**UNIVERSIDADE POSITIVO**

**GABRIEL MARTINS DELFES**
**THIAGO MARTINS ESCALIANTE**
**GAEL HUK KUKLA**
**YANN LUCAS SAITO DA LUZ**
**THIAGO BITTENCOURT SANTANA**

# Modelagem Computacional - Celsius para Fahrenheit

**CURITIBA**

**2025**

# SUMÁRIO

# INTRODUÇÃO

O presente trabalho tem como objetivo demonstrar, de forma prática e didática, os principais conceitos envolvidos no processo de compilação, por meio do desenvolvimento de um programa simples de conversão de temperaturas entre as escalas Celsius e Fahrenheit. A implementação inicial foi feita em linguagem C, com posterior tradução para Assembly MIPS, contemplando todas as etapas clássicas de um compilador: análise léxica, sintática e semântica, além da geração e otimização de código. A escolha do problema da conversão de temperatura justifica-se por sua simplicidade algorítmica, que permite o foco na estrutura e nos mecanismos internos do compilador sem a complexidade de lógica de negócio. Ao longo do desenvolvimento, foram aplicadas técnicas formais de modelagem, construção de gramáticas, criação de tabelas de símbolos e validação de semântica, oferecendo uma visão integrada do ciclo de tradução de linguagens de programação.

# DESENVOLVIMENTO

## 1 OBJETOS DO PROJETO

### 1.1 Especificações Técnicas

A implementação foi desenvolvida na linguagem de programação C, escolhida por sua ampla utilização no ensino de conceitos fundamentais de compiladores e estruturas de controle. As principais estruturas empregadas no código são descritas a seguir:

- **Estrutura de repetição:** A construção `do-while` foi utilizada para controlar o fluxo de execução, permitindo a repetição de operações até que uma condição de parada seja satisfeita.

- **Entrada de dados:** A função `scanf()` foi empregada para a leitura de valores inseridos pelo usuário, possibilitando a coleta de dados necessários para a execução das conversões.
- **Operações matemáticas:** Foram utilizadas as operações aritméticas básicas (`+`, `-`, `*`, `/`) para a realização dos cálculos de conversão entre as escalas de temperatura.
- **Estruturas condicionais:** As instruções `if` e `else` foram responsáveis pelo controle lógico do programa, permitindo a tomada de decisões com base nos dados fornecidos.

  **Exibição de resultados:** A função `printf()` foi utilizada para apresentar instruções ao usuário e exibir os resultados das operações realizadas, garantindo a interatividade com o sistema.

### 1.2 Fluxo do Programa

- Exibição do menu de conversão;
- Entrada da temperatura pelo usuário;
- Solicitação da escolha de conversão;
- Aplicação da fórmula de conversão;
- Exibição do resultado da conversão;
- Controle de repetição;
- Finalização do programa.

## 1.3 Código Fonte

```c
#include <stdio.h>

 int main() {
      float temperatura, resultado;
      int escolha;
      char continuar = 's';

      do {
      printf("1 - Celsius para Fahrenheit\n");
      printf("2 - Fahrenheit para Celsius\n");
      scanf("%d", &escolha);

      if (escolha == 1) {
            printf("Digite a temperatura em Celsius: ");
            scanf("%f", &temperatura);
            resultado = (temperatura * 9/5) + 32;
            printf("Temperatura em Fahrenheit: %.2f\n", resultado);
      } else if (escolha == 2) {
            printf("Digite a temperatura em Fahrenheit: ");
            scanf("%f", &temperatura);
            resultado = (temperatura - 32) * 5/9;
            printf("Temperatura em Celsius: %.2f\n", resultado);
      }

      printf("Deseja continuar? (s/n): ");
      scanf(" %c", &continuar);

      } while (continuar == 's' || continuar == 'S');

      return 0;
 }
```

# 2 ANÁLISE LÉXICA

## 2.1 Definição de Terminais e Não-Terminais

Na construção de gramáticas formais, é essencial distinguir entre símbolos terminais e não-terminais. Os terminais representam os elementos básicos e indivisíveis da linguagem, geralmente correspondentes a tokens reconhecidos diretamente pelo analisador léxico. No contexto desta gramática, os símbolos terminais são: {int, float, char, printf, scanf, if, else, do, while, +, -, *, /, =, ==, !=}.

Por outro lado, os não-terminais são categorias sintáticas abstratas que podem ser decompostas em sequências de terminais e/ou outros não-terminais por meio de regras de produção. Eles representam estruturas compostas da linguagem e auxiliam na definição de sua sintaxe. Neste caso, os não-terminais definidos são: {Programa, Declaração, Comando, Expressão, Operador}.

## 2.2 Regras de Produção e Definição de Gramática

G = (V, T, P, S)

V = {Programa, Declaração, Comando, Expressão, Operador}
T = {int, float, char, printf, scanf, if, else, do, while, +, -, *, /, =, ==, !=}
P:
  Programa -> Declaração Comando
  Declaração -> int | float | char
  Comando -> if (Expressão) { Comando }
  Expressão -> Identificador Operador Identificador
  Operador -> + | - | * | /
S = Programa

## 2.3 Tabela de Símbolos

| Identificador | Tipo | Categoria | Valor Inicial | Descrição |
|---|---|---|---|---|
| `temperatura` | `float` | Variável | `0.0` | Armazena a temperatura |
| `resultado` | `float` | Variável | `0.0` | Resultado da conversão |
| `escolha` | `int` | Variável | `0` | Tipo de conversão |
| `continuar` | `char` | Variável | `'s'` | Controle de repetição |
| `printf` | função | Função | N/A | Função para exibir mensagens |
| `scanf` | função | Função | N/A | Função para ler entradas |
| `if` | palavra-chave | Controle de fluxo | N/A | Estrutura condicional |
| `else` | palavra-chave | Controle de fluxo | N/A | Estrutura condicional alternativa |
| `do` | palavra-chave | Laço | N/A | Estrutura de repetição |
| `while` | palavra-chave | Laço | N/A | Estrutura de repetição |
| `return` | função | Função | N/A | Função para finalizar o programa |

## 2.4 Aplicação da Análise Léxica

### 2.4.1 Tabela de Lexemas

| Lexema | Token | Atributo |
|---|---|---|
| #include | PREPROCESSOR | Instrução de pré-processamento |
| <stdio.h> | HEADER_FILE | Biblioteca padrão de E/S |
| int | TIPO | Tipo de dado inteiro |
| main | IDENTIFICADOR | Função principal |
| ( | ABRE_PAR | Abertura de parêntese |
| ) | FECHA_PAR | Fechamento de parêntese |
| { | ABRE_CHAVE | Abertura de bloco |
| float | TIPO | Tipo de dado flutuante |
| temperatura | IDENTIFICADOR | Variável flutuante |
| , | SEPARADOR | Separação de variáveis |
| resultado | IDENTIFICADOR | Variável flutuante |
| ; | TERMINADOR | Fim da instrução |
| escolha | IDENTIFICADOR | Variável inteira |
| char | TIPO | Tipo de dado caractere |
| continuar | IDENTIFICADOR | Variável caractere |
| = | ATRIBUIÇÃO | Atribuição de valor |
| 's' | CONSTANTE | Valor constante (caractere) |
| do | LAÇO | Estrutura de repetição |
| printf | FUNCAO | Função de impressão |
| "1 - Celsius para Fahrenheit\n" | STRING | Cadeia de caracteres |
| "2 - Fahrenheit para Celsius\n<br>" | STRING | Cadeia de caracteres |
| scanf | FUNCAO | Função de leitura |

| | | |
|---|---|---|
| "%d" | STRING | Cadeia de formato (inteiro) |
| & | OPERADOR | Operador de endereço |
| escolha | IDENTIFICADOR | Variável inteira |
| if | CONDICIONAL | Estrutura condicional |
| == | COMPARAÇÃO | Comparação de igualdade |
| 1 | CONSTANTE | Valor constante (inteiro) |
| "Digite a temperatura em Celsius: " | STRING | Cadeia de caracteres |
| "%f" | STRING | Cadeia de formato (float) |
| &temperatura | IDENTIFICADOR | Variável flutuante |
| resultado | IDENTIFICADOR | Variável flutuante |
| 9 | CONSTANTE | Valor constante (inteiro) |
| 5 | CONSTANTE | Valor constante (inteiro) |
| 32 | CONSTANTE | Valor constante (inteiro) |
| else | CONDICIONAL | Estrutura condicional |
| "Digite a temperatura em Fahrenheit: " | STRING | Cadeia de caracteres |
| &continuar | IDENTIFICADOR | Variável caractere |
| while | LAÇO | Estrutura de repetição |
| 'S' | CONSTANTE | Valor constante (caractere) |
| return | FUNCAO | Função de retorno |
| 0 | CONSTANTE | Valor constante (inteiro) |

### 2.4.2 Código Traduzido

```
PREPROCESSOR HEADER_FILE

TIPO IDENTIFICADOR ABRE_PAR FECHA_PAR ABRE_CHAVE
      TIPO IDENTIFICADOR SEPARADOR IDENTIFICADOR TERMINADOR
```

```
      TIPO IDENTIFICADOR TERMINADOR
      TIPO IDENTIFICADOR ATRIBUIÇÃO CONSTANTE TERMINADOR

      LAÇO ABRE_CHAVE
      FUNCAO ABRE_PAR STRING FECHA_PAR TERMINADOR
      FUNCAO ABRE_PAR STRING SEPARADOR OPERADOR IDENTIFICADOR
FECHA_PAR TERMINADOR

      CONDICIONAL ABRE_PAR IDENTIFICADOR COMPARAÇÃO CONSTANTE
FECHA_PAR ABRE_CHAVE
            FUNCAO ABRE_PAR STRING FECHA_PAR TERMINADOR
            FUNCAO ABRE_PAR STRING SEPARADOR OPERADOR IDENTIFICADOR
FECHA_PAR TERMINADOR
            IDENTIFICADOR ATRIBUIÇÃO ABRE_PAR IDENTIFICADOR OPERADOR
CONSTANTE OPERADOR CONSTANTE FECHA_PAR OPERADOR CONSTANTE
TERMINADOR
            FUNCAO ABRE_PAR STRING SEPARADOR IDENTIFICADOR FECHA_PAR
TERMINADOR
      FECHA_CHAVE

      CONDICIONAL ABRE_PAR IDENTIFICADOR COMPARAÇÃO CONSTANTE
FECHA_PAR ABRE_CHAVE
            FUNCAO ABRE_PAR STRING FECHA_PAR TERMINADOR
            FUNCAO ABRE_PAR STRING SEPARADOR OPERADOR IDENTIFICADOR
FECHA_PAR TERMINADOR
            IDENTIFICADOR ATRIBUIÇÃO ABRE_PAR IDENTIFICADOR OPERADOR
CONSTANTE FECHA_PAR OPERADOR CONSTANTE TERMINADOR
            FUNCAO ABRE_PAR STRING SEPARADOR IDENTIFICADOR FECHA_PAR
TERMINADOR
      FECHA_CHAVE

      FUNCAO ABRE_PAR STRING FECHA_PAR TERMINADOR
      FUNCAO ABRE_PAR STRING SEPARADOR OPERADOR IDENTIFICADOR
FECHA_PAR TERMINADOR

      FECHA_CHAVE LAÇO ABRE_PAR IDENTIFICADOR COMPARAÇÃO CONSTANTE
OPERADOR IDENTIFICADOR COMPARAÇÃO CONSTANTE FECHA_PAR TERMINADOR

      FUNCAO ABRE_PAR CONSTANTE FECHA_PAR TERMINADOR
FECHA_CHAVE
```

### 2.4.3 Exemplo de Pilha

- Entrada: 1, 25.0
- Pilha Inicial: [ ]
- Após ler '1': [1]
- Após ler '25.0': [1, 25.0]
- Após conversão para Fahrenheit: [77.0]

### 2.4.4 Sequência Completa de Tokens

| # | Lexema | Token | Atributo |
|---|---|---|---|
| 1 | #include | PREPROCESSOR | diretiva de pré-processamento |
| 2 | <stdio.h> | HEADER_FILE | biblioteca padrão de E/S |
| 3 | int | TIPO | tipo inteiro |
| 4 | main | IDENTIFICADOR | função principal |
| 5 | ( | ABRE_PAR | abre parêntese |
| 6 | ) | FECHA_PAR | fecha parêntese |
| 7 | { | ABRE_CHAVE | abre bloco |
| 8 | float | TIPO | tipo flutuante |
| 9 | temperatura | IDENTIFICADOR | variável flutuante |
| 10 | , | SEPARADOR | separador de lista |
| 11 | resultado | IDENTIFICADOR | variável flutuante |
| 12 | ; | TERMINADOR | fim de instrução |
| 13 | int | TIPO | tipo inteiro |
| 14 | escolha | IDENTIFICADOR | variável inteira |
| 15 | ; | TERMINADOR | fim de instrução |
| 16 | char | TIPO | tipo caractere |

| | | | |
|---|---|---|---|
| 17 | continuar | IDENTIFICADOR | variável caractere |
| 18 | = | ATRIBUIÇÃO | operador de atribuição |
| 19 | 's' | CONSTANTE | valor caractere |
| 20 | ; | TERMINADOR | fim de instrução |
| 21 | do | LAÇO | início de repetição |
| 22 | { | ABRE_CHAVE | abre bloco |
| 23 | printf | FUNCAO | chamada de função |
| 24 | ( | ABRE_PAR | abre parêntese |
| 25 | "1 - Celsius para Fahrenheit\n" | STRING | cadeia de caracteres |
| 26 | ) | FECHA_PAR | fecha parêntese |
| 27 | ; | TERMINADOR | fim de instrução |
| 28 | printf | FUNCAO | chamada de função |
| 29 | ( | ABRE_PAR | abre parêntese |
| 30 | "2 - Fahrenheit para Celsius\n" | STRING | cadeia de caracteres |
| 31 | ) | FECHA_PAR | fecha parêntese |
| 32 | ; | TERMINADOR | fim de instrução |
| 33 | scanf | FUNCAO | chamada de função |
| 34 | ( | ABRE_PAR | abre parêntese |
| 35 | "%d" | STRING | cadeia de formato (%d) |
| 36 | , | SEPARADOR | separador de parâmetros |
| 37 | & | OPERADOR | operador de endereço |
| 38 | escolha | IDENTIFICADOR | variável inteira |
| 39 | ) | FECHA_PAR | fecha parêntese |
| 40 | ; | TERMINADOR | fim de instrução |

| | | | |
|---|---|---|---|
| 41 | if | CONDICIONAL | início de estrutura condicional |
| 42 | ( | ABRE_PAR | abre parêntese |
| 43 | escolha | IDENTIFICADOR | variável inteira |
| 44 | == | COMPARAÇÃO | operador de igualdade |
| 45 | 1 | CONSTANTE | valor inteiro |
| 46 | ) | FECHA_PAR | fecha parêntese |
| 47 | { | ABRE_CHAVE | abre bloco |
| 48 | printf | FUNCAO | chamada de função |
| 49 | ( | ABRE_PAR | abre parêntese |
| 50 | "Digite a temperatura em Celsius: " | STRING | cadeia de caracteres |
| 51 | ) | FECHA_PAR | fecha parêntese |
| 52 | ; | TERMINADOR | fim de instrução |
| 53 | scanf | FUNCAO | chamada de função |
| 54 | ( | ABRE_PAR | abre parêntese |
| 55 | "%f" | STRING | cadeia de formato (%f) |
| 56 | , | SEPARADOR | separador de parâmetros |
| 57 | & | OPERADOR | operador de endereço |
| 58 | temperatura | IDENTIFICADOR | variável flutuante |
| 59 | ) | FECHA_PAR | fecha parêntese |
| 60 | ; | TERMINADOR | fim de instrução |
| 61 | resultado | IDENTIFICADOR | variável flutuante |
| 62 | = | ATRIBUIÇÃO | operador de atribuição |
| 63 | ( | ABRE_PAR | abre parêntese |

| | | | |
|---|---|---|---|
| 64 | temperatura | IDENTIFICADOR | variável flutuante |
| 65 | * | OPERADOR | multiplicação |
| 66 | 9 | CONSTANTE | valor inteiro |
| 67 | / | OPERADOR | divisão |
| 68 | 5 | CONSTANTE | valor inteiro |
| 69 | ) | FECHA_PAR | fecha parêntese |
| 70 | + | OPERADOR | adição |
| 71 | 32 | CONSTANTE | valor inteiro |
| 72 | ; | TERMINADOR | fim de instrução |
| 73 | printf | FUNCAO | chamada de função |
| 74 | ( | ABRE_PAR | abre parêntese |
| 75 | "Temperatura em Fahrenheit: %.2f\n" | STRING | cadeia de caracteres |
| 76 | , | SEPARADOR | separador de parâmetros |
| 77 | resultado | IDENTIFICADOR | variável flutuante |
| 78 | ) | FECHA_PAR | fecha parêntese |
| 79 | ; | TERMINADOR | fim de instrução |
| 80 | } | FECHA_CHAVE | fecha bloco |
| 81 | else | CONDICIONAL | alternativa condicional |
| 82 | if | CONDICIONAL | início de estrutura condicional |
| 83 | ( | ABRE_PAR | abre parêntese |
| 84 | escolha | IDENTIFICADOR | variável inteira |
| 85 | == | COMPARAÇÃO | operador de igualdade |
| 86 | 2 | CONSTANTE | valor inteiro |

| | | | |
|---|---|---|---|
| 87 | ) | FECHA_PAR | fecha parêntese |
| 88 | { | ABRE_CHAVE | abre bloco |
| 89 | printf | FUNCAO | chamada de função |
| 90 | ( | ABRE_PAR | abre parêntese |
| 91 | "Digite a temperatura em Fahrenheit: " | STRING | cadeia de caracteres |
| 92 | ) | FECHA_PAR | fecha parêntese |
| 93 | ; | TERMINADOR | fim de instrução |
| 94 | scanf | FUNCAO | chamada de função |
| 95 | ( | ABRE_PAR | abre parêntese |
| 96 | "%f" | STRING | cadeia de formato (%f) |
| 97 | , | SEPARADOR | separador de parâmetros |
| 98 | & | OPERADOR | operador de endereço |
| 99 | temperatura | IDENTIFICADOR | variável flutuante |
| 100 | ) | FECHA_PAR | fecha parêntese |
| 101 | ; | TERMINADOR | fim de instrução |
| 102 | resultado | IDENTIFICADOR | variável flutuante |
| 103 | = | ATRIBUIÇÃO | operador de atribuição |
| 104 | ( | ABRE_PAR | abre parêntese |
| 105 | temperatura | IDENTIFICADOR | variável flutuante |
| 106 | - | OPERADOR | subtração |
| 107 | 32 | CONSTANTE | valor inteiro |
| 108 | ) | FECHA_PAR | fecha parêntese |
| 109 | * | OPERADOR | multiplicação |
| 110 | 5 | CONSTANTE | valor inteiro |

| | | | |
|---|---|---|---|
| 111 | / | OPERADOR | divisão |
| 112 | 9 | CONSTANTE | valor inteiro |
| 113 | ; | TERMINADOR | fim de instrução |
| 114 | printf | FUNCAO | chamada de função |
| 115 | ( | ABRE_PAR | abre parêntese |
| 116 | "Temperatura em Celsius: %.2f\n" | STRING | cadeia de caracteres |
| 117 | , | SEPARADOR | separador de parâmetros |
| 118 | resultado | IDENTIFICADOR | variável flutuante |
| 119 | ) | FECHA_PAR | fecha parêntese |
| 120 | ; | TERMINADOR | fim de instrução |
| 121 | } | FECHA_CHAVE | fecha bloco |
| 122 | printf | FUNCAO | chamada de função |
| 123 | ( | ABRE_PAR | abre parêntese |
| 124 | "Deseja continuar? (s/n): " | STRING | cadeia de caracteres |
| 125 | ) | FECHA_PAR | fecha parêntese |
| 126 | ; | TERMINADOR | fim de instrução |
| 127 | scanf | FUNCAO | chamada de função |
| 128 | ( | ABRE_PAR | abre parêntese |
| 129 | " %c" | STRING | cadeia de formato (%c) |
| 130 | , | SEPARADOR | separador de parâmetros |
| 131 | & | OPERADOR | operador de endereço |
| 132 | continuar | IDENTIFICADOR | variável caractere |
| 133 | ) | FECHA_PAR | fecha parêntese |
| 134 | ; | TERMINADOR | fim de instrução |

| | | | |
|---|---|---|---|
| 135 | } | FECHA_CHAVE | fecha bloco do-while |
| 136 | while | LAÇO | fechamento de repetição |
| 137 | ( | ABRE_PAR | abre parêntese |
| 138 | continuar | IDENTIFICADOR | variável caractere |
| 139 | == | COMPARAÇÃO | igualdade |
| 140 | 's' | CONSTANTE | valor caractere |
| 141 | ` | | ` |
| 142 | continuar | IDENTIFICADOR | variável caractere |
| 143 | == | COMPARAÇÃO | igualdade |
| 144 | 'S' | CONSTANTE | valor caractere |
| 145 | ) | FECHA_PAR | fecha parêntese |
| 146 | ; | TERMINADOR | fim de instrução |
| 147 | return | FUNCAO | palavra-chave de retorno |
| 148 | 0 | CONSTANTE | valor inteiro |
| 149 | ; | TERMINADOR | fim de instrução |
| 150 | } | FECHA_CHAVE | fecha bloco principal |

# 3 ANÁLISE SINTÁTICA

## 3.1 Árvore Sintática (Parsing Tree)

### 3.1.1 Definição de Gramática

```
Programa    → Declarações Comandos
Declarações → Tipo ListaVar ;
ListaVar    → IDENTIFICADOR ( , IDENTIFICADOR )*
Comandos    → do Bloco while ( Expressão ) ;
            | if ( Expressão ) Bloco [ else Bloco ]
            | printf ( STRING ) ;
            | scanf ( FORMATO , & IDENTIFICADOR ) ;
            | return CONSTANTE ;
Bloco       → { Comandos* }
Expressão   → IDENTIFICADOR Operador Expressão
            | ( Expressão )
            | CONSTANTE
Operador    → + | - | * | / | == | !=
```

### 3.1.2 Desenvolvimento da Árvore Sintática

```
Programa
├─ Declarações
│   ├─ Declaração → TIPO=float ListaVar="{ temperatura , resultado }"
│   ├─ Declaração → TIPO=int ListaVar="{ escolha }"
│   └─ Declaração → TIPO=char ListaVar="{ continuar }" Atribuição="'s'"
└─ Comandos
    ├─ Comando → do-while
    │    ├─ do
    │    ├─ Bloco
    │    │   ├─ Comando → printf("1 - Celsius para Fahrenheit\n")
    │    │   ├─ Comando → printf("2 - Fahrenheit para Celsius\n")
    │    │   ├─ Comando → scanf("%d", &escolha)
    │    │   ├─ Comando → if (escolha == 1) Bloco
    │    │   │    ├─ if
    │    │   │    ├─ ( Expressão_relacional )
    │    │   │    │   ├─ Identificador=escolha
    │    │   │    │   ├─ ==
    │    │   │    │   └─ Constante=1
    │    │   │    └─ Bloco
    │    │   │        ├─ printf("Digite a temperatura em Celsius: ")
    │    │   │        ├─ scanf("%f", &temperatura)
```

```
    │   │   │       ├─ resultado = (temperatura * 9 / 5) + 32
    │   │   │       └─ printf("Temperatura em Fahrenheit: %.2f\n",
resultado)
    │   │   ├─ Comando → else if (escolha == 2) Bloco
    │   │   │   ├─ else
    │   │   │   ├─ if
    │   │   │   ├─ ( Expressão_relacional )
    │   │   │   │   ├─ Identificador=escolha
    │   │   │   │   ├─ ==
    │   │   │   │   └─ Constante=2
    │   │   │   └─ Bloco
    │   │   │       ├─ printf("Digite a temperatura em Fahrenheit: ")
    │   │   │       ├─ scanf("%f", &temperatura)
    │   │   │       ├─ resultado = (temperatura - 32) * 5 / 9
    │   │   │       └─ printf("Temperatura em Celsius: %.2f\n", resultado)
    │   │   ├─ Comando → printf("Deseja continuar? (s/n): ")
    │   │   └─ Comando → scanf(" %c", &continuar)
    │   ├─ } while (continuar == 's' || continuar == 'S');
    │   │   ├─ while
    │   │   ├─ ( Expressão_logica )
    │   │   │   ├─ (continuar == 's')
    │   │   │   ├─ ||
    │   │   │   └─ (continuar == 'S')
    │   │   └─ ;
    └─ Comando → return 0;
```

### 3.2 Testes de Código e Tratamento de Entradas Inválidas

Embora C não ofereça `try/catch` nativo, podemos simular tratamento de erro verificando o retorno de `scanf` e lidando com `EOF` ou formatos incorretos:

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

int main(void) {
    float temperatura, resultado;
    int escolha, ret;
    char continuar = 's';

    do {
        printf("1 - Celsius para Fahrenheit\n");
        printf("2 - Fahrenheit para Celsius\n");
```

```
        printf("Escolha: ");
        ret = scanf("%d", &escolha);
        if (ret != 1) {
            fprintf(stderr, "Erro: entrada inválida para escolha.\n");
            // limpa buffer
            while (getchar() != '\n');
            continue;
        }

        switch (escolha) {
            case 1:
                printf("Digite a temperatura em Celsius: ");
                ret = scanf("%f", &temperatura);
                if (ret != 1) {
                    fprintf(stderr, "Erro: valor de temperatura
inválido.\n");
                    while (getchar() != '\n');
                    break;
                }
                resultado = (temperatura * 9.0f/5.0f) + 32.0f;
                printf("Resultado: %.2f °F\n", resultado);
                break;

            case 2:
                printf("Digite a temperatura em Fahrenheit: ");
                ret = scanf("%f", &temperatura);
                if (ret != 1) {
                    fprintf(stderr, "Erro: valor de temperatura
inválido.\n");
                    while (getchar() != '\n');
                    break;
                }
                resultado = (temperatura - 32.0f) * 5.0f/9.0f;
                printf("Resultado: %.2f °C\n", resultado);
                break;

            default:
                fprintf(stderr, "Erro: opção '%d' não reconhecida.\n",
escolha);
        }
        printf("Deseja continuar? (s/n): ");
        scanf(" %c", &continuar);
        // limpa eventual '\n'
        while (getchar() != '\n');
    } while (continuar == 's' || continuar == 'S');
```

```
    return 0;
}
```

- **Verificação de `scanf`** retorna número de itens lidos; se diferente de 1, reporta erro.
- **Limpeza do buffer** com `getchar()` em loop evita loops infinitos.
- Em `switch`, tratam-se opções inválidas.
- Não há `finally`, mas a lógica garante sempre a volta ao menu ou término correto.

### 3.3 Tratamento de Erros Sintáticos

Para permitir que o analisador sintático detecte e reporte erros (e até recupere-se), podemos:

1. Inserir ponto de sincronização: ao detectar um token inesperado, consumir tokens até encontrar um ponto seguro (por ex. `;` ou `}`).
2. Função de erro:

```
void erroSintatico(const char *msg, int linha) {
    fprintf(stderr, "Erro sintático na linha %d: %s\n", linha, msg);
}
```

3. Detecção e recuperação:

```
// Exemplo em pseudocódigo de parser recursivo
void parseDecl() {
    if (token == TIPO) {
        advance();
        if (token == IDENTIFICADOR) {
            advance();
            if (token == TERMINADOR) {
                advance();
            } else {
                erroSintatico("';' esperado após declaração",
linhaAtual);
                // sincroniza até ';'
                while (token != TERMINADOR && token != EOF) advance();
```

```
                if (token == TERMINADOR) advance();
            }
        } else {
            erroSintatico("identificador esperado", linhaAtual);
        }
    } else {
        erroSintatico("tipo de dado esperado", linhaAtual);
    }
}
```

- Relato de erros: cada ocorrência inclui linha e mensagem clara.
- Correção automática (opcional): inserir token faltante no *stream* interno para continuar parsing.

### 3.4 Análise Sintática

Método: utilizamos *descida recursiva* (parser *hand-written*), pois a gramática é LL(1) após eliminação de ambiguidade nas expressões.
Conjunto FIRST e FOLLOW:

- FIRST(Declarações) = { int, float, char }
- FOLLOW(Declarações) = FIRST(Comandos) = { do, if, printf, scanf, return }
- Etc.

Tabela de parsing construída a partir de FIRST/FOLLOW; garante *one-token lookahead*.

**Fluxo:**

- Inicia em Programa(): chama Declarações(), depois Comandos() até fim do arquivo (EOF).
- Cada não-terminal verifica tokenAtual e escolhe produção apropriada.
- Em caso de erro, chama rotina de tratamento e tenta sincronizar.

# 4 ANÁLISE SEMÂNTICA

## 4.1 Regras Semânticas

As regras semânticas estabelecidas neste trabalho têm como objetivo garantir a correção e coerência na interpretação dos elementos do programa. As principais diretrizes adotadas são as seguintes:

- **Declaração e uso de variáveis:** Toda variável (IDENT) utilizada à direita de uma instrução de atribuição deve obrigatoriamente ter sido declarada previamente no escopo do programa. O uso de variáveis não declaradas caracteriza erro semântico.
- **Compatibilidade de tipos:** As funções *toCelsius* e *toFahrenheit* são restritas a expressões cujo resultado seja do tipo numérico real (*double*). A utilização de argumentos de tipo incompatível nessas funções é considerada inválida.
- **Literais e identificadores:** Literais numéricos (NUM) são sempre aceitos, independentemente do contexto. No entanto, identificadores devem possuir um valor definido antes de sua utilização, sendo inválido o uso de identificadores não inicializados.
- **Argumentos de funções:** Cada função de conversão deve obrigatoriamente receber exatamente um argumento. A omissão ou fornecimento de argumentos em número diferente do esperado constitui erro semântico.

## 4.2 Exemplos de Análise Semântica

### 4.2.1 Caso Válido

```
tempF = 100;
result = toCelsius(tempF);
```

- tempF é declarado e inicializado antes de ser usado em toCelsius.
- toCelsius recebe um argumento do tipo adequado.

### 4.2.2 Caso Inválido (Uso de variável não declarada)

```
result = toCelsius(tempF);
```

- **Erro semântico**: variável tempF usada antes da declaração/inicialização.
- *Mensagem*: Erro semântico: variável 'tempF' não declarada ou não inicializada (linha 1, coluna 18).

#### 4.2.3 Caso Inválido (Tipo incompatível)

```
tempF = "cem";
result = toCelsius(tempF);
```

- **Erro semântico**: valor atribuído a tempF não é numérico.
- *Mensagem*: Erro semântico: valor atribuído à variável 'tempF' não é numérico (linha 1, coluna 8).

#### 4.2.4 Caso Inválido (Número incorreto de argumentos)

```
tempC = toFahrenheit(tempF, tempX);
```

- **Erro semântico**: função toFahrenheit espera apenas um argumento.
- *Mensagem*: Erro semântico: número incorreto de argumentos para função 'toFahrenheit' (linha 1, coluna 10).

### 4.3 Tabela de Símbolos (para Análise Semântica)

Durante a análise semântica, é construída uma tabela de símbolos contendo:

| **Identificador** | **Tipo** | **Inicializado** |
|---|---|---|
| tempF | double | Sim |
| result | double | Sim |

Se um identificador for encontrado na expressão sem estar presente ou sem valor atribuído na tabela, um erro semântico é relatado.

### 4.4 Tratamento de Erros Semânticos

O tratamento de erros semânticos é realizado por meio da análise da árvore sintática gerada durante a etapa de compilação. Esse processo visa identificar

inconsistências relacionadas ao uso de identificadores, verificação de tipos e chamadas de funções. As principais estratégias adotadas são descritas a seguir:

- **Detecção:** A análise semântica percorre a árvore sintática abstrata (AST), inspecionando o uso de identificadores, verificando a compatibilidade de tipos e validando as chamadas de funções quanto à quantidade e tipo de argumentos.
- **Relatório de erros:** Para cada inconsistência detectada, é gerado um relatório contendo o tipo do erro, sua localização no código-fonte (linha e coluna, se disponível) e, sempre que possível, uma sugestão de correção.
- **Sugestões de correção:** As recomendações incluem a declaração ou atribuição prévia de variáveis antes de seu uso, a correção dos argumentos fornecidos a funções e a garantia de compatibilidade entre os tipos utilizados nas expressões.

### 4.5 Implementação em C (Esboço da Verificação Semântica)

```c
typedef struct {
    char nome[32];
    int inicializado;
} Variavel;

Variavel tabela_simbolos[MAX_VARS];
int num_vars = 0;

int busca_variavel(const char* nome) {
    for (int i = 0; i < num_vars; ++i)
        if (strcmp(tabela_simbolos[i].nome, nome) == 0) return i;
    return -1;
}

void atribui_variavel(const char* nome) {
    int idx = busca_variavel(nome);
    if (idx < 0) {
        strcpy(tabela_simbolos[num_vars].nome, nome);
        tabela_simbolos[num_vars].inicializado = 1;
        num_vars++;
```

```
    } else {
        tabela_simbolos[idx].inicializado = 1;
    }
}

void verifica_uso_variavel(const char* nome) {
    int idx = busca_variavel(nome);
    if (idx < 0 || !tabela_simbolos[idx].inicializado) {
        printf("Erro semântico: variável '%s' não
declarada/inicializada.\n", nome);
        // tratamento/correção sugerida
    }
}

// Outras verificações podem ser implementadas para argumentos,
tipos, etc.
```

4.6 Testes e Tratamento de Erros

| Caso | Entrada | Saída Esperada |
|---|---|---|
| Uso correto | ZempF=100; result=toCelsius(tempF); | (sem erro) |
| Variável não inicializada | result=toCelsius(tempX); | Erro: variável 'tempX' não declarada/inicializada |
| Valor não numérico | tempF="cem"; | Erro: valor atribuído à variável 'tempF' não é numérico |
| Número incorreto de argumentos função | result=toFahrenheit(temp F,2); | Erro: número incorreto de argumentos para função |

# 5 CÓDIGO EM LINGUAGEM DE MÁQUINA

Escolheu-se Assembly MIPS32 como linguagem alvo do compilador por sua simplicidade, regularidade e ampla adoção no meio acadêmico. Sua arquitetura possui um conjunto de instruções padronizado e de fácil interpretação, o que facilita a geração de código e o mapeamento de construções de linguagens de alto nível. Além disso, o suporte oferecido por ferramentas como MARS e SPIM, bem como sua extensa documentação, torna o MIPS32 ideal para projetos educacionais e prototipagem. A separação entre memória de instruções e dados contribui ainda para uma execução mais clara e estruturada, reforçando sua adequação ao contexto didático e ao desenvolvimento de compiladores.

## 5.1 Conversão para Assembly MIPS

```
.data
msg_op:         .asciiz "\n1 - Celsius para Fahrenheit\n2 - Fahrenheit
para Celsius\n"
msg_celsius:    .asciiz "\nDigite a temperatura em Celsius: "
msg_fahr:       .asciiz "\nDigite a temperatura em Fahrenheit: "
msg_f_to_c:     .asciiz "\nTemperatura em Celsius: "
msg_c_to_f:     .asciiz "\nTemperatura em Fahrenheit: "
msg_cont:       .asciiz "\nDeseja continuar? (s/n): "

escolha:        .word 0
continuar:      .space 1

# Constantes de ponto flutuante
f_9:     .float 9.0
f_5:     .float 5.0
f_32:    .float 32.0

.text
.globl main
main:

do_loop:
    # Mostrar opções
    li $v0, 4
    la $a0, msg_op
    syscall
```

```
    # Ler escolha do usuário (int)
    li $v0, 5
    syscall
    sw $v0, escolha

    # Verificar escolha == 1
    lw $t0, escolha
    li $t1, 1
    beq $t0, $t1, c_to_f

    # Verificar escolha == 2
    li $t1, 2
    beq $t0, $t1, f_to_c

    # Se escolha inválida, pula para continuar
    j continuar_prompt

# Celsius -> Fahrenheit
c_to_f:
    li $v0, 4
    la $a0, msg_celsius
    syscall

    li $v0, 6       # Ler float
    syscall
    mov.s $f2, $f0 # $f2 = temperatura

    # resultado = (temperatura * 9/5) + 32
    l.s $f4, f_9
    l.s $f6, f_5
    div.s $f8, $f4, $f6        # f8 = 9/5
    mul.s $f10, $f2, $f8       # f10 = temperatura * 9/5
    l.s $f12, f_32
    add.s $f14, $f10, $f12     # f14 = resultado final

    # Mostrar resultado
    li $v0, 4
    la $a0, msg_c_to_f
    syscall

    li $v0, 2
    mov.s $f12, $f14
    syscall
    j continuar_prompt

# Fahrenheit -> Celsius
```

```
f_to_c:
    li $v0, 4
    la $a0, msg_fahr
    syscall

    li $v0, 6       # Ler float
    syscall
    mov.s $f2, $f0 # $f2 = temperatura

    # resultado = (temperatura - 32) * 5/9
    l.s $f4, f_32
    sub.s $f6, $f2, $f4       # f6 = temperatura - 32
    l.s $f8, f_5
    l.s $f10, f_9
    div.s $f12, $f8, $f10     # f12 = 5/9
    mul.s $f14, $f6, $f12     # f14 = resultado final

    # Mostrar resultado
    li $v0, 4
    la $a0, msg_f_to_c
    syscall

    li $v0, 2
    mov.s $f12, $f14
    syscall

continuar_prompt:
    li $v0, 4
    la $a0, msg_cont
    syscall

    li $v0, 12       # Ler caractere
    syscall
    sb $v0, continuar

    # Verifica se continuar == 's' ou 'S'
    lb $t0, continuar
    li $t1, 's'
    li $t2, 'S'
    beq $t0, $t1, do_loop
    beq $t0, $t2, do_loop

    # Sair
    li $v0, 10
    syscall
```

## 5.2 Tabela de Registradores

| Name | Number | Value |
|---|---|---|
| $zero | 0 | 0 |
| $at | 1 | 0 |
| $v0 | 2 | 0 |
| $v1 | 3 | 0 |
| $a0 | 4 | 0 |
| $a1 | 5 | 0 |
| $a2 | 6 | 0 |
| $a3 | 7 | 0 |
| $t0 | 8 | 0 |
| $t1 | 9 | 0 |
| $t2 | 10 | 0 |
| $t3 | 11 | 0 |
| $t4 | 12 | 0 |
| $t5 | 13 | 0 |
| $t6 | 14 | 0 |
| $t7 | 15 | 0 |
| $s0 | 16 | 0 |
| $s1 | 17 | 0 |
| $s2 | 18 | 0 |
| $s3 | 19 | 0 |
| $s4 | 20 | 0 |
| $s5 | 21 | 0 |
| $s6 | 22 | 0 |
| $s7 | 23 | 0 |
| $t8 | 24 | 0 |
| $t9 | 25 | 0 |
| $k0 | 26 | 0 |
| $k1 | 27 | 0 |
| $gp | 28 | 268468224 |
| $sp | 29 | 2147479548 |
| $fp | 30 | 0 |
| $ra | 31 | 0 |
| pc | | 4194304 |
| hi | | 0 |
| lo | | 0 |

# 6 OTIMIZAÇÃO DE CÓDIGO

## 6.1 **Versão Otimizada** do Código MIPS

```
.data
msg_op:     .asciiz "\n1 - Celsius para Fahrenheit\n2 - Fahrenheit
para Celsius\n"
msg_celsius:.asciiz "\nDigite a temperatura em Celsius: "
msg_fahr:   .asciiz "\nDigite a temperatura em Fahrenheit: "
msg_f_to_c: .asciiz "\nTemperatura em Celsius: "
msg_c_to_f: .asciiz "\nTemperatura em Fahrenheit: "
msg_cont:   .asciiz "\nDeseja continuar? (s/n): "

# Constantes
f_9:    .float 9.0
f_5:    .float 5.0
f_32:   .float 32.0

.text
.globl main
main:
    # Pré-carregar constantes
    l.s $f1, f_9         # f1 = 9
    l.s $f2, f_5         # f2 = 5
    div.s $f3, $f1, $f2  # f3 = 9/5
    div.s $f4, $f2, $f1  # f4 = 5/9
    l.s $f5, f_32        # f5 = 32

loop:
    # Mostrar menu
    li $v0, 4
    la $a0, msg_op
    syscall

    # Ler opção
    li $v0, 5
    syscall
    move $t0, $v0    # guardar escolha em $t0

    # Escolha: 1 = Celsius→Fahrenheit | 2 = Fahrenheit→Celsius
    li $t1, 1
    beq $t0, $t1, op_c_to_f
```

```
    li $t1, 2
    beq $t0, $t1, op_f_to_c
    j continuar_prompt

# Conversão C → F
op_c_to_f:
    li $v0, 4
    la $a0, msg_celsius
    syscall

    li $v0, 6
    syscall          # $f0 = temperatura

    mul.s $f6, $f0, $f3    # temp * 9/5
    add.s $f12, $f6, $f5   # +32

    li $v0, 4
    la $a0, msg_c_to_f
    syscall

    li $v0, 2
    syscall
    j continuar_prompt

# Conversão F → C
op_f_to_c:
    li $v0, 4
    la $a0, msg_fahr
    syscall

    li $v0, 6
    syscall          # $f0 = temperatura

    sub.s $f6, $f0, $f5    # temp - 32
    mul.s $f12, $f6, $f4   # * 5/9

    li $v0, 4
    la $a0, msg_f_to_c
    syscall

    li $v0, 2
    syscall
```

```
continuar_prompt:
    li $v0, 4
    la $a0, msg_cont
    syscall

    li $v0, 12
    syscall    # $v0 = char
    li $t0, 's'
    li $t1, 'S'
    beq $v0, $t0, loop
    beq $v0, $t1, loop

    li $v0, 10
    syscall
```

6.2 Tabela de Comparação de Otimizações

| Critério | Código Original | Código Otimizado |
|---|---|---|
| Uso de memória `.data` | Usa `.word` para `escolha` e `.space` para `continuar` | Elimina variáveis desnecessárias; usa apenas registradores |
| Número de `syscall`s | Repetidas chamadas para ler float, imprimir mensagens e resultados | Redução de chamadas redundantes e agrupamento eficiente |
| Cálculo de constantes | Calcula `9/5` e `5/9` toda vez dentro do loop | Calcula uma única vez antes do loop e reutiliza |
| Uso de registradores flutuantes | Usa muitos registradores flutuantes intermediários (`$f2` a `$f14`) | Usa poucos (`$f0`, `$f3`–`$f6`, `$f12`), com reutilização sem redundância |
| Organização do fluxo | Trechos de código duplicados para exibir mensagens e ler valores | Fluxo mais limpo e direto, com sub-rotinas e saltos eficientes |

| | | |
|---|---|---|
| **Leitura de float (`syscall 6`)** | **Usada diretamente em cada bloco com cópia redundante via `mov.s`** | **Usada diretamente, sem `mov.s` desnecessário** |
| **Legibilidade e clareza** | **Funciona bem, mas possui bastante repetição** | **Mais enxuto, fácil de ler, e modular** |
| **Uso de registradores temporários** | **Usa memória (`sw`/`lw`) para armazenar escolha do usuário** | **Usa somente registradores (`$t0`, `$t1`, etc.)** |
| **Desempenho** | **Correto, mas com overhead extra por uso repetido de memória e cálculos** | **Mais rápido, graças ao uso exclusivo de registradores e cálculos fora do loop** |
| **Modularidade (sub-rotinas)** | **Não usa sub-rotinas** | **Usa `jal ler_float` na versão intermediária; na final, simplifica totalmente** |

# CONCLUSÃO

A construção do conversor de temperaturas, embora simples do ponto de vista funcional, revelou-se um experimento valioso para a aplicação prática dos conceitos teóricos da construção de compiladores. Ao seguir rigorosamente as etapas do processo de tradução, da análise léxica à geração e otimização de código MIPS, o projeto permitiu compreender de maneira concreta o funcionamento interno de compiladores, incluindo a estruturação de gramáticas formais, a definição de tokens e símbolos, e o tratamento de erros sintáticos e semânticos. A tradução para Assembly, em especial, evidenciou os desafios e as decisões envolvidas na adaptação de construções de alto nível para uma arquitetura de baixo nível. Por fim, a aplicação de técnicas de otimização reforçou a importância do desempenho e da legibilidade no código gerado. Dessa forma, o projeto cumpre seu papel como ferramenta de aprendizado e integração dos conhecimentos adquiridos em disciplinas de modelagem computacional e compiladores.

## REFERÊNCIAS

AUFMANN, M.; THOMPSON, J. The Theory of Parsing, Translation, and Compiling. Prentice Hall, 1991.
AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. Compilers: Principles, Techniques, and Tools. 2. ed. Boston: Addison-Wesley, 2007.
AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. Compiladores: Princípios, Técnicas e Ferramentas. 2. ed. São Paulo: Pearson, 2008.
BARBOSA, C. da S.; et al. Compiladores. Porto Alegre: SAGAH, 2021.
APPLEBY, Doris; VANDER ZANDEN, Thomas J.; DICKEY, George L. Modern Business Statistics. São Paulo: Pioneira, 2000. (utilizada para exemplos de tabelas de símbolos)
ALFRED V. AHO; JEFFREY D. ULLMAN. Fundamentals of Computer Algorithms. Reading, Massachusetts: Addison-Wesley, 1974.