(mapping.items())

    files = []

    num_files = len(sorted_mapping) // pairs_per_file
    if len(sorted_mapping) % pairs_per_file != 0:
        num_files += 1

    for i in range(num_files):
        files.append(([], []))

    for i in range(len(sorted_mapping)):
        file_index = (i % num_files)
        lower, upper = sorted_mapping[i]
        files[file_index][0].append(chr(lower))
        files[file_index][1].append(chr(upper))

    for i, (lower_chars, upper_chars) in enumerate(files):
        filename = os.path.join(output_dir, f"mapping_file_{i+1:03}.rob")
        with open(filename, "w", encoding="utf-8") as file:
            file.write('use string.utf8lenstr;\n')
            file.write('use string.upperlowerstr;\n')
            file.write('int32 main() {\n')
            lowercase_str = "".join(lower_chars)
            file.write(f'    ol_str = "{lowercase_str}";\n')
            uppercase_str = "".join(upper_chars)
```

```
52              file.write(f'    ou_str = "{uppercase_str}";\n')
53              lowercase_str = "".join(lower_chars)
54              file.write(f'    l_str = "{lowercase_str}";\n')
55              uppercase_str = "".join(upper_chars)
56              file.write(f'    u_str = "{uppercase_str}";\n')
57              file.write(f'    strtouppercase(l_str);\n')
58              file.write(f'    strtolowercase(u_str);\n')
59              file.write('    if equalstr (ol_str, u_str) and equalstr(ou_str, l_str) {\
    n')
60              file.write('        return 0;\n')
61              file.write('    } else {\n')
62              file.write('        return 1;\n')
63              file.write('    }\n')
64              file.write('}\n')
65
66 generate_python_files(mapping, output_dir="robtests")
```