

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

Luiz Eduardo Rezende Moraes

**Inferência de Modelos de Inteligência Artificial na
Robotics Language usando TensorFlow Lite Micro para
TinyML**

Jataí-Goiás

Dezembro de 2025

Luiz Eduardo Rezende Moraes

Inferência de Modelos de Inteligência Artificial na Robotics Language usando TensorFlow Lite Micro para TinyML

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

Dezembro de 2025

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

Computação, Aluno do Curso de Ciência da
Trabalho Final do Curso de Ciência da Computação / Aluno do
Curso de Ciência da Computação. - 2023.
VXXIV, 74 f.

Orientador: Prof. Dr. Professor do Curso de Ciência da
Computação.
Trabalho de Conclusão de Curso (Graduação) - Universidade
Federal de Jataí, Instituto de Ciências Exatas e Tecnológicas, Ciência
da Computação, Jataí, 2023.
Bibliografia. Anexos. Apêndice.
Inclui siglas, gráfico, tabelas, lista de figuras, lista de tabelas.

1. Palavra 1. 2. Palavra 2. 3. Palavra 3. I. Computação, Professor
do Curso de Ciência da, orient. II. Título.

CDU 004

Instituto de Ciências Exatas e Tecnológicas – ICET
Curso de Ciência da Computação

Documento a ser gerado pelo SEI, e incluído como PDF no corpo da monografia.

Informações: Orientando, Título, Orientador, Avaliador 1 e Avaliador 2

Jataí-Goiás
2023

Lista de ilustrações

Figura 1 – Esquema simplificado de um compilador. Adaptado de Cooper (2013).	13
Figura 2 – Compilador dividido em duas etapas. Adaptado de Cooper (2013).	14
Figura 3 – Etapas do <i>frontend</i> de um compilador. Adaptado de Fischer, Cytron e LeBlanc (2010).	14
Figura 4 – RE para reconhecer operações aritméticas entre duas variáveis, como $a + b \times c$. Adaptado de Cooper (2013).	16
Figura 5 – Regras de derivação de uma CFG. Adaptado de Cooper (2013).	17
Figura 6 – Árvore sintática de uma CFG. Adaptado de Cooper (2013).	17
Figura 7 – Exemplo de LLVM-IR de um código “hello world”. Adaptado de The LLVM Project (2025).	19
Figura 8 – Microcontrolador de 8 bits ATmega328PB, anteriormente produzido pela Atmel, atualmente fabricado pela Microchip Technology como parte da família AVR. Possui 32KB de memória Flash, 2KB de SRAM e 27 portas de entrada/saída Microchip Technology Inc. (2025).	21
Figura 9 – Placa de desenvolvimento com o MCU STM32F103C8T6, que possui 72 MHz de frequência máxima, 20 kB de memória SRAM, 128 kB de memória FLASH e 37 portas de entrada/saída (STMicroelectronics, 2023).	21
Figura 10 – Exemplo de código com a nova sintaxe para inferência de um modelo tfLite.	32
Figura 11 – Ilustração da arquitetura de compilação e interoperabilidade. A biblioteca estática (.a) do TFLM, o arquivo objeto (.o) do wrapper C++, e o arquivo objeto (.o) do programa em Robotics Language são processados de forma independente e, na etapa final, unificados pelo linker. O linker resolve as referências entre os componentes para criar o executável único.	34

Lista de tabelas

Tabela 1 – Comparativo entre trabalhos	29
--	----

Lista de abreviaturas e siglas

AF	Autômato Finito
ARM	Advanced RISC Machine
AST	Árvore Sintática Abstrata
AVR	Alf and Vegard's RISC processor
CFG	Context-Free Grammar
CPU	Central Processing Unit
IA	Inteligência Artificial
IoT	Internet of Things
IR	Intermediate Representation
MCU	Microcontroller Unit
ML	Machine Learning
RAM	Random access memory
RE	Regular Expression
RL	Robotics Language
TFLM	TensorFlow Lite Micro

Sumário

1	Introdução	9
	INTRODUÇÃO	9
1.1	MOTIVAÇÃO (OBJETO DE ESTUDO E PROBLEMA)	9
1.2	OBJETIVO DO TRABALHO	12
1.3	CONTRIBUIÇÃO DO TRABALHO	12
2	Referencial Teórico	13
2.1	INTRODUÇÃO	13
2.2	COMPILADOR	13
2.3	FRONTEND	14
2.3.1	Analizador Léxico	15
2.3.2	Analizador Sintático	15
2.3.3	Análise Semântica	17
2.3.4	Geração de Código Intermediário	18
2.4	BACKEND	19
2.5	SISTEMAS EMBARCADOS	19
2.6	MICROCONTROLADORES	20
2.7	INTELIGÊNCIA ARTIFICIAL	22
2.7.1	Engines de Inferência	23
2.7.2	Engines para o TinyML	23
2.7.3	TensorFlow Lite Micro	24
2.8	INTEROPERABILIDADE ENTRE LINGUAGENS DE PROGRAMAÇÃO	25
2.9	ROBOTICS LANGUAGE AND COMPILER	26
3	Trabalhos relacionados	28
3.1	INTRODUÇÃO	28
3.2	CRITÉRIOS DE BUSCA	28
3.3	METODOLOGIA DE ANÁLISE	28
3.3.1	Critério 1 (C1)	28
3.3.2	Critério 2 (C2)	28
3.3.3	Critério 3 (C3)	28
3.3.4	Critério 4 (C4)	28
3.4	TRABALHOS ANALISADOS	28
3.4.1	Trabalho 1 (T1)	28
3.4.2	Trabalho 2 (T2)	28
3.4.3	Trabalho 3 (T3)	28
3.5	RESUMO COMPARATIVO	28
4	Metodologia	30
4.1	INTRODUÇÃO	30

4.2	CLASSIFICAÇÃO DA PESQUISA	30
4.3	MATERIAIS	30
4.3.1	Software	30
4.3.2	Hardware	31
4.4	PROJETO E ARQUITETURA DA SOLUÇÃO	31
4.5	PROCEDIMENTOS DE VALIDAÇÃO FUNCIONAL	32
4.6	VALIDAÇÃO DA EFICIÊNCIA E ANÁLISE COMPARATIVA	32
5	Implementação	34
5.1	INTRODUÇÃO	34
5.2	ARQUITETURA DA CAMADA DE INTEROPERABILIDADE	34
5.3	IMPLEMENTAÇÃO DO WRAPPER C	35
5.4	ADAPTAÇÃO DO FRONTEND DO COMPILADOR	35
5.5	CONSIDERAÇÕES FINAIS	35
6	Funcionamento ou Demonstração	36
6.1	INTRODUÇÃO	36
6.2	EXEMPLO DE APLICAÇÃO	36
6.3	CONSIDERAÇÕES FINAIS	36
7	Avaliação e Testes	37
7.1	INTRODUÇÃO	37
7.2	AMBIENTE EXPERIMENTAL	37
7.3	VALIDAÇÃO FUNCIONAL	37
7.4	VALIDAÇÃO DA EFICIÊNCIA E ANÁLISE COMPARATIVA	37
7.5	CONSIDERAÇÕES FINAIS	37
8	Conclusões e Trabalhos Futuros	38
8.1	INTRODUÇÃO	38
8.2	CONCLUSÕES	38
8.2.1	Quanto à área aplicada	38
8.2.2	Quanto à área específica	38
8.3	TRABALHOS FUTUROS	38
8.4	CONSIDERAÇÕES FINAIS	38
	Referências	39
	Anexos	42
	ANEXO A Exemplo de Formulário	43

1 Introdução

1.1 Motivação (objeto de estudo e problema)

O Machine Learning (ML) está se tornando cada vez mais comum em diversas áreas da sociedade, como em veículos autônomos, reconhecimento facial e monitoramento de saúde. Esse recurso inovador pode estar presente não somente em supercomputadores, celulares e robôs, mas também em sistemas embarcados. Aplicações nessas plataformas já são uma realidade, como apresentado por [Hoang, Nam e Park \(2019\)](#), que implementou um modelo de Inteligência Artificial (IA) para melhorar a identificação de marcações viárias em carros autônomos, e por [Attaran et al. \(2018\)](#), que utilizou circuitos dedicados e plataformas embarcadas para detectar o estresse humano. No entanto, a adoção de ML em microcontroladores de baixo custo ainda é um campo pouco explorado, devido a limitações de *hardware* e falta de ferramentas intuitivas para o desenvolvimento.

Ao contrário do que se imagina, microcontroladores (MCU, do inglês *Microcontroller Unit*) podem ser encontrados em diversos aparelhos cotidianos como eletrodomésticos, automóveis, dispositivos de saúde e lâmpadas inteligentes. Qualquer dispositivo que meça, armazene, controle, calcule ou exiba informações é um candidato a ter um microcontrolador embutido ([AXELSON, 1997](#)). Eles podem ser definidos como um computador em um único circuito integrado que inclui uma *Central Processing Unit* (CPU), *Random Access Memory* (RAM), alguma forma de *Read-Only Memory* (ROM) e portas de *Input/Output* (I/O) ([HUSSAIN et al., 2016](#)). Frequentemente, o termo pode ser utilizado como sinônimo para sistema embarcado, porém existe uma distinção clara entre os dois conceitos. Enquanto o MCU refere-se apenas ao *hardware*, o sistema embarcado compreende o sistema completo (*hardware + firmware*), já pronto para uso.

O uso de sistemas embarcados também está fortemente ligado à ideia de *Internet of Things* (IoT), uma rede de dispositivos eletrônicos de baixo custo onde a comunicação e a coleta de dados ocorrem automaticamente, por meio de protocolos de comunicação ([WU; QIU; ZHANG, 2020](#)). Atualmente, esse tipo de sistema já é amplamente utilizado para captura e processamento de dados em projetos de IoT como cidades inteligentes, automação residencial e agricultura de precisão. Quando o projeto envolve IA, após a captura dos dados, eles devem ser enviados para essa IA para uma tomada de decisão. Isso pode ocorrer de duas formas: utilizando *Cloud AI* ou *Edge AI*.

Edge AI (Inteligência Artificial na Borda) refere-se à prática de realizar computações de IA próximo aos usuários na borda da rede, em vez de em localizações centralizadas como os *data centers* de provedores de serviços em nuvem (*Cloud AI*) ([SINGH; GILL, 2023](#)). Segundo [Singh e Gill \(2023\)](#), *Edge AI* se destaca em situações que precisam de

baixa latência, privacidade e eficiência energética. Já a *Cloud AI* é mais indicada para tarefas que exigem um grande poder de processamento e muito espaço de armazenamento. O uso de ML em MCUs se enquadra no conceito de IA na borda, mas em um campo ainda mais específico e emergente: o *Tiny Machine Learning* (TinyML).

De acordo com [Warden e Situnayake \(2019\)](#), “Se você consegue executar um modelo de rede neural com um custo energético abaixo de 1 mW, isso torna possíveis muitas aplicações completamente novas”. Essa é a principal premissa e também definição do termo TinyML, desse modo qualquer aplicação que consiga executar um modelo a um custo de energia baixíssimo pode ser considerada como TinyML. Esse número pode parecer arbitrário (1mW), mas, em termos concretos, significa que um dispositivo alimentado por uma bateria de botão pode funcionar por um ano. Isso resulta em um produto que é pequeno o suficiente para ser instalado em qualquer ambiente e capaz de operar por um tempo útil sem intervenção humana ([WARDEN; SITUNAYAKE, 2019](#)).

Para a execução de TinyML, além dos microcontroladores convencionais, podem ser utilizados outros circuitos integrados (CI), como ASICs (*Application-Specific Integrated Circuits*), FPGAs (*Field-Programmable Gate Arrays*) e DSPs (*Digital Signal Processors*). Apesar da existência de diferentes opções, os MCUs mostram-se mais vantajosas por geralmente serem menores, apresentarem baixíssimo consumo de energia e possuírem custo extremamente baixo. Embora o *hardware* especializado proporcione o melhor desempenho para TinyML, microcontroladores de propósito geral oferecem maior flexibilidade, como discutido por [Abadade et al. \(2023\)](#).

Devido à falta de poder computacional, executar modelos de ML em sistemas embarcados não é uma tarefa fácil; no entanto, segundo [Soro \(2021\)](#), existem três formas principais de realizar esse feito:

- Codificação Manual, implementação manual do modelo de ML diretamente em código de baixo nível, como C ou C++. É uma maneira trabalhosa, demorada e propensa a erros, devido ao grau de complexidade.
- Geração automática de código, uso de ferramentas para converter um modelo pré-treinado em código otimizado para o dispositivo embarcado. A ferramenta converte, por exemplo, um arquivo .tflite em código C ou C++.
- Interpretador de ML, é utilizado um interpretador para executar o modelo diretamente no dispositivo.

Várias ferramentas para TinyML já existem no mercado, algumas são produzidas por grandes empresas como a Google, outras são desenvolvidas por grupos menores de desenvolvedores. Cada uma delas possui em sua estrutura uma técnica de geração

automática de código ou um interpretador de ML. O objetivo de um *framework* TinyML é fornecer uma solução abrangente para a construção e implantação de modelos de ML em dispositivos de baixo consumo de energia, facilitando o desenvolvimento de aplicações de *edge computing* por desenvolvedores (ABADADE et al., 2023).

Desenvolvido pela Google, o TensorFlow Lite¹ foi lançado em 2017 e se destacou como um dos pioneiros em *Edge AI*, tendo grande importância na democratização do ML em dispositivos limitados. Atualmente, seu nome foi alterado para LiteRT², um produto oficial da Google AI Edge, uma mudança estratégica da empresa para unificar ferramentas de ML no dispositivo. Agora, o LiteRT consegue cobrir múltiplos *frameworks* de treinamento, como JAX³, Keras⁴, PyTorch⁵ e TensorFlow. Apesar das atualizações no nome e na amplitude do produto, a ferramenta oficial da Google para desenvolvimento de TinyML em microcontroladores teve seu nome mantido como TensorFlow Lite Micro (TFLM).

Conforme o trabalho de David et al. (2021), o *framework* trabalha com alocação estática de memória, possui um interpretador de ML com menos de 2 kB e já foi validado em diversas arquiteturas, como Arm Cortex-M, ESP32 e DSPs. Sua abordagem baseada em interpretador (que lê o modelo .tflite e executa as operações nele contidas) viabiliza a atualização de modelos “em campo”, ou seja, em dispositivos que já estão em posse do usuário final ou instalados em seu local de operação. Isso ocorre porque o *firmware* pode ser programado para carregar o modelo de uma área de memória separada, como uma partição Flash, permitindo a substituição do arquivo do modelo sem a necessidade de recompilar o *firmware*.

No contexto dos sistemas embarcados, o projeto Robotics Language (PI05974-2024), realizado na Universidade Federal de Jataí (UFJ), desenvolve uma linguagem de programação especializada no campo da robótica e dos microcontroladores. Essa iniciativa é uma evolução de um projeto anterior que lidava com kits robóticos educacionais (PI02361-2018) e busca, de maneira geral, tornar a programação em microcontroladores mais simples e acessível.

Para realizar tal objetivo, a linguagem busca isolar as especificidades de microcontroladores, fornecendo uma camada de abstração de *hardware* dentro do próprio compilador, em vez de depender de *frameworks* ou bibliotecas externas. Essa abordagem permite que os desenvolvedores escrevam o código uma única vez, eliminando a necessidade de adaptar o código para cada *hardware* através de macros condicionais. Ao aproveitar a análise semântica do compilador, é possível prevenir erros comuns encontrados no desenvolvimento

¹ Documentação do TensorFlow Lite: <<https://www.tensorflow.org/lite/guide?hl=pt-br>>

² Documentação do Lite RT: <<https://ai.google.dev/edge/litert?hl=pt-br>>

³ Documentação do JAX: <<https://docs.jax.dev/en/latest/>>

⁴ Site do Keras: <<https://keras.io>>

⁵ Site do PyTorch: <<https://pytorch.org>>

de *firmware* em linguagens não específicas de domínio, como C/C++ (OLIVEIRA, 2018).

1.2 Objetivo do Trabalho

Este trabalho teve como objetivo integrar o TFLM à linguagem, estendendo sua sintaxe para viabilizar a execução de modelos de ML nos microcontroladores já compatíveis com o *framework* e suportados pela linguagem. Os objetivos específicos foram:

- Adicionar novas palavras-chave, regras sintáticas e semânticas na linguagem RL para a inferência de modelos de TinyML;
- Implementar um *wrapper* parcial da API TFLM em C, para possibilitar a ligação do TFLM com o código compilado em RL;
- Validar a integração por meio de testes funcionais com modelos de IA existentes e já treinados; e
- Realizar uma análise comparativa de eficiência entre a solução em RL e uma implementação nativa em C++.

1.3 Contribuição do Trabalho

A principal contribuição deste trabalho é a extensão da RL para viabilizar o desenvolvimento de aplicações de ML em microcontroladores de forma nativa. Ao integrar o *framework* TFLM, este projeto dota a linguagem de uma sintaxe de alto nível que abstrai a complexidade da inferência. Dessa forma, os desenvolvedores podem aliar o poder do TinyML aos benefícios já oferecidos pela RL, como o desenvolvimento de firmware com baixo acoplamento, alta coesão e elevada manutenibilidade.

2 Referencial Teórico

2.1 Introdução

Para facilitar a compreensão deste projeto de pesquisa, este capítulo apresenta as definições e conceitos que serão utilizados ao longo do trabalho.

2.2 Compilador

Compiladores são programas com uma função aparentemente simples: traduzir código de uma linguagem para outra. No entanto, por trás dessa definição sucinta, está um dos sistemas mais complexos da computação, formado por muitos componentes internos, algoritmos e interações complexas entre eles (COOPER, 2013).

Inicialmente, o compilador pode ser visto como uma caixa-preta que recebe como entrada um código fonte (ou programa fonte) e o transforma em um código objeto (ou programa objeto). A Figura 1 ilustra esse esquema simplificado, que ainda desconsidera as etapas internas do processo de compilação. O código fonte pode ser escrito em diversas linguagens, como C, C++, Fortran, Java ou até mesmo a RL com seu compilador Robcmp. Já a linguagem do código objeto corresponde ao conjunto de instruções de uma CPU ou MCU específico.



Figura 1 – Esquema simplificado de um compilador. Adaptado de Cooper (2013).

É essencial conhecer as etapas envolvidas no processo de um compilador. As duas principais divisões existentes são: *frontend* e *backend*. Essa divisão de tarefas pode ser visualizada na Figura 2, que sintetiza a estrutura do processo de compilação. O *frontend* do compilador é responsável por analisar o programa fonte, incluindo o processamento léxico, sintático e semântico, resultando em uma representação intermediária, como uma Árvore Sintática Abstrata (AST, do inglês *Abstract Syntax Tree*) ou um código intermediário. O *backend* recebe essa representação intermediária e gera código alvo otimizado, lidando com detalhes específicos da máquina, como alocação de registradores e seleção de instruções (FISCHER; CYTRON; LEBLANC, 2010).

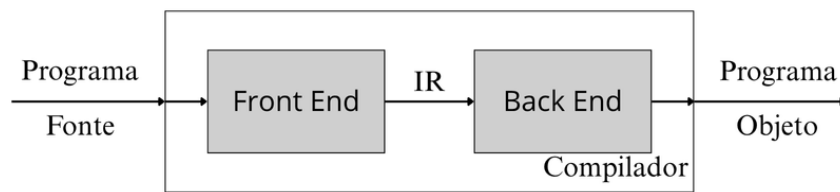


Figura 2 – Compilador dividido em duas etapas. Adaptado de Cooper (2013).

2.3 Frontend

Para que o compilador consiga traduzir o código do programa fonte, ele precisa entender tanto a forma (sintaxe) quanto o significado (semântica). O *frontend* é a primeira etapa da compilação e é responsável por determinar se o código está bem construído em termos de sintaxe e semântica. Se ele identificar um código válido, uma representação intermediária é criada; caso contrário, reporta o erro ao usuário para que o problema seja identificado (COOPER, 2013).

A Figura 3 ilustra as principais etapas do *frontend* consideradas neste estudo. Inicialmente, o código-fonte é submetido ao analisador léxico, que converte a sequência de caracteres em *tokens* (unidades significativas). Em seguida, o analisador sintático verifica a estrutura desses *tokens* e gera uma AST. Essa AST é então utilizada pelo analisador semântico para produzir uma Árvore Sintática Decorada, que finalmente é processada para gerar a Representação Intermediária (IR, do inglês *Intermediate Representation*). Todas essas etapas utilizam a tabela de símbolos, seja para armazenar novas entradas ou para verificar a existência de símbolos já declarados.

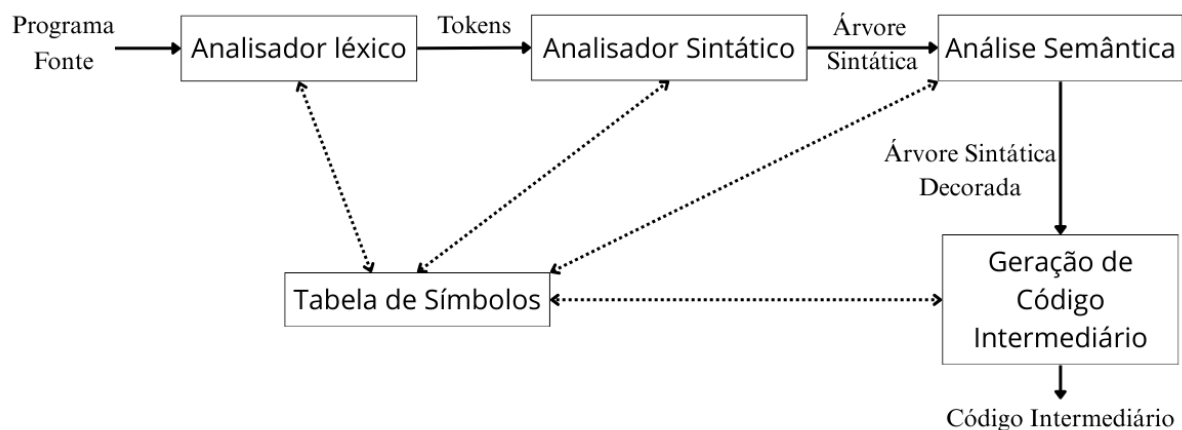


Figura 3 – Etapas do *frontend* de um compilador. Adaptado de Fischer, Cytron e LeBlanc (2010).

2.3.1 Analisador Léxico

A primeira etapa do *frontend* consiste em um analisador Léxico, ou *Scanner* em inglês. Ele inicia a análise do programa fonte lendo o texto de entrada (caractere por caractere) e agrupando caracteres individuais em *tokens*, como identificadores, números inteiros, palavras reservadas e delimitadores (FISCHER; CYTRON; LEBLANC, 2010, p. 16). No analisador léxico, autômatos finitos (AFs) atuam como núcleo do processo de reconhecimento de *tokens*. Eles simulam um fluxo de transições entre estados, guiados pelos caracteres de entrada, até determinar se uma sequência é válida (aceita) ou inválida (rejeitada) (COOPER, 2013, p. 23).

O conjunto de palavras aceitas por um autômato finito, F , forma uma linguagem, indicada por $L(F)$. O diagrama de transição do AF especifica, em detalhes precisos, esta linguagem. Para qualquer AF, também podemos descrever sua linguagem usando uma notação chamada expressão regular (RE, do inglês *Regular Expression*). A linguagem descrita por uma RE é chamada de linguagem regular (COOPER, 2013, p. 26). Tanto um AF quanto uma RE podem ser usados em um programa gerador de *scanner*, um programa que produz efetivamente um analisador léxico funcional. Geradores de analisadores léxicos são ferramentas valiosas para a construção de compiladores (FISCHER; CYTRON; LEBLANC, 2010, p. 16).

2.3.2 Analisador Sintático

A principal tarefa do analisador sintático (ou *parser*, em inglês) é verificar se o programa de entrada constitui uma sentença sintaticamente válida na linguagem-fonte. Para essa finalidade, essa etapa utiliza gramáticas livres de contexto, uma vez que REs não são mais suficientes para descrever a sintaxe complexa presente na maioria das linguagens de programação (COOPER, 2013, p. 69, 70).

Apesar de uma RE, como a apresentada na Figura 4, poder reconhecer a expressão $a + b \times c$ corretamente, ela não pode especificar a ordem de precedência entre os operadores. Conforme as regras algébricas convencionais, as operações de multiplicação e divisão possuem precedência sobre adição e subtração. Essa ordem de avaliação pode também ser alterada utilizando parênteses como mecanismo de agrupamento. A simples inclusão de parênteses em REs não resolve adequadamente o problema de precedência, pois é impossível definir uma RE que reconheça todas as possíveis combinações de parênteses balanceados (COOPER, 2013, p. 71). Em outras palavras, a linguagem reconhecida por uma RE não é suficiente para o reconhecimento sintático do compilador.

$$[a\dots z]([a\dots z] \mid [0\dots 9])^* ((+ \mid - \mid \times \mid \div) [a\dots z]([a\dots z] \mid [0\dots 9])^*)^*$$

Figura 4 – RE para reconhecer operações aritméticas entre duas variáveis, como $a + b \times c$. Adaptado de Cooper (2013).

Em vez de utilizar REs para verificar sentenças, o analisador sintático emprega uma gramática livre de contexto (CFG, do inglês *Context-Free Grammar*), um conjunto de regras que descreve como as sentenças podem ser formadas. A coleção de todas as sentenças deriváveis a partir de G é denominada linguagem definida por G , representada por $L(G)$ (COOPER, 2013, p. 71). Esse mecanismo demonstra maior poder expressivo e complexidade na especificação de construções sintáticas, uma vez que, diferentemente das REs, é capaz de:

- Validar construções com parênteses balanceados; A regra $\text{Expr} \rightarrow (\text{Expr})$ na Figura 5 demonstra essa capacidade, permitindo que expressões sejam aninhadas recursivamente dentro de parênteses. O analisador sintático utiliza essa regra para garantir que, para cada parêntese de abertura, exista um correspondente de fechamento.
- Incorporar naturalmente regras de precedência de operadores, como ilustrado pelas regras de derivação na Figura 5. Estas regras servem como blocos de construção para expressões. Através da criação de diferentes níveis hierárquicos para os operadores (algo que esta gramática simplificada não faz, mas que as CFGs permitem), é possível forçar o analisador a reconhecer a precedência correta, como avaliar a multiplicação antes da adição.
- Representar hierarquias sintáticas através de árvores de derivação, conforme exemplificado na Figura 6 para a expressão $a + b \times c$. Nesta figura, a árvore ilustra a estrutura $(a + b) \times c$. Isso ocorre porque a regra $\text{Expr} \rightarrow \text{Expr Op nome}$ força uma avaliação da esquerda para a direita (associatividade à esquerda). A árvore mostra a operação de adição $a + b$ sendo formada em um nível mais baixo e, em seguida, o resultado dessa subárvore é usado como operando para a multiplicação por c , que está no topo da árvore.

1	<i>Expr</i>	→	(<i>Expr</i>)
2			<i>Expr Op nome</i>
3			nome
4	<i>Op</i>	→	+
5			-
6			×
7			÷

Figura 5 – Regras de derivação de uma CFG. Adaptado de Cooper (2013).

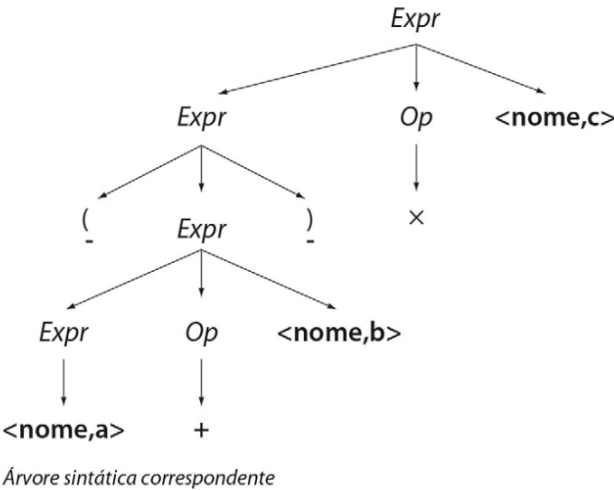


Figura 6 – Árvore sintática de uma CFG. Adaptado de Cooper (2013).

Como resultado da análise sintática, constrói-se uma AST utilizando as informações obtidas do programa fonte e do analisador léxico, assegurando a validação da estrutura sintática do código. Simultaneamente, a tabela de símbolos é atualizada com metadados essenciais sobre cada identificador, incluindo seu tipo de dado, o tamanho de sua representação em memória durante a execução e, no caso de *arrays*, o número de dimensões e os intervalos indexados de cada dimensão (COOPER, 2013, p. 176).

2.3.3 Análise Semântica

A Análise Semântica, como o próprio nome indica, tem como objetivo examinar o significado (semântica) do programa fonte. Para isso, ela utiliza a AST construída pelo Analisador Sintático, juntamente com as informações contidas na tabela de símbolos. Uma das principais funções da análise semântica é a verificação de tipos, na qual o compilador verifica a compatibilidade entre os operandos de cada operador (AHO et al., 2008).

Entre as regras semânticas mais comuns destacam-se:

- A exigência de que índices de *arrays* sejam do tipo inteiro;
- A restrição de que operandos em operações aritméticas possuam tipos compatíveis;
- A proibição de redeclaração de variáveis no mesmo escopo;
- A verificação de que expressões condicionais retornem valores booleanos.

No final dessa fase, temos como resultado a Árvore Sintática Decorada, uma versão aprimorada da AST original. A grande diferença é que agora ela traz consigo uma informação importante: os tipos de dados de todos os identificadores usados no programa. Essas informações adicionais são vitais para as fases seguintes, especialmente para a geração de código, onde é necessário saber exatamente qual é o tipo de cada variável e expressão.

2.3.4 Geração de Código Intermediário

Durante todo o processo de compilação, podem ser produzidas uma ou mais IRs, as quais podem assumir diversas formas. As árvores sintáticas constituem um tipo de IR, assim como outros formatos de código em baixo nível ou semelhantes à linguagem de máquina. Esses códigos intermediários devem possuir duas propriedades essenciais: precisam ser facilmente gerados e simples de traduzir para a linguagem da máquina alvo (AHO et al., 2008).

O Robcmp utiliza o *backend* da infraestrutura LLVM; por essa razão, sua IR segue o padrão LLVM-IR. O LLVM consiste em uma coleção modular e reutilizável de tecnologias de compilador, que proporciona um otimizador moderno e independente de alvo, além de suporte para a geração de código para diversas CPUs populares.

A representação de código do LLVM foi projetada para ser utilizada de três formas distintas: como uma IR de compilador em memória, como uma representação em bitcode armazenada em disco (adequada para carregamento rápido por um compilador *Just-In-Time*) e como uma linguagem de montagem de alto nível, legível por humanos (The LLVM Project, 2025).

A Figura 7 ilustra uma IR que realiza três operações sequenciais: (1) define uma variável contendo a *string* “hello world”, (2) declara a função externa “puts”, responsável por imprimir *strings* na saída padrão, e (3) implementa a função “main”, que invoca “puts” para exibir a mensagem armazenada.

```
; 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 @captures(none)) 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 7 – Exemplo de LLVM-IR de um código “hello world”. Adaptado de [The LLVM Project \(2025\)](#).

2.4 Backend

O papel do *backend* é receber o código IR e produzir um código para a máquina-alvo. Ele seleciona as operações específicas da máquina-alvo para implementar cada operação da IR, determina a ordem mais eficiente de execução dessas operações e decide quais valores serão armazenados em registradores e quais residirão na memória (COOPER, 2013, p. 13). Essas decisões podem ser ajustadas pelo desenvolvedor por meio de flags de otimização, como -Oz (otimização para tamanho extremo) e -Os (otimização para tamanho).

Uma das vantagens do *backend* da LLVM é seu suporte a diversas plataformas, incluindo CPUs (como x86 e ARM) e MCUs (como as famílias AVR e ARM Cortex-M). Isso permite que a RL também tenha portabilidade para um amplo espectro de *hardware*. Por exemplo, é possível compilar código para microcontroladores específicos como o ATmega328P (da família AVR) ou o STM32f103c8t6 (baseado em ARM Cortex-M). Estes microcontroladores frequentemente compõem sistemas embarcados, como será detalhado a seguir.

2.5 Sistemas Embarcados

Segundo Marwedel (2021), os sistemas embarcados são sistemas de processamento de informação embutidos nos produtos que os contêm, ou seja, são parte de um sistema maior. Atualmente, estão presentes em inúmeras aplicações, como carros, trens, aviões, equipamentos industriais e de telecomunicações.

Apesar da definição abrangente, é possível obter muitas informações a partir

dessa descrição. Como um sistema embarcado faz parte de um sistema maior, ele é frequentemente dedicado a uma tarefa específica. Desse modo, processadores que controlam determinados sistemas de carros ou trens, por exemplo, sempre executam o mesmo *software* (MARWEDEL, 2021, p. 17). Esse *software*, por ser o programa fundamental que gerencia o *hardware* para essas funções específicas, é corretamente classificado como *firmware*.

Além de serem parte fundamental de dispositivos individuais, esses componentes também formam o núcleo de aplicações de IoT. De acordo com Giusto et al. (2010), o termo *Internet of Things* é um paradigma que abrange objetos físicos que se conectam à Internet para compartilhar informações. Tais informações podem ser capturadas por sensores, processadas por um microcontrolador e repassadas a outros dispositivos, a fim de que atinjam um objetivo em comum.

Existe um grande potencial para aplicações de processamento de informações no contexto de IoT e sistemas embarcados. Sua enorme abrangência inclui áreas como eletrônica automotiva, aviônica, transporte ferroviário, engenharia marítima, engenharia mecânica, robótica, engenharia civil, recuperação de desastres, engenharia agrícola, setor de saúde, entre outras (MARWEDEL, 2021).

Para que o processamento de informações ocorra, os dados devem ser processados por uma CPU. Nos sistemas embarcados, MCUs são frequentemente empregados para essa tarefa. A CPU integrada a um MCU não só permite executar todas as operações necessárias, como também possibilita que o sistema se beneficie das vantagens inerentes a um microcontrolador.

2.6 Microcontroladores

O núcleo da maioria dos Sistemas Embarcados é o seu centro de processamento, constituído por um MCU. Segundo Hussain et al. (2016), microcontroladores são computadores em um único chip: um circuito que integra uma unidade central de processamento, memória de acesso aleatório, alguma forma de memória apenas de leitura e portas de entrada/saída.

Ao contrário dos computadores de uso geral (como computadores pessoais, *smartphones* e *tablets*), os MCUs são dedicados a tarefas específicas e executam uma única aplicação (HUSSAIN et al., 2016). O MCU é um componente eletrônico único; já os sistemas embarcados são o conjunto completo, que inclui, além do próprio MCU, outros elementos como sensores, atuadores, fontes de alimentação e interfaces de comunicação, todos trabalhando em conjunto para realizar uma função determinada.

Algumas das famílias de microcontroladores mais comuns e que também possuem microcontroladores suportados pela RL são:

- AVR: É uma família de microcontroladores de 8 bits, originalmente desenvolvida pela Atmel e atualmente produzida pela Microchip Technology. Estes microcontroladores combinam desempenho eficiente com baixo consumo de energia, oferecendo flexibilidade para uma vasta gama de aplicações embarcadas ([Microchip Technology Inc., 2025](#)). A [Figura 8](#) apresenta um exemplo de MCU com essa arquitetura.
- ARM: A família de processadores Arm Cortex-M também é uma série de chips otimizados para eficiência energética e operações determinísticas. Trata-se de uma arquitetura amplamente utilizada em MCUs de 32 bits, uma alternativa atraente às arquiteturas de MCU de propósito geral de 8 bits e 16 bits ([STMicroelectronics, 2025](#)). A [Figura 9](#) mostra uma placa de desenvolvimento que contém um MCU dessa família.

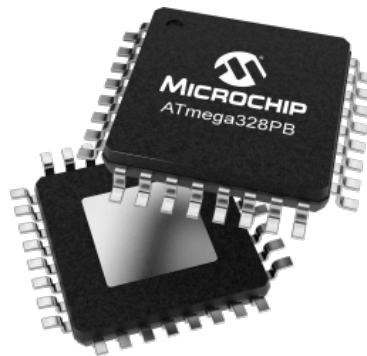


Figura 8 – Microcontrolador de 8 bits ATmega328PB, anteriormente produzido pela Atmel, atualmente fabricado pela Microchip Technology como parte da família AVR. Possui 32KB de memória Flash, 2KB de SRAM e 27 portas de entrada/saída ([Microchip Technology Inc. \(2025\)](#)).

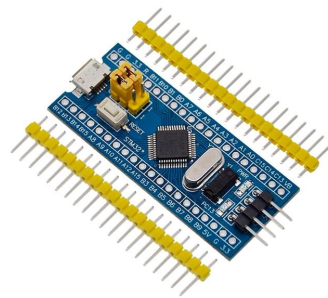


Figura 9 – Placa de desenvolvimento com o MCU STM32F103C8T6, que possui 72 MHz de frequência máxima, 20 kB de memória SRAM, 128 kB de memória FLASH e 37 portas de entrada/saída ([STMicroelectronics, 2023](#)).

2.7 Inteligência Artificial

Na última década, o assunto IA tornou-se popular e impossível de evitar em canais midiáticos como televisão, redes sociais e até mesmo em conversas cotidianas. Pode-se dizer que ele se tornou um termo *mainstream*, expressão do inglês que designa um conteúdo considerado normal que é aceito pela maioria das pessoas (Cambridge University Press, 2025). Essa tecnologia ganhou notoriedade devido ao seu notável crescimento e aplicação em diversas áreas da sociedade, consolidando-se como uma inovação promissora que marcou o início de uma nova era a partir da década de 2020.

Apesar de ter se tornado comum e de fácil compreensão em discussões informais, o conceito de IA é frequentemente confundido com ML, como se fossem sinônimos. Segundo Russell e Norvig (2020), o ramo da IA preocupa-se em estudar e construir entidades inteligentes capazes de computar como agir eficientemente em um amplo espectro de situações. Além disso, a IA atualmente abrange diversas subáreas, que vão desde capacidades gerais (como aprendizagem, raciocínio e percepção) até tarefas específicas, tais como jogar xadrez, provar teoremas matemáticos, dirigir veículos ou diagnosticar doenças. Por essa razão, a IA constitui um campo relevante para qualquer tarefa que exija raciocínio intelectual, abrangendo inúmeros domínios do conhecimento.

Enquanto a IA possui um espectro mais abrangente e um objetivo filosófico mais amplo, o ML é um subcampo da IA que estuda como melhorar o desempenho com base na experiência. Ou seja, utiliza dados para realizar um treinamento, permitindo que o sistema computacional tome decisões baseadas em experiências passadas obtidas desses dados. Alguns sistemas de IA empregam métodos de ML, mas outros não (RUSSELL; NORVIG, 2020).

O resultado de treinamentos utilizando grandes bases de dados são modelos de ML, que podem ser executados em diferentes tipos de computadores, dependendo do *hardware* requerido pelo modelo. Esses modelos podem ser executados na borda da rede (*Edge AI*), isto é, em celulares, computadores pessoais e sistemas embarcados, ou na nuvem (*Cloud AI*), isto é, em *data centers* e servidores, para que o modelo possa ser acessado via Internet.

Segundo Singh e Gill (2023), *Edge AI* se destaca em situações que precisam de baixa latência, privacidade e eficiência energética. Já a *Cloud AI* é mais indicada para tarefas que exigem um grande poder de processamento e muito espaço de armazenamento. O uso de ML em MCUs se enquadra no conceito de IA na Borda, mas em um campo ainda mais específico e emergente: o TinyML, que é o foco principal deste trabalho.

Um exemplo de modelo aplicável a TinyML é o Micro Speech¹, um modelo exemplo

¹ <https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech>

do repositório oficial do TFLM. O exemplo Micro Speech demonstra a execução de inferência para reconhecimento de palavras-chave (“sim” e “não”). O sistema consiste em dois modelos principais:

- Pré-processador de áudio: Converte amostras de áudio bruto em dados espectrográficos; e
- Modelo Micro Speech: Modelo compacto (<20 kB) que classifica as palavras-chave a partir dos espectrogramas.

Este projeto irá expandir a RL para suportar a execução de modelos de ML nos MCUs suportados pela linguagem. Dessa forma, será possível realizar inferências com maior privacidade e menor latência, uma vez que não será necessário acessar modelos hospedados em servidores na Internet, mantendo os dados estritamente no âmbito local.

2.7.1 *Engines de Inferência*

Engines de Inferência (ou Motores de Inferência) são mecanismos que permitem a execução de um modelo de ML já treinado. Na fase de inferência, esse modelo é implementado, e o motor de inferência utiliza a “inteligência” aprendida para compreender e fazer previsões ou classificações sobre dados novos e não vistos.

Geralmente, os mecanismos de inferência fazem parte de um *framework* maior, que contém ferramentas tanto para o treinamento quanto para a inferência de modelos. Entre os exemplos mais famosos estão TensorFlow, PyTorch, Caffe, Keras e JAX.

Esses *frameworks* podem ser usados para treinar desde modelos simples até os mais complexos, mas a escolha do *hardware* para o treinamento depende da necessidade de cada modelo. Enquanto modelos simples podem ser treinados utilizando computadores comuns, modelos mais complexos, como os de *Deep Learning* (Aprendizagem Profunda), são muito mais exigentes e precisam ser treinados em computadores equipados com Unidades de Processamento Gráfico (GPUs, do inglês *Graphics Processing Units*) potentes ou especializadas.

2.7.2 *Engines para o TinyML*

Considera-se TinyML qualquer aplicação capaz de executar modelos de ML com um consumo de energia extremamente baixo. Isso se alinha à premissa destacada por [Warden e Situnayake \(2019\)](#), que afirma: “Se você consegue executar um modelo de rede

neural com um custo energético abaixo de 1 mW, isso torna possíveis muitas aplicações completamente novas”, definindo assim o potencial e o escopo do TinyML.

No contexto do TinyML, diferentemente do contexto geral de ML, os *frameworks* disponíveis para o desenvolvimento possuem somente os mecanismos para inferência. Afinal, o *hardware* de dispositivos como um microcontrolador não é suficiente para o treinamento de um modelo. A fase de treinamento, nesses casos, é feita em uma máquina de desenvolvimento, para que, depois, apenas a inferência seja realizada no dispositivo de borda (SORO, 2021).

Para o desenvolvimento de TinyML em microcontroladores, há diversos *frameworks* disponíveis, incluindo TFLM² (Google), uTensor³ (ARM), MicroTVM⁴ (uTVM), STM32Cube.AI⁵ (STMicroelectronics), NanoEdge AI Studio⁶ (STMicroelectronics), Eloquent TinyML⁷, emlearn⁸ e EON compiler⁹ (Edge Impulse). Embora cada um apresente suas particularidades e suporte a diferentes famílias de MCUs, o TFLM foi selecionado para este trabalho por suas vantagens e conveniência, alinhando-se aos interesses do projeto.

Dentre as conveniências estão o suporte para a família ARM Cortex-M do Robcomp e do TFLM, a existência de modelos já treinados em formato .tflite que podem ser aproveitados, e uma licença que permite uso gratuito, modificação e distribuição, a qual pode ser encontrada no repositório do GitHub, que aponta para a licença Apache License, Version 2.0¹⁰. Dentre os interesses do projeto estão: manter o código aberto, buscar expandir para outras plataformas futuramente e o foco em portabilidade e suporte a *hardware* heterogêneo.

2.7.3 TensorFlow Lite Micro

Conforme David et al. (2021), o TFLM possui uma série de vantagens que podem ser observadas após a análise das decisões de design e implementação. Dentre essas vantagens, as mais pertinentes e em conformidade com o objetivo da RL são:

- Código Aberto: TFLM é um *framework* de inferência TinyML de código aberto.
- Abordagem Baseada em Interpretador: Utiliza um interpretador que oferece flexibili-

² Documentação do TFLM: <<https://github.com/tensorflow/tflite-micro>>

³ Documentação do uTensor: <<https://github.com/uTensor/uTensor>>

⁴ Documentação do MicroTVM: <<https://daobook.github.io/tvm/docs/topic/microtvm/index.html>>

⁵ Documentação do STM32Cube.AI: <<https://stm32ai.st.com/stm32-cube-ai/>>

⁶ Site do NanoEdge AI Studio: <<https://stm32ai.st.com/nanoedge-ai/>>

⁷ Documentação do Eloquent TinyML: <<https://github.com/eloquentarduino/EloquentTinyML>>

⁸ Documentação do emlearn: <<https://github.com/emlearn/emlearn>>

⁹ Documentação do EON Compiler: <<https://docs.edgeimpulse.com/docs/edge-impulse-studio/deployment/eon-compiler>>

¹⁰ Disponível em: <<https://www.apache.org/licenses/LICENSE-2.0>>

dade e portabilidade, facilitando a adaptação a novas aplicações e recursos. Diferente da geração de código, o interpretador permite atualizar modelos substituindo apenas o arquivo/área de memória do modelo, sem recompilar tudo.

- Independência de *hardware*: Minimiza o uso de dependências externas e requisitos de biblioteca para ser independente em relação ao *hardware*.
- Gerenciamento de Memória Eficiente: Não depende de alocação dinâmica; utiliza uma “arena” de memória fornecida pela aplicação.
- Reutilização de Ferramentas TensorFlow: Integra-se fortemente com o ambiente de treinamento do TensorFlow e reutiliza as ferramentas e *kernels* de referência do TensorFlow Lite, facilitando a conversão, otimização e garantindo um ambiente harmonizado.
- Suporte a Otimizações: Suporta quantização (como 8 bits) e outras otimizações (*folding*, remoção de *dropout*) através da *toolchain* do TensorFlow Lite.

2.8 Interoperabilidade entre linguagens de programação

Conforme [International Organization for Standardization \(ISO\)](#), [International Electrotechnical Commission \(IEC\)](#) e [Institute of Electrical and Electronics Engineers \(IEEE\)](#) (2017), interoperabilidade pode ser definida como “O grau em que dois ou mais sistemas, produtos ou componentes podem trocar informações e usar tais informações”. Em essência, a interoperabilidade é a capacidade de diferentes sistemas ou componentes de *software* trocarem informações e utilizarem as funcionalidades uns dos outros, mesmo que tenham sido desenvolvidos com tecnologias, plataformas ou linguagens de programação distintas.

A interoperabilidade pode ser alcançada de diferentes maneiras, dependendo das tecnologias ou linguagens de programação utilizadas no projeto. Ela pode ocorrer entre diferentes linguagens de programação, por exemplo, C e Python, por meio de módulos existentes, como `ctypes`¹¹ e `ffi`¹². Nesses casos, a interoperabilidade pode ser interpretada como Interface de Função Estrangeira (FFI, do inglês *Foreign Function Interface*).

No contexto deste projeto, a RL se integrará ao TFLM. Para que a RL execute as funções de inferência da API C++, será elaborada uma API intermediária ou *wrapper*, na linguagem C, facilitando a interoperabilidade entre esses dois componentes em tempo de ligação (*link-time*), ou seja, o código em RL, compilado com o `robcmp`, poderá ser ligado ao código em C++ para o TFLM, compilado usando um compilador padrão C++.

¹¹ Disponível em: <<https://docs.python.org/3/library/ctypes.html>>

¹² Disponível em: <<https://ffi.readthedocs.io/en/latest/>>

2.9 Robotics Language and Compiler

Desenvolvidos na UFJ em 2018, os trabalhos de Sousa (2018) e Rodrigues (2018) integraram o projeto “Especificação e Construção de Protótipos Funcionais de Kits Robóticos de Baixo Custo para uso em Processos de Ensino-Aprendizagem” (PI02361-2018), que visava criar um ecossistema de robótica educacional de baixo custo. Sousa dedicou-se à especificação do *hardware*, avaliando componentes por preço e desempenho, enquanto Rodrigues criou o *software*, desenvolvendo a Linguagem para Robótica Educacional (LRE) em português e seu respectivo compilador para facilitar a programação. Posteriormente, Subhi (2019) realizou adaptações e melhorias na linguagem, como a mudança da língua para o inglês e a adição de funcionalidades como funções com parâmetros, vetores e matrizes estáticas.

Posteriormente, surgiu o projeto denominado “Robotics Language: Uma Linguagem de Programação de Propósito Específico para Microcontroladores” (PI05974-2024). Trabalhos relacionados a esse projeto implementam funcionalidades ainda inexistentes na biblioteca padrão da linguagem. Por exemplo, Santos (2024) implementou funções matemáticas, como seno e cosseno. Já Vieira (2024) adicionou funções para manipulação de *strings*.

A RL busca isolar as especificidades de microcontroladores, fornecendo uma camada de abstração de *hardware* dentro do próprio compilador, em vez de depender de *frameworks* ou bibliotecas externas. Essa abordagem permite que os desenvolvedores escrevam o código uma única vez, eliminando a necessidade de adaptar o código para cada *hardware* através de macros condicionais. Ao aproveitar a análise semântica do compilador, é possível prevenir erros comuns encontrados no desenvolvimento de *firmware* em linguagens não específicas de domínio, como C/C++ (OLIVEIRA, 2018). Além disso, ela possui suporte a interfaces, tipos complexos (com variáveis e métodos, similar a classes da orientação a objetos), além de um sistema de tipagem implícito.

Seu compilador, o Robcmp, foi escrito em C++ e utilizou ferramentas como o Flex¹³ (versão 2.6.4) para análise léxica e o Bison¹⁴ (versão 3.8.2) para análise sintática. Para a interpretação da representação intermediária, foi utilizado o *backend* oferecido pelo LLVM. Desde o começo do projeto, a versão do LLVM utilizada é constantemente atualizada, para que o compilador se mantenha sempre atualizado.

Para o desenvolvimento de programas e *firmwares* na Robotics Language é recomendado o uso do editor de texto e IDE (*Integrated Development Environment*) Visual Studio Code¹⁵, devido ao seu fácil acesso a extensões. A principal extensão necessária

¹³ Disponível em: <<https://github.com/westes/flex>>

¹⁴ Disponível em: <<https://www.gnu.org/software/bison/>>

¹⁵ Disponível em: <<https://code.visualstudio.com>>

para o desenvolvimento é a PlatformIO¹⁶. Ela é responsável pela execução, depuração e carregamento dos *firmwares*. Outra extensão essencial é a RobCmpSyntax, foi criada com o intuito de ajudar o programador a visualizar a estrutura de seu código a partir de cores diferentes para a sintaxe.

¹⁶ Disponível em: <<https://platformio.org/>>

3 Trabalhos relacionados

3.1 Introdução

Neste capítulo, são apresentadas publicações relacionadas a este trabalho, que abordam temas como, decisões de design do TFLM e ferramentas para *TinyML* e *Edge AI*. Esse material acadêmico foi escolhido a partir de uma revisão narrativa de literatura (RNL). Além disso também será abordado implementações práticas de wrappers do framework TFLM para o desenvolvimento de software para sistemas embarcados.

3.2 Critérios de busca

3.3 Metodologia de análise

3.3.1 Critério 1 (C1)

3.3.2 Critério 2 (C2)

3.3.3 Critério 3 (C3)

3.3.4 Critério 4 (C4)

3.4 Trabalhos analisados

Com base nos critérios apresentados na seção anterior, foram analisados X trabalhos...

3.4.1 Trabalho 1 (T1)

3.4.2 Trabalho 2 (T2)

3.4.3 Trabalho 3 (T3)

3.5 Resumo Comparativo

Observa-se que...

Como verificado na Tabela 1, os trabalhos T1 e T2 utilizam ...

Tabela 1 – Comparativo entre trabalhos

	C1	C2	C3 α	C3 β	C4
T1	Sim	Não	Sim	Não	Sim
T2	Não	Não	Nao	Não	Sim
T3	Sim	Sim	Sim	Não	Sim
TR	Sim	Sim	Sim	Sim	Sim

Dessa forma, os trabalhos elencados neste capítulo ...

4 Metodologia

4.1 Introdução

Este capítulo descreve os procedimentos metodológicos adotados e os recursos empregados para alcançar os objetivos deste trabalho. A apresentação é dividida em seções que abordam, respectivamente, a classificação formal da pesquisa, os materiais de software e hardware utilizados e, por fim, o projeto e a arquitetura da solução que serviram de base para a implementação detalhada no próximo capítulo.

4.2 Classificação da pesquisa

Este projeto propôs ampliar a linguagem de programação RL, adicionando novas funcionalidades de TinyML por meio da integração de uma biblioteca de software. Sendo assim, quanto à sua natureza, esta pesquisa caracterizou-se como aplicada, pois buscou gerar conhecimento para uso prático direto na ampliação da linguagem RL com funcionalidades de TinyML. Quanto aos objetivos, foi exploratória, pois investigou como tais recursos poderiam ser incorporados na linguagem. Em relação aos procedimentos, além de ter sido documental e bibliográfica, por se basear em artigos e materiais já publicados, a pesquisa foi também experimental, uma vez que envolveu a implementação prática e a avaliação das novas funcionalidades na linguagem RL. Por fim, a abordagem foi quantitativa, ao avaliar o desempenho por meio de dados como latência, e qualitativa, com a análise de atributos como simplicidade e manutenibilidade da solução.

4.3 Materiais

Nessa seção, serão abordados todos os materiais que serão utilizados durante a pesquisa. Isso inclui não somente dispositivos físicos, mas também as tecnologias e os *softwares* que serão empregados.

4.3.1 Software

Os novos recursos e funcionalidades implementados diretamente na linguagem RL foram escritos em C++ e desenvolvidos com o auxílio das ferramentas Flex (versão 2.6.4) e Bison (versão 3.8.2), responsáveis por gerar os analisadores léxico e sintático do Robcmp. As adições foram incorporadas à sintaxe da linguagem por meio da criação de novas

palavras reservadas.

O ambiente de desenvolvimento foi o Visual Studio Code (versão 1.101), com o auxílio de duas extensões principais: o PlatformIO (versão 6.1.18), para depurar e exportar os *firmwares* para as placas, e o RobCmpSyntax (versão 1.0), que fornece o realce de sintaxe para arquivos .rob. Para a fase de testes e simulações, foi empregado o emulador QEMU¹ (versão 10.0.2), capaz de emular os microcontroladores da família ARM utilizados no projeto.

4.3.2 Hardware

A máquina de desenvolvimento utilizada foi um *notebook* Lenovo IdeaPad 3 15ITL6, equipado com um processador Intel® Core™ i7-1165G7, 16 GB de RAM e sistema operacional Ubuntu 24.04.1 LTS. Além da máquina de desenvolvimento, a execução e validação das aplicações ocorreram em placas com microcontroladores da família ARM, STM32F401RE e o STM32F405RGT6.

4.4 Projeto e Arquitetura da Solução

Para que a integração fosse desenvolvida corretamente, foi fundamental definir quais funções da biblioteca TFLM seriam utilizáveis no Robcmp. Essa tarefa também incluiu a determinação de quais funcionalidades seriam nativas da linguagem e quais pertenceriam à biblioteca padrão.

A implementação de uma funcionalidade de forma nativa implicou em adicionar novas palavras reservadas à sintaxe da linguagem, ou seja, modificar as análises léxica e sintática. Em contrapartida, as funcionalidades que não modificaram as análises léxica e sintática do Robcmp foram oferecidas por meio da biblioteca padrão. Escritas em arquivos com a extensão .rob, elas puderam ser reaproveitadas pelos desenvolvedores ao importá-las no início de seus códigos. A biblioteca do Robcmp já contemplava alguns módulos, como o math, que podia ser importado para a utilização de funções matemáticas.

A Figura 10 exibe um trecho de código que exemplifica o fluxo de um programa utilizando as novas palavras-chave da linguagem RL. A palavra-chave `model` foi utilizada para carregar o modelo e criar um objeto que conteria todas as informações necessárias para seu correto funcionamento. Já o comando `invoke` ficou responsável por realizar a inferência, ou seja, processar os dados de entrada (*input*) para gerar a saída (*output*).

¹ Disponível em: <<https://www.qemu.org>>


```
1 // Uma implementacao de microcontrolador sera vinculada aqui.
2 mmcu = mcu();
3
4 int16 main() {
5
6     // 1. Define os dados de entrada e um buffer para a saida.
7     input = {0.77, 1.57, 2.3, 3.14};
8     output = {5:0.0};
9
10    // 2. Carrega o modelo
11    model meu_modelo("meu_modelo.tflite", arena_size: 4096);
12
13    // 3. Fornece a entrada para o modelo.
14    meu_modelo.input = input;
15
16    // 4. Executar a inferencia.
17    invoke meu_modelo;
18
19    // 5. Obtem o resultado.
20    output = meu_modelo.output;
21
22    //Resto do codigo
23
24 }
```

Figura 10 – Exemplo de código com a nova sintaxe para inferência de um modelo tflite.

4.5 Procedimentos de Validação Funcional

A etapa de validação funcional teve como objetivo verificar se a solução proposta cumpria o que prometia, sem levar em consideração sua eficiência, aplicabilidade, facilidade ou qualquer outro fator que mede sua qualidade.

Nesta etapa, as novas funcionalidades foram validadas utilizando o simulador QEMU, com suporte para a emulação de plataformas da família ARM. Nesse processo, o QEMU foi empregado para executar e testar os firmwares gerados pelo compilador Robcmp, que utilizaram diferentes modelos de ML. O emulador operou no modo de Emulação de Sistema Completo (Full System Emulation), atuando como uma “placa de desenvolvimento virtual” que consegue executar o firmware e simular os periféricos necessários. A configuração e a operação da máquina virtual foram controladas por linha de comando, por meio de parâmetros que especificavam o modelo da plataforma, a CPU exata, o arquivo de firmware a ser executado, entre outros aspectos.

4.6 Validação da eficiência e Análise Comparativa

Para validar a eficiência da integração do TFLM à RL, foi realizada uma análise comparativa entre duas abordagens de desenvolvimento de firmwares. Foi comparada a

solução proposta em RL com um firmware implementado diretamente em C++, utilizando o próprio TFLM como base para ambos. Para garantir uma avaliação justa, ambos os testes foram conduzidos em um MCU idêntico e com o mesmo modelo de ML. Nesta avaliação, foram considerados critérios como a facilidade de desenvolvimento, a manutenibilidade do código, a latência de inferência e o tamanho final do executável.

5 Implementação

5.1 Introdução

Neste capítulo são apresentados os detalhes da implementação das funcionalidades propostas neste trabalho. O código-fonte completo pode ser encontrado no repositório online <<https://github.com/LuizEduardoRezende/robcmp>>. O referido repositório é um *fork* do projeto Robcmp original, e todo o desenvolvimento foi realizado em uma *branch* dedicada. Futuramente, será submetido um *pull request* para integrar estas contribuições à *branch* principal do projeto Robcmp, a fim de que as novas funcionalidades fiquem disponíveis para a comunidade.

5.2 Arquitetura da Camada de Interoperabilidade

O processo de compilação e linkedição do Robcmp pode ser explicado da seguinte forma: Programas escritos com a extensão .rob são compilados e se tornam arquivos objetos (.o), enquanto que bibliotecas externas como o TFLM ou outra são compiladas e um arquivo de biblioteca estática (.a) é gerado. Depois de compilar todos programas e bibliotecas necessários o linker entra em ação com o objetivo de resolver todas referências entre os componentes para criar o executável único que pode ser chamado de firmware.

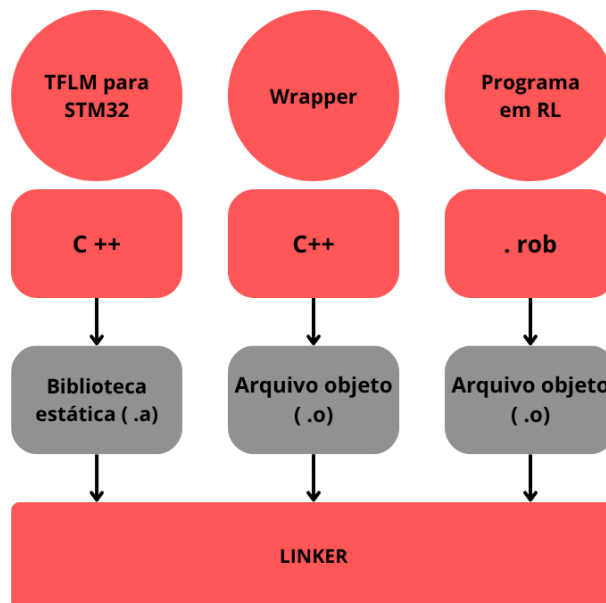


Figura 11 – Ilustração da arquitetura de compilação e interoperabilidade. A biblioteca estática (.a) do TFLM, o arquivo objeto (.o) do wrapper C++, e o arquivo objeto (.o) do programa em Robotics Language são processados de forma independente e, na etapa final, unificados pelo linker. O linker resolve as referências entre os componentes para criar o executável único.

5.3 Implementação do Wrapper C

5.4 Adaptação do Frontend do Compilador

5.5 Considerações Finais

Este trabalho tem como objetivo inicial implementar o suporte para TinyML na família de microcontroladores ARM Cortex-M. Contudo, como o compilador Robcmp está em constante evolução e pode suportar novas arquiteturas no futuro, o mesmo poderá ser feito para o TinyML. Para que isso ocorra, as bibliotecas do TFLM e os wrappers deverão ser recompilados para a plataforma-alvo específica e disponibilizados no repositório do Robcmp.

6 Funcionamento ou Demonstração

6.1 Introdução

Nesta seção é apresentado o funcionamento daquilo que foi planejado no capítulo 4 (metodologia), construído no capítulo 5 (construção). Em muitos casos é necessário usar um estudo de caso para apresentar a contribuição (neste modelo de pesquisa o estudo de caso não é tão importante e sim a contribuição). Outros casos, quando existe uma preocupação com o estudo de caso, ligados diretamente ao objeto de estudo que deu origem ao problema, o mesmo (estudo de caso) pode ser apresentado no capítulo 4 (descrição necessária para definição da solução) e novamente nesta seção (aplicação da solução).

6.2 Exemplo de Aplicação

6.3 Considerações Finais

7 Avaliação e Testes

7.1 Introdução

Neste capítulo é apresentada uma análise da aplicação ou uso do que foi construído e demonstrado no capítulo anterior. Esta análise pode ser realizada usando normas (ISO/EIC) já estabelecidas. Pode ser por meio de testes de laboratório, pode ser simulada, pode ser aplicada em um contexto (universo/população/amostragem) real. E, também pode ser por meio da criação de uma metodologia própria para a análise.

7.2 Ambiente Experimental

7.3 Validação Funcional

7.4 Validação da eficiência e Análise Comparativa

7.5 Considerações Finais

8 Conclusões e Trabalhos Futuros

8.1 Introdução

Este capítulo tem como objetivo apresentar os principais pontos discutidos no trabalho, relacionar os possíveis trabalhos futuros advindos desta pesquisa e avaliar a principal contribuição deste trabalho para a área científica.

8.2 Conclusões

As conclusões devem estabelecer uma descrição sucinta e sintética daquilo que o autor concluiu ao desenvolver sua pesquisa. Deve haver um cuidado para que a mesma não seja óbvia e também que não seja impossível de identificar no texto. Conclusões sobre um ou outro tipo de tecnologia usada, conclusões sobre caminhos que foram tomados na condução da pesquisa são importantes. E, cabe ressaltar que este texto é do autor, portanto não cabe nesta seção a inserção de referências bibliográficas.

8.2.1 *Quanto à área aplicada*

8.2.2 *Quanto à área específica*

8.3 Trabalhos futuros

Em todo trabalho científico, vários caminhos podem ser estabelecidos. Porém cabe geralmente ao autor definir um único para viabilizar a produção e divulgação da sua pesquisa. Estes outros caminhos podem ser apresentados nesta seção, detalhando claramente os motivos da não escolha pelos mesmos. Também muito importante nesta seção, é vislumbrar o que ainda pode ser realizado na sequência do próprio trabalho. Toda pesquisa é provavelmente infinita, o que a classifica como concluída, é apenas um ponto de parada para sua divulgação. Portanto, outras contribuições são e serão sempre passíveis. É exatamente isso que cria a evolução, o desenvolvimento e gera inovação. Na área da Ciências Exatas, o termo “Estado da Arte” dá a exatidão desta sequência.

8.4 Considerações finais

Referências

- ABADADE, Y. et al. A comprehensive survey on tinymml. *IEEE Access*, IEEE, v. 11, p. 96892–96922, 2023. Citado 2 vezes nas páginas 10 e 11.
- AHO, A. V. et al. *Compiladores: princípios, técnicas e ferramentas*. 2. ed. São Paulo: Pearson Addison-Wesley, 2008. 648 p. ISBN 978-85-88639-24-9. Citado 2 vezes nas páginas 17 e 18.
- ATTARAN, N. et al. Embedded low-power processor for personalized stress detection. *IEEE Transactions on Circuits and Systems II: Express Briefs*, v. 65, n. 12, p. 2032–2036, 2018. Citado na página 9.
- AXELSON, J. *The Microcontroller Idea Book: Circuits, Programs & Applications Featuring the 8052-BASIC Microcontroller*. [S.l.]: lakeview research llc, 1997. Citado na página 9.
- Cambridge University Press. *Cambridge Dictionary: Mainstream*. 2025. <<https://dictionary.cambridge.org/dictionary/english/mainstream>>. Acesso em: 19 mai. 2025. Citado na página 22.
- COOPER, K. *Construindo Compiladores*. GEN LTC, 2013. ISBN 9788535255645. Disponível em: <<https://books.google.com.br/books?id=M2cIrgEACAAJ>>. Citado 7 vezes nas páginas 4, 13, 14, 15, 16, 17 e 19.
- DAVID, R. et al. Tensorflow lite micro: Embedded machine learning for tinymml systems. In: SMOLA, A.; DIMAKIS, A.; STOICA, I. (Ed.). *Proceedings of Machine Learning and Systems*. [s.n.], 2021. v. 3, p. 800–811. Disponível em: <https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf>. Citado 2 vezes nas páginas 11 e 24.
- FISCHER, C.; CYTRON, R.; LEBLANC, R. *Crafting a Compiler*. Addison-Wesley, 2010. (Crafting a compiler with C). ISBN 9780136067054. Disponível em: <https://books.google.com.br/books?id=G4Y_AQAIAAJ>. Citado 4 vezes nas páginas 4, 13, 14 e 15.
- GIUSTO, D. et al. (Ed.). *The Internet of Things, 20th Tyrrhenian Workshop on Digital Communications*. Berlin: Springer, 2010. ISBN 978-1-4419-1673-0. Citado na página 20.
- HOANG, T. M.; NAM, S. H.; PARK, K. R. Enhanced detection and recognition of road markings based on adaptive region of interest and deep learning. *IEEE Access*, v. 7, p. 109817–109832, 2019. Citado na página 9.
- HUSSAIN, A. et al. Programming a microcontroller. *Int. J. Comput. Appl*, v. 155, n. 5, p. 21–26, 2016. Citado 2 vezes nas páginas 9 e 20.
- International Organization for Standardization (ISO); International Electrotechnical Commission (IEC); Institute of Electrical and Electronics Engineers (IEEE). Norma Internacional, *ISO/IEC/IEEE 24765: Systems and software engineering — Vocabulary*. 2017. Citado na página 25.

- MARWEDEL, P. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. 4th. ed. Springer, 2021. (Embedded Systems). 1st edition: Springer US 2006; 2nd edition: Springer Netherlands 2011; 3rd edition: Springer International Publishing 2018. ISBN 978-3-030-60909-2. Disponível em: <<https://doi.org/10.1007/978-3-030-60910-8>>. Citado 2 vezes nas páginas 19 e 20.
- Microchip Technology Inc. *AVR® 8-bit MCUs*. 2025. <<https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/avr-mcus>>. Acessado em: 25 de maio de 2025. Citado 2 vezes nas páginas 4 e 21.
- OLIVEIRA, T. Borges de. *RobCMP - Robotic Compiler*. 2018. <<https://github.com/thborges/robcmp/>>. Acesso em: 14 abril de 2025. Disponível em: <<https://github.com/thborges/robcmp/>>. Citado 2 vezes nas páginas 12 e 26.
- 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 26.
- RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. Pearson, 2020. (Pearson series in artificial intelligence). ISBN 9780134610993. Disponível em: <<https://books.google.com.br/books?id=koFptAEACAAJ>>. Citado na página 22.
- SANTOS, R. F. M. dos. *Design and Implementation of the Standard Library for a Domain-Specific Programming Language in Robotics: Mathematical Functions*. 49 p. Monografia — Universidade Federal de Jataí, Jataí, GO, Brasil, 2024. Citado na página 26.
- SINGH, R.; GILL, S. S. Edge ai: A survey. *Internet of Things and Cyber-Physical Systems*, v. 3, p. 71–92, 2023. ISSN 2667-3452. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S2667345223000196>>. Citado 2 vezes nas páginas 9 e 22.
- SORO, S. Tinyml for ubiquitous edge ai. *arXiv preprint arXiv:2102.01255*, 2021. Citado 2 vezes nas páginas 10 e 24.
- SOUSA, W. G. de. *Avaliação Técnica de Componentes Eletrônicos e Microprocessadores para uso no Processo de Ensino-Aprendizagem de Ciências Exatas*. 55 p. Monografia — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2018. Citado na página 26.
- STMicroelectronics. *STM32F103x8 STM32F103xB: Medium-density performance line Arm®-based 32-bit MCU with 64 or 128 KB Flash, USB, CAN, 7 timers, 2 ADCs, 9 com. interfaces*. [S.l.], 2023. Production data. Citado 2 vezes nas páginas 4 e 21.
- STMicroelectronics. *Arm Microcontrollers - 32-bit MCUs - STMicroelectronics*. 2025. <https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers.html>. Acessado em: 25 de maio de 2025. Última atualização informada pela página: 2025. Citado na página 21.
- 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 na página 26.

The LLVM Project. *LLVM Language Reference Manual*. 2025. <<https://llvm.org/docs/LangRef.html>>. Acessado em 18 de maio 2025. Citado 3 vezes nas páginas 4, 18 e 19.

VIEIRA, G. K. de A. *Design and Implementation of the Standard Library for a Domain-Specific Programming Language in Robotics: String Manipulation*. 57 p. Monografia — Universidade Federal de Jataí, Jataí, GO, Brasil, 2024. Citado na página 26.

WARDEN, P.; SITUNAYAKE, D. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019. Disponível em: <<https://www.oreilly.com/library/view/tinyml/9781492052036/>>. Citado 2 vezes nas páginas 10 e 23.

WU, Z.; QIU, K.; ZHANG, J. A smart microcontroller architecture for the internet of things. *Sensors*, v. 20, n. 7, 2020. ISSN 1424-8220. Disponível em: <<https://www.mdpi.com/1424-8220/20/7/1821>>. Citado na página 9.

Anexos

ANEXO A – Exemplo de Formulário

Este formulário tem como objetivo ...:

Pergunta 1:

Pergunta 2:

Pergunta 3: