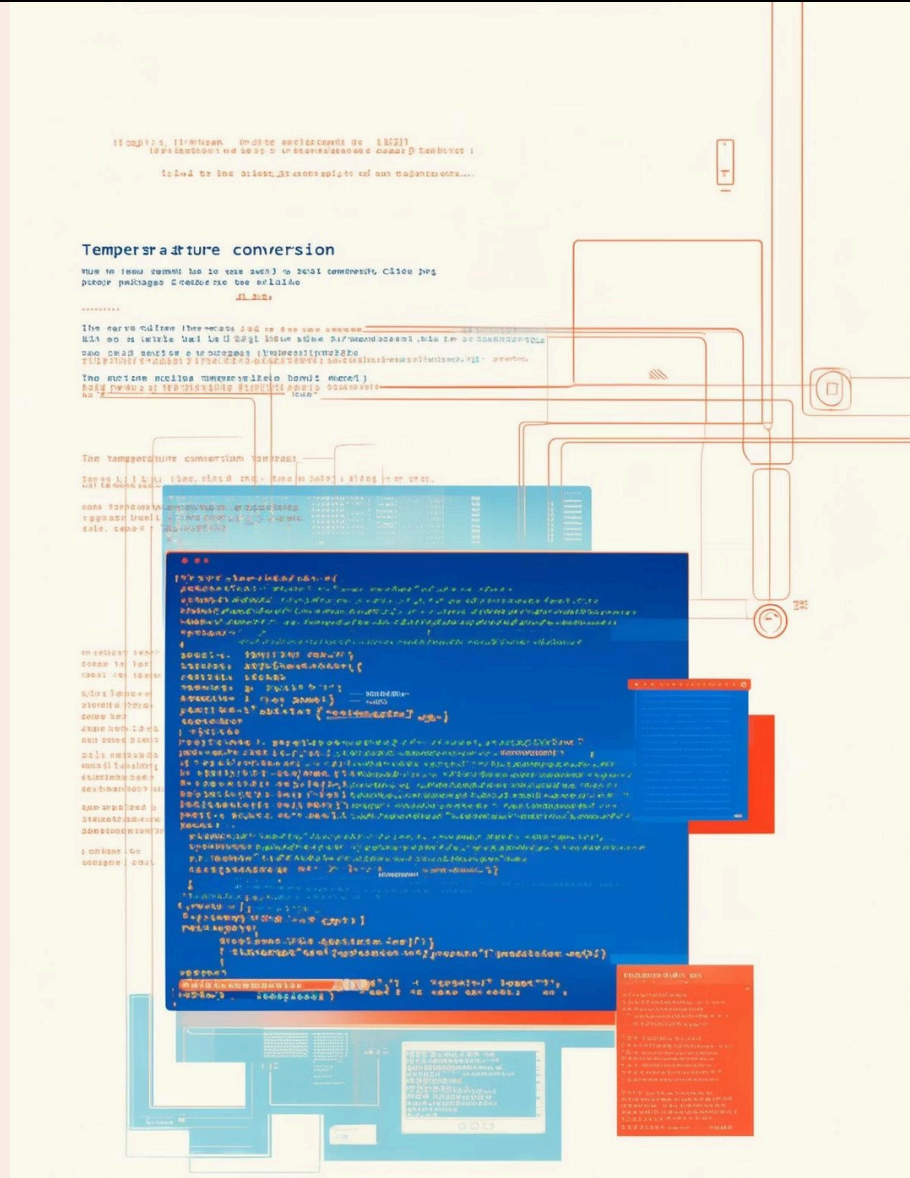
Modelagem Computacional - Celsius para Fahrenheit

Este trabalho demonstra, 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.

Gabriel Martins Delfes, Gael Huk Kukla, Thiago Bittencourt Santana, Thiago Martins Escaliante e Yann Lucas Saito da Luz



Objetivos do Projeto

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.

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.

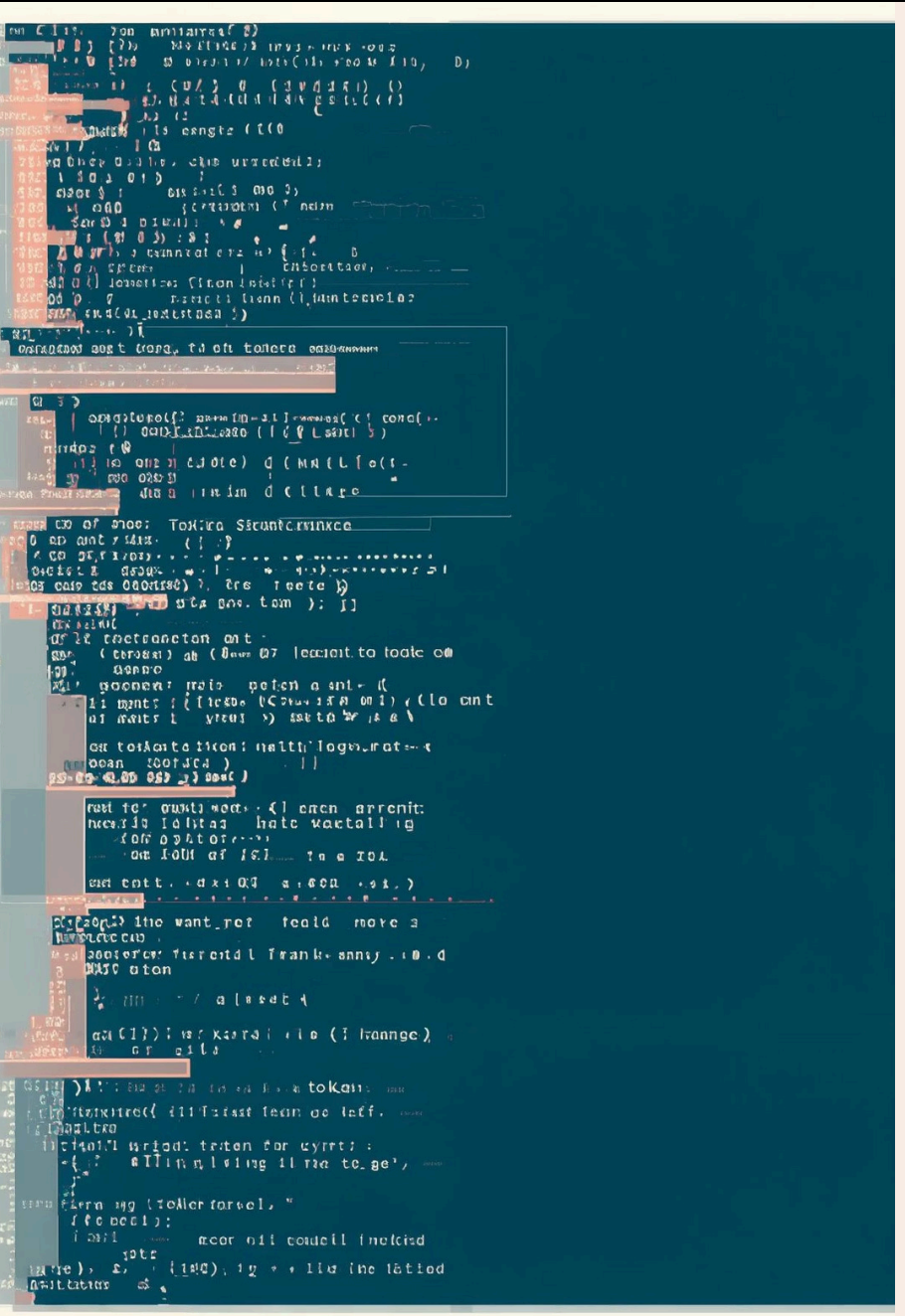
Operações matemáticas

Foram utilizadas as operações aritméticas básicas (+, -, *, /) para a realização dos cálculos de conversão entre as escalas de temperatura.

Código Fonte em C

O programa implementa um conversor de temperatura que permite ao usuário escolher entre converter de Celsius para Fahrenheit ou vice-versa. A estrutura do código utiliza um loop do-while para permitir múltiplas conversões até que o usuário decida encerrar o programa.

```
#include
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;
}
```



Análise Léxica

A análise léxica é a primeira fase do processo de compilação, responsável por transformar o código-fonte em uma sequência de tokens que serão utilizados nas fases subsequentes.

Identificação de Lexemas

O analisador léxico identifica os elementos básicos do código como identificadores, palavras-chave, operadores e constantes.

Classificação em Tokens

Cada lexema é classificado em um tipo específico de token, como TIPO, IDENTIFICADOR, OPERADOR, etc.

Criação da Tabela de Símbolos

Os identificadores são armazenados em uma tabela de símbolos para uso posterior nas análises sintática e semântica.

Geração da Sequência de Tokens

O resultado final é uma sequência ordenada de tokens que representa o programa original.

Regras de Produção e Definição de Gramática

Conjunto de regras que definem como se constrói uma linguagem formal, especificando quais as sequências de símbolos são consideradas válidas dentro daquela linguagem.

Regras de Produção e Definição de Gramática

Terminais e Não-Terminais

Símbolos Terminais: {int, float, char, printf, scanf, if, else, do, while, +, -, *, /, =, ==, !=}

Símbolos Não-Terminais: {Programa, Declaração, Comando, Expressão, Operador}

Regras de Produção

- G = (V, T, P, S)
- V = {PROGRAMA, INSTRUCAO, TIPO, ID, NUM_INT, NUM_FLOAT, CHAR, OPERADOR_REL, OPERADOR_ARIT, DELIM, STRING, CMD_IO}
- T = {"int", "float", "char", "return", "if", "else", "do", "while", "main", "#include", "<stdio.h>", "==", "!=", "<", "<=", ">", ">=", "=", "+", "-", "*", "/", ";", ",", "(", ")", "{", "}", identificadores, números inteiros, números float, caracteres, strings}
- P:

1. PROGRAMA → #include <stdio.h> TIPO main ( ) { INSTRUCAO }
2. INSTRUCAO → DECLARACAO | ATRIBUICAO | CMD_IO | CONDICIONAL | REPETICAO | RETURN | INSTRUCAO INSTRUCAO
3. DECLARACAO → TIPO ID ; | TIPO ID = VALOR ;
4. TIPO → int | float | char
5. ID → [a-zA-Z_][a-zA-Z0-9_]*
6. VALOR → NUM_INT | NUM_FLOAT | CHAR
7. NUM_INT → [0-9]+
8. NUM_FLOAT → [0-9]+.[0-9]+
9. CHAR → '[a-zA-Z0-9]'
10. ATRIBUICAO → ID = EXPRESSAO ;
11. EXPRESSAO → ID | NUM_INT | NUM_FLOAT | EXPRESSAO OPERADOR_ARIT EXPRESSAO | (EXPRESSAO)
12. OPERADOR_ARIT → + | - | * | /
13. CMD_IO → printf (STRING) ; | scanf (STRING , &ID) ;
14. STRING → ".*"
15. CONDICIONAL → if (EXPRESSAO OPERADOR_REL EXPRESSAO) { INSTRUCAO } | else if (EXPRESSAO OPERADOR_REL EXPRESSAO) { INSTRUCAO } | else { INSTRUCAO }
16. OPERADOR_REL → == | != | < | > | <= | >=
17. REPETICAO → do { INSTRUCAO } while (EXPRESSAO) ;
18. RETURN → return NUM_INT ;

- S = Programa

Tabela de Símbolos

A tabela de símbolos é uma estrutura de dados fundamental no processo de compilação, armazenando informações sobre os identificadores encontrados no programa. Ela é consultada e atualizada durante as diferentes fases da compilação.

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

Tabela de Lexemas

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]

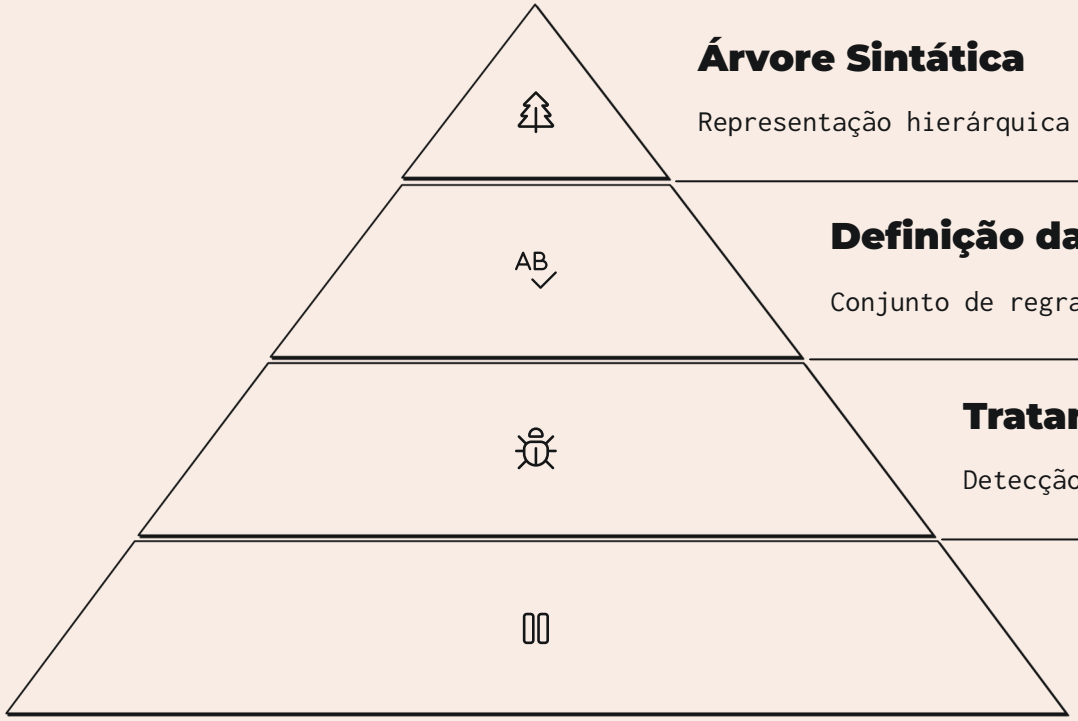
Lexema	*Token*	*Atributo*
#include	PREPROCESSOR	Instrução de pré-processamento
<stdio.h>	HEADER_FILE	Biblioteca padrão de E/S
int	TIPO	Tipo de dado inteiro
main	IDENTIFICADOR	Função principal
(	ABRE_PAR	Abertura de parêntese
)	FECHA_PAR	Fechamento de parêntese

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
```

Análise Sintática

A análise sintática verifica se a sequência de tokens gerada pela análise léxica está de acordo com a gramática da linguagem. Ela constrói uma árvore sintática que representa a estrutura do programa.



Árvore Sintática

Representação hierárquica da estrutura do programa

Definição da Gramática

Conjunto de regras que definem a sintaxe da linguagem

Tratamento de Erros

Detecção e recuperação de erros sintáticos

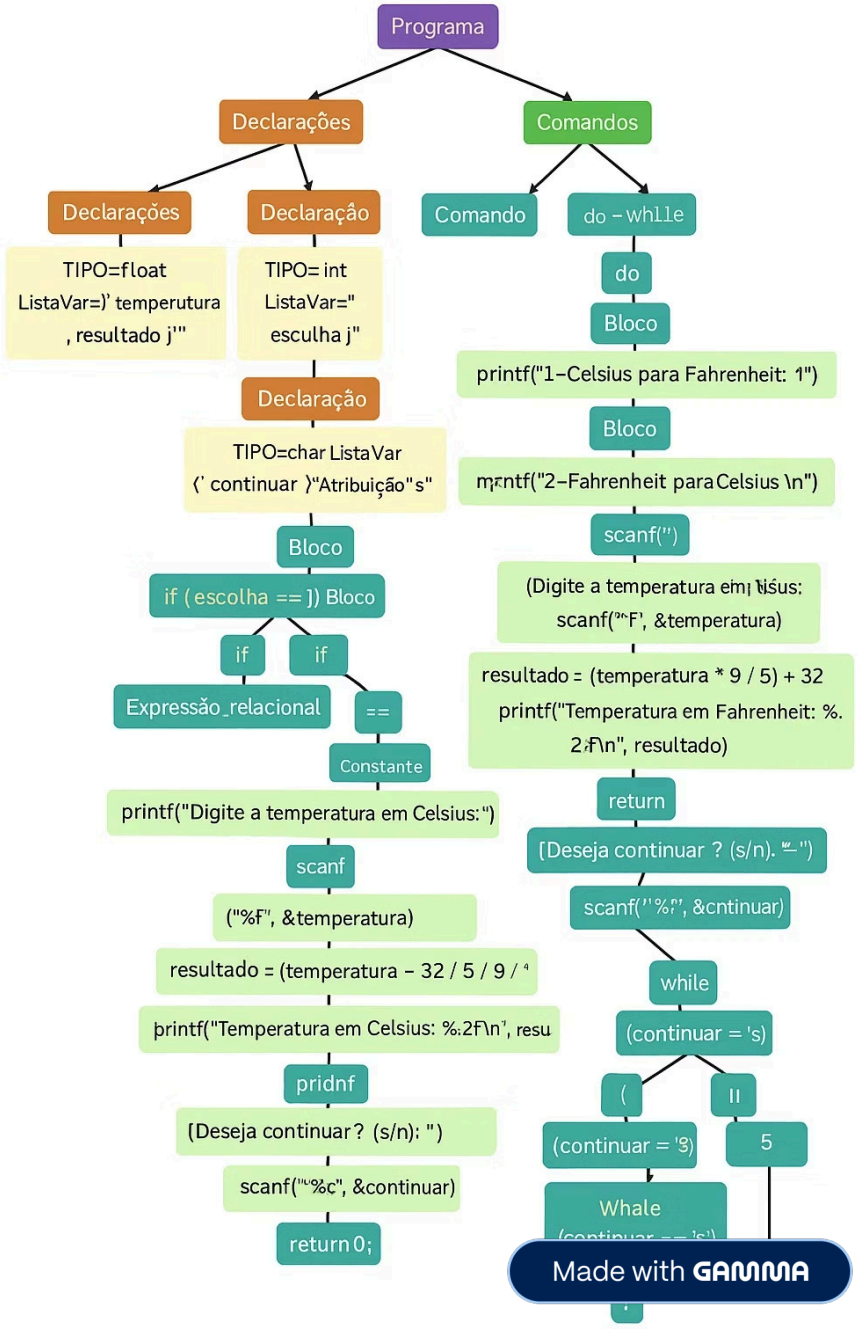
Parsing

Processo de análise da sequência de tokens

O método utilizado foi a descida recursiva (parser hand-written), pois a gramática é LL(1) após eliminação de ambiguidade nas expressões. Foram definidos conjuntos FIRST e FOLLOW para construir a tabela de parsing, garantindo one-token lookahead.

Definição da 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 → + | - | * | / | == | !=



Tratamento de Erros

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

- 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.

```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;
}
```

Erros Sintáticos

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:

```
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);
  }
}
```

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

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.

Análise Semântica

A análise semântica verifica se o programa tem significado de acordo com as regras da linguagem, como verificação de tipos e uso correto de variáveis. (Usa da Árvore Sintática e Tabela de Símbolos)

1 Declaração e uso de variáveis

Toda variável utilizada deve ter sido declarada previamente no escopo do programa. O uso de variáveis não declaradas caracteriza erro semântico.

2 Compatibilidade de tipos

As funções de conversão são restritas a expressões cujo resultado seja do tipo numérico real. A utilização de argumentos de tipo incompatível é considerada inválida.

3 Literais e identificadores

Literais numéricos são sempre aceitos, independentemente do contexto. Identificadores devem possuir um valor definido antes de sua utilização.

4 Argumentos de funções

Cada função de conversão deve receber exatamente um argumento. A omissão ou fornecimento de argumentos em número diferente constitui erro semântico.

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.

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

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)

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).

Caso Inválido (Regras Específicas da Linguagem)

```
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).

Tratamento de Erros Semânticos

- **Detecção**
- **Relatório 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
Regras Específicas da Linguagem	`result=toFahrenheit(tempF,2);`	Erro: número incorreto de argumentos para função

```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.
```

Código em Linguagem de Máquina

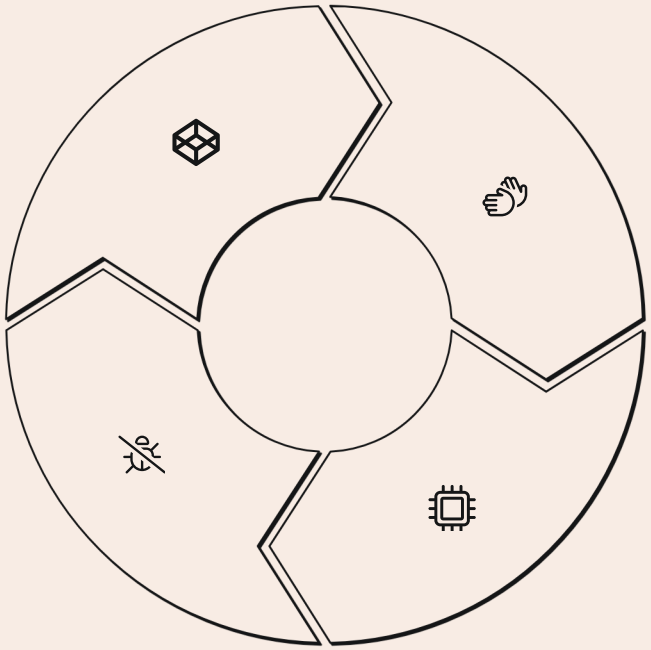
Escolheu-se Assembly MIPS32 como linguagem alvo do compilador por sua simplicidade, regularidade e ampla adoção no meio acadêmico. Sua arquitetura possui um conjunto de instruções padronizado e de fácil interpretação.

Código Fonte em C

Programa original escrito em linguagem de alto nível

Otimização

Melhoria do código para maior eficiência



Tradução para Assembly

Conversão do código para instruções MIPS

Código de Máquina

Representação binária final executável pelo processador

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 para uma execução mais clara e estruturada.

Código Assembly

```
.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
```

Código Assembly

```
# 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
```

Código Assembly

```
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
```

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

Otimização de Código

A otimização de código é uma etapa crucial no processo de compilação, visando melhorar o desempenho e a eficiência do programa gerado sem alterar sua funcionalidade.

Código Original

O código Assembly MIPS original funciona corretamente, mas apresenta ineficiências como:

- Uso excessivo de memória para variáveis
- Cálculos repetidos de constantes
- Múltiplas chamadas de sistema redundantes
- Uso ineficiente de registradores

Código Otimizado

As principais otimizações implementadas incluem:

- Pré-cálculo de constantes (9/5 e 5/9) fora do loop
- Uso exclusivo de registradores, eliminando acessos à memória
- Redução de chamadas de sistema redundantes
- Fluxo de código mais limpo e direto
- Reutilização eficiente de registradores

Código Assembly Otimizado

```
.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
```

Código Assembly Otimizado

```
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
```

Código Assembly Otimizado

```
# 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
```

Tabela 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 syscalls	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.

Análise do Processo

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.

Desafios Superados

A tradução para Assembly evidenciou os desafios e as decisões envolvidas na adaptação de construções de alto nível para uma arquitetura de baixo nível.

Resultados Alcançados

A aplicação de técnicas de otimização reforçou a importância do desempenho e da legibilidade no código gerado, cumprindo o papel do projeto como ferramenta de aprendizado e integração dos conhecimentos adquiridos.