24 Fase 4 - Geração de Código Intermediário e Assembly

Este trabalho pode ser realizado em grupos de até **4** alunos. Grupos com mais de 4 membros terão o trabalho anulado. Leia todo este documento antes de começar e siga o seguinte código de ética: você pode discutir as questões com colegas, professores e amigos, consultar livros da disciplina, bibliotecas virtuais ou físicas, e a Internet em geral, em qualquer idioma. Com ou sem o uso de ferramentas de Inteligência Artificial. Contudo, o trabalho é seu e deve ser realizado por você. Plágio resultará na anulação do trabalho.

Os trabalhos entregues serão avaliados por uma ferramenta de Inteligência Artificial, que verificará a originalidade do texto e a autoria do código e o cumprimento de todas as regras de entrega e desenvolvimento. Para isso, o trabalho deve ser entregue em um repositório público no GitHub. O resultado desta avaliação automatizada será a nota base do trabalho, que poderá ser alterada em resultado da prova de autoria (ver seção 10).

Este é um projeto "Melhorado com o uso de Ferramentas de Inteligência Artificial". O uso de ferramentas de Inteligência Artificial não é apenas permitido, mas incentivado para tarefas específicas. No entanto, seu uso é governado por um princípio fundamental: **Você é o arquiteto e o engenheiro; a Inteligência Artificial é sua assistente. Você mantém total responsabilidade pelo código que envia**. Você poderá usar as ferramentas de Inteligência Artificial para:

- Gerar código repetitivo (*boilerplate*) e arquivos de configuração;
- Explicar conceitos complexos e documentação;
- Depurar mensagens de erro e sugerir correções;
- Escrever testes unitários e documentação;
- Otimizar e refatorar código existente;
- Fazer *brainstorming* de ideias de projetos e arquitetura.

Não é permitido:

- **Envio às Cegas**: enviar código gerado por uma Inteligência Artificial sem revisão, compreensão ou modificação é uma violação da integridade acadêmica. Você deve ser capaz de explicar qualquer código do seu projeto, linha por linha. E provocará a anulação do trabalho. Explicar o código não é ler o que está escrito, mas sim entender sua lógica e funcionamento e explicar por que ele foi escrito daquela forma;
- **Contornar o Aprendizado**: usar Inteligência Artificial para completar os objetivos centrais de aprendizado por você é proibido, por exemplo, pedir a uma Inteligência Artificial para construir um componente inteiro do projeto que você tem a tarefa de aprender.
- **Violação de Licenças**: é proibido enviar código que viole as políticas de honestidade acadêmica da universidade, a licença dos dados de treinamento, ou qualquer outra restrição legal.

24.1 1. Objetivo

Pesquisar e praticar os conceitos de geração de código intermediário, otimização e geração de código Assembly desenvolvendo um programa em **Python**, **C**, ou **C++**. O objetivo é complementar o material de aula, aprimorando sua formação acadêmica e profissional por meio da construção de um gerador de código intermediário (*Three Address Code* - TAC), um otimizador de código e um gerador de código Assembly AVR para a arquitetura do Arduino Uno, utilizando como entrada a árvore sintática abstrata atribuída gerada na Fase 3.

24.2 2. Descrição do Trabalho

O objetivo é desenvolver um programa capaz de:

1. Ler um arquivo de texto contendo o código-fonte de um programa escrito na linguagem especificada nas fases anteriores (uma expressão ou declaração por linha). Este arquivo deve estar em formato texto simples (.txt) usando apenas caracteres ASCII;
2. Utilizar a árvore sintática abstrata atribuída gerada pelo analisador semântico da Fase 3 como entrada;
3. Implementar um gerador de código intermediário em formato *Three Address Code* (TAC);
4. Implementar um otimizador de código para o TAC gerado, aplicando técnicas de otimização local;
5. Implementar um gerador de código Assembly AVR compatível com o Arduino Uno (ATmega328P);
6. Compilar e executar o código Assembly no Arduino Uno, validando o funcionamento correto do compilador completo;
7. Detectar e reportar erros em todas as fases (léxica, sintática, semântica e de geração de código) de forma clara e informativa.

Observação MUITO IMPORTANTE: Esta é a fase final do projeto. Todos os quatro analisadores/geradores serão utilizados em conjunto. O arquivo de teste passará sequencialmente pelo analisador léxico (Fase 1), analisador sintático (Fase 2), analisador semântico (Fase 3) e finalmente pela geração de código (Fase 4). Portanto, é fundamental que o grupo utilize os mesmos formatos e estruturas definidos nas fases anteriores. Qualquer divergência resultará em erros de integração e perda de pontos na avaliação.

24.2.1 2.1. Características da Linguagem

As declarações continuam sendo escritas em notação polonesa reversa (RPN), no formato `(A B op)`, conforme especificado nas fases anteriores. A linguagem suporta:

Operadores Aritméticos: - **Adição**: `+` - **Subtração**: `-` - **Multiplicação**: `*` - **Divisão Real**: `|` - **Divisão Inteira**: `/` - **Resto da Divisão Inteira**: `%` - **Potenciação**: `^`

Operadores Relacionais (retornam tipo booleano): - `>` : maior que - `<` : menor que - `>=` : maior ou igual - `<=` : menor ou igual - `==` : igual - `!=` : diferente

Comandos Especiais: - `(N RES)`: Retorna o resultado da expressão N linhas anteriores - `(V MEM)`: Armazena o valor em uma memória - `(MEM)`: Retorna o valor armazenado em memória

Estruturas de Controle: - Tomada de decisão (definida na Fase 2) - Laços de repetição (definida na Fase 2)

Precisão Numérica: Para esta fase, consideramos exclusivamente a arquitetura do Arduino Uno R3 (ATmega328P, 8 bits), utilizando meia precisão (16 bits, IEEE 754) para números de ponto flutuante.

24.3 3. *Three Address Code* (TAC)

O *Three Address Code* é uma representação intermediária de código na qual cada instrução possui **no máximo três operandos**. Esta representação facilita tanto a análise quanto a otimização do código, servindo como ponte entre a árvore sintática abstrata e o código Assembly final.

24.3.1 3.1. Formato do TAC

Cada instrução TAC deve seguir um dos seguintes formatos:

- **Atribuição Simples**: `t1 = a`
- **Operação Binária**: `t1 = a op b` (onde `op` pode ser `+`, `-`, `*`, `/`, `|`, `%`, `^`, `>`, `<`, `>=`, `<=`, `==`, `!=`)
- **Operação Unária**: `t1 = op a` (onde `op` pode ser `-` para negação)
- **Cópia**: `a = b`
- **Salto Incondicional**: `goto L1`
- **Salto Condicional**: `if a goto L1` ou `ifFalse a goto L1`
- **Rótulo**: `L1:`
- **Chamada de Função**: `t1 = call funcao, n` (onde `n` é o número de parâmetros)
- **Retorno**: `return a`
- **Acesso à Memória**: `t1 = MEM[a]` e `MEM[a] = t1`

24.3.2 3.2. Variáveis Temporárias

O gerador de TAC deve criar variáveis temporárias para armazenar resultados intermediários. As variáveis temporárias devem seguir a convenção de nomenclatura `t0`, `t1`, `t2`, etc.

24.3.3 3.3. Rótulos

Rótulos são usados para identificar pontos de salto no código. Devem seguir a convenção `L0`, `L1`, `L2`, etc.

24.4 4. Otimizações

O otimizador de código deve implementar as seguintes técnicas de otimização local:

24.4.1 4.1. *Constant Folding*

Avaliar expressões constantes em tempo de compilação. Por exemplo:

- `t1 = 2 + 3` → `t1 = 5`
- `t2 = 4 * 5` → `t2 = 20`

24.4.2 4.2. *Constant Propagation*

Propagar valores constantes através do código. Por exemplo:

```
t1 = 5
t2 = t1 + 3
```

Após otimização:

```
t1 = 5
t2 = 8
```

24.4.3 4.3. *Dead Code Elimination*

Remover código que não afeta o resultado do programa. Por exemplo:

```
t1 = 5
t2 = 3
t3 = t1 + 2
```

Se `t2` nunca é usado, pode ser eliminado:

```
t1 = 5
t3 = t1 + 2
```

24.4.4 4.4. Eliminação de Saltos Redundantes

Remover saltos para a próxima instrução ou para rótulos não utilizados.

24.4.5 4.5. Documentação das Otimizações

O grupo deve documentar todas as otimizações implementadas, incluindo:

- Descrição da técnica
- Exemplos de código antes e depois da otimização
- Impacto esperado no desempenho ou tamanho do código

24.5 5. Geração de Código Assembly AVR

O gerador de código Assembly deve produzir código compatível com a arquitetura AVR do Arduino Uno (ATmega328P).

24.5.1 5.1. Características do Assembly AVR

- Arquitetura de 8 bits
- 32 registradores de uso geral (R0-R31)
- Instruções de dois operandos
- Memória de programa separada da memória de dados (arquitetura Harvard)

24.5.2 5.2. Convenções de Uso dos Registradores

O grupo deve definir e documentar convenções para uso dos registradores:

- Registradores para variáveis temporárias
- Registradores para parâmetros de função
- Registradores para valores de retorno
- Registradores reservados para o sistema

Sugestão de Convenção:

- R16-R23: Variáveis temporárias
- R24-R25: Parâmetros e valores de retorno
- R26-R27 (X): Ponteiro de endereço
- R28-R29 (Y): Frame pointer
- R30-R31 (Z): Ponteiro de pilha

24.5.3 5.3. Mapeamento TAC para Assembly

O gerador deve implementar o mapeamento de cada instrução TAC para uma ou mais instruções Assembly AVR. Exemplos:

TAC: `t1 = a + b`

Assembly:

```
ld r16, a
ld r17, b
add r16, r17
st t1, r16
```

TAC: `goto L1`

Assembly:

```
rjmp L1
```

TAC: `if a goto L1`

Assembly:

```
ld r16, a
cpi r16, 0
brne L1
```

24.5.4 5.5. Suporte a Operações de Ponto Flutuante

Para operações com números reais em meia precisão (16 bits), o gerador deve:

- Utilizar bibliotecas de ponto flutuante para AVR (se disponíveis), ou
- Implementar rotinas básicas de aritmética de ponto flutuante, ou
- Converter operações de ponto flutuante para operações com inteiros escalados

24.6 6. Arquivos de Teste

Diferentemente das fases anteriores, esta fase requer **3 arquivos de teste** específicos:

24.6.1 6.1. Teste 1: Cálculo de Fatorial

O primeiro arquivo de teste, `fatorial.txt`, deve implementar o cálculo do fatorial de números de 1 até 8. O programa deve:

- Calcular `n!` para `n` de 1 a 8
- Armazenar ou exibir os resultados
- Utilizar pelo menos uma estrutura de laço
- Demonstrar o uso de variáveis (memórias)

24.6.2 6.2. Teste 2: Sequência de Fibonacci

O segundo arquivo de testes, `fibonacci.txt`, deve calcular a sequência de Fibonacci até a 24ª posição. O programa deve:

- Calcular os 24 primeiros números da sequência de Fibonacci
- Armazenar ou exibir os resultados
- Utilizar pelo menos uma estrutura de laço
- Demonstrar o uso de variáveis (memórias)
- Opcionalmente, utilizar uma estrutura de decisão

24.7 6.3. Teste 3: Série de Taylor para o Cosseno

O terceiro arquivo de teste, `taylor.txt`, deve implementar o cálculo da **Série de Taylor para** $\cos(\mathbf{X})$ em torno de $x_0 = 0$, limitada aos **quatro primeiros termos** não nulos. O programa deve demonstrar o uso e a precisão da aritmética de ponto flutuante de 16 bits.

A fórmula truncada a ser utilizada é:

$$\cos(X) \approx 1 - \frac{X^2}{2!} + \frac{X^4}{4!} - \frac{X^6}{6!}$$

O código-fonte deve:

1. **Calcular** a aproximação de $\cos(X)$ para um valor em radianos.
2. **Utilizar** estritamente a notação RPN e o operador de **Divisão Real (`|`)** para os denominadores (fatoriais: 2.0, 24.0, 720.0).
3. **Armazenar** o valor de X e o resultado final em memórias (`MEM`) com nomes distintos (ex.: `X_VAL`, `FINAL_COS`).
4. **Enfatizar** o uso de números literais em **ponto flutuante** (ex.: `1.0`, `0.5`, `2.0`, etc.) para todas as operações aritméticas.

5. **Testar** a manipulação da precisão de **16 bits (meia precisão)**, conforme a arquitetura alvo, e a geração de código Assembly correspondente.

Observação: O resultado esperado para $\cos(0.5)$ é ≈ 0.87758. O grupo deve documentar o valor obtido na saída do compilador e no Arduino, considerando a perda de precisão inerente ao formato de 16 bits.

24.7.1 6.4. Formato dos Arquivos de Teste

- Cada arquivo deve conter código válido na linguagem especificada nas fases anteriores
- Os arquivos devem estar no mesmo diretório do código-fonte
- Os arquivos devem ser processados via argumento de linha de comando (ex.: `./compilador fatorial.txt`)

24.7.2 6.5. Validação no Arduino

Todos os programas de teste devem ser:

1. Compilados em Assembly AVR pelo compilador desenvolvido
2. Montados e linkados usando ferramentas AVR
3. Carregados no Arduino Uno
4. Executados com sucesso, produzindo os resultados corretos

Os resultados podem ser exibidos através de:

- LEDs (representando valores em binário)
- Display LCD
- Comunicação serial (mais recomendado para facilitar validação)

24.8 7. Hospedagem no GitHub

O projeto deve ser hospedado no mesmo repositório público no GitHub utilizado nas fases anteriores.

O repositório deve conter:

- Código-fonte completo do compilador (todas as 4 fases integradas)
- Todos os arquivos de teste (fatorial, Fibonacci e Série de Taylor)
- Código Assembly gerado para todos os testes (antes e depois das otimizações)
- Arquivos `.hex` prontos para upload no Arduino
- Documentação completa (README.md)
- Arquivos markdown com a representação do TAC (antes e depois das otimizações) e o código Assembly gerado

O repositório deve ser organizado com *commits* claros. As contribuições de cada aluno devem estar registradas na forma de *pull requests*. O trabalho deve ser público para permitir a avaliação automática.

24.9 8. Requisitos do Código

As primeiras linhas do código devem conter:

- Nomes dos integrantes do grupo, em ordem alfabética, seguidos do usuário deste aluno no GitHub
- Nome do grupo no ambiente virtual de aprendizagem (Canvas)

O programa deve receber o nome do arquivo de teste como argumento na linha de comando e rodar todos os analisadores/geradores sequencialmente, gerando todos os arquivos intermediários e finais.

O código deve ser escrito em **Python**, **C**, ou **C++**, mantendo a mesma linguagem utilizada nas fases anteriores.

24.10 9. Divisão de Tarefas para a Fase 4

Para resolver o problema de geração de código da Fase 4, o trabalho será dividido entre até quatro alunos. Cada aluno será responsável por uma parte específica do sistema, com interfaces claras para facilitar a integração.

> Warning
>
> **Nota**: As tarefas podem ser divididas da forma que cada grupo achar mais conveniente, desde que as funções e interfaces sejam respeitadas e todos os requisitos sejam cumpridos.

24.10.1 9.1. Aluno 1: Função `gerarTAC` e Geração de Código Intermediário

Responsabilidades:

- Implementar `gerarTAC(arvoreAtribuida)` para gerar o código intermediário em formato TAC a partir da árvore sintática abstrata atribuída
- Criar estruturas de dados para representar instruções TAC
- Implementar o algoritmo de geração de código para expressões aritméticas
- Implementar a geração de código para comandos especiais (`RES`, `MEM`)
- Gerenciar a criação de variáveis temporárias e rótulos
- Documentar o formato do TAC gerado

Tarefas Específicas:

- Percorrer a árvore sintática abstrata atribuída em pós-ordem
- Gerar instruções TAC para cada nó da árvore
- Implementar alocação de variáveis temporárias (t0, t1, t2, ...)
- Implementar geração de rótulos (L0, L1, L2, ...)
- Criar funções auxiliares para diferentes tipos de nós (operações binárias, unárias, literais, etc.)
- Testar com expressões simples e aninhadas

- Salvar o TAC gerado em arquivo texto

Interface:

- Entrada: Árvore sintática abstrata atribuída (da Fase 3)
- Saída: Lista de instruções TAC e arquivo texto contendo o TAC
- Fornece lista de instruções TAC para a função `otimizarTAC()`

24.10.2 9.2. Aluno 2: Função `otimizarTAC` e Otimização de Código

Responsabilidades:

- Implementar `otimizarTAC(tac)` para aplicar otimizações no código TAC
- Implementar *constant folding* (avaliação de expressões constantes)
- Implementar *constant propagation* (propagação de constantes)
- Implementar *dead code elimination* (eliminação de código morto)
- Implementar eliminação de saltos redundantes
- Documentar cada técnica de otimização implementada com exemplos

Tarefas Específicas:

- Analisar o TAC para identificar oportunidades de otimização
- Implementar cada técnica de otimização como uma passagem separada sobre o código
- Criar funções auxiliares para análise de uso de variáveis (liveness analysis básico)
- Manter a semântica do programa durante as otimizações
- Testar cada otimização isoladamente
- Gerar relatório comparando TAC antes e depois das otimizações
- Salvar o TAC otimizado em arquivo texto

Interface:

- Entrada: Lista de instruções TAC (do Aluno 1)
- Saída: Lista de instruções TAC otimizadas e arquivo texto contendo o TAC otimizado
- Fornece TAC otimizado para a função `gerarAssembly()`

24.10.3 9.3. Aluno 3: Função `gerarAssembly` e Geração de Código Assembly AVR

Responsabilidades:

- Implementar `gerarAssembly(tacOtimizado)` para gerar código Assembly AVR a partir do TAC otimizado
- Definir e documentar convenções de uso dos registradores AVR
- Implementar o mapeamento de cada tipo de instrução TAC para Assembly
- Implementar gestão da pilha e alocação de registradores

- Gerar código de inicialização e finalização do programa
- Criar rotinas auxiliares para operações complexas (multiplicação, divisão, operações de ponto flutuante)

Tarefas Específicas:

- Mapear cada instrução TAC para uma ou mais instruções Assembly AVR
- Implementar alocação de registradores (pode ser simples, baseado em heurísticas)
- Gerar código para prólogo e epílogo de funções
- Implementar acesso a memória (variáveis globais)
- Tratar conversões de tipo quando necessário
- Gerar comentários no Assembly indicando a instrução TAC correspondente
- Salvar o código Assembly em arquivo `.s`
- Testar a compilação do Assembly usando `avr-gcc`

Interface:

- Entrada: Lista de instruções TAC otimizadas (do Aluno 2)
- Saída: Arquivo de código Assembly AVR (`.s`) e arquivo `.hex` para upload
- O Assembly gerado deve compilar sem erros usando a toolchain AVR

24.10.4 9.4. Aluno 4: Função `main`, Integração, Testes e Validação no Arduino

Responsabilidades:

- Implementar a função `main()` que orquestra a execução de todas as fases do compilador
- Integrar todos os módulos das 4 fases
- Implementar a compilação e upload do código Assembly para o Arduino
- Criar os 2 arquivos de teste (fatorial e Fibonacci)
- Testar o compilador completo com ambos os arquivos de teste
- Validar a execução no Arduino Uno
- Gerar todos os relatórios e documentação final

Tarefas Específicas:

- Implementar o `main()` para chamar sequencialmente:
    1. Analisador Léxico (Fase 1)
    2. Analisador Sintático (Fase 2)
    3. Analisador Semântico (Fase 3)
    4. Gerador de TAC
    5. Otimizador de TAC

6. Gerador de Assembly

- Implementar tratamento de erros em cada fase
- Criar scripts para compilação e upload do Assembly
- Desenvolver os 2 programas de teste (fatorial e Fibonacci)
- Testar o compilador completo end-to-end
- Validar os resultados no Arduino Uno (usando serial, LEDs ou display)
- Gerar relatórios em markdown: TAC, TAC otimizado, Assembly, e resultados da execução
- Atualizar o README com instruções completas

Interface:

- Entrada: Nome do arquivo de código-fonte via linha de comando
- Saída: Todos os arquivos intermediários (tokens, árvore sintática, árvore atribuída, TAC, TAC otimizado, Assembly) e relatórios
- Gerencia a execução completa do compilador e validação no hardware

24.11 10. Considerações para Integração

Interfaces: Concordar com os formatos de dados entre todas as fases (estrutura da árvore sintática atribuída, formato das instruções TAC, formato do Assembly).

Depuração: Testar cada módulo isoladamente antes da integração final.

Passos de Integração:

1. Utilizar o sistema integrado das Fases 1, 2 e 3 como base
2. Integrar a função `gerarTAC()` do Aluno 1
3. Integrar a função `otimizarTAC()` do Aluno 2
4. Integrar a função `gerarAssembly()` do Aluno 3
5. Atualizar o `main()` (do Aluno 4) para orquestrar todo o processo
6. Realizar testes completos com os 2 arquivos de teste
7. Validar a execução no Arduino Uno

Resolução de Conflitos: Discutir problemas imediatamente na sala ou de forma remota.

Depuração Final: Testar o compilador completo, validando:

- A geração correta do TAC
- As otimizações aplicadas
- O código Assembly gerado
- A execução correta no Arduino Uno

24.12 11. Avaliação

O trabalho será pré-avaliado de forma automática e novamente durante a prova de autoria, com os seguintes critérios:

24.12.1 11.1. Funcionalidades do Compilador (70%)

- Implementação correta da geração de TAC: Base da nota
- Falha na geração de TAC para operações aritméticas: **-20%**
- Falha na geração de TAC para estruturas de controle: **-20%**
- Ausência de pelo menos 1 das 3 técnicas de otimização documentadas e implementadas: **-20%**
- Falha na geração de Assembly AVR: **-30%**
- Assembly gerado não compila ou não executa corretamente no Arduino: **-80%**
- Falha em um dos arquivos de teste: **-15%** por arquivo
- Código Assembly não otimizado ou ineficiente: **-10%**

24.12.2 11.2. Organização e Legibilidade do Código (15%)

- Código claro, adequadamente comentado e bem estruturado. Na dúvida, use ferramentas de IA para melhorar a qualidade dos comentários usando como referência o Google C++ Style Guide ou o PEP 8 para Python
- README bem escrito contendo, no mínimo:
  - Nome da instituição de ensino, ano, disciplina, professor
  - Integrantes do grupo em ordem alfabética
  - Instruções para compilar, executar e depurar todas as fases
  - Documentação das otimizações implementadas
  - Instruções para compilar e carregar o Assembly no Arduino
  - Documentação das convenções de registradores
- Repositório GitHub organizado com *commits* claros e *pull requests*

Atenção: A participação dos integrantes do grupo é fundamental para o sucesso do projeto. Esta participação será avaliada automaticamente. O desbalanceamento na participação dos integrantes do grupo resultará na redução da nota final do trabalho.

24.12.3 11.3. Robustez e Validação (15%)

- Tratamento de erros em todas as fases com mensagens claras
- Geração correta de todos os arquivos de saída
- Validação bem-sucedida dos 2 programas de teste no Arduino Uno
- Evidências de teste (fotos/vídeos do Arduino executando os programas, saída serial, etc.)
- Documentação do processo de validação no hardware

> **Aviso**: Trabalhos identificados como cópias terão a nota zerada.

24.13 12. Prova de Autoria

- Um aluno do grupo será sorteado usando um aplicativo online disponível em https://frankalcantara.com/sorteio.html
- Depois de explicar o projeto e responder as dúvidas do professor, o aluno sorteado escolherá um número de 1 a 10, que corresponderá a uma pergunta sobre o projeto
- A falha na resposta, ou na explicação do projeto, implicará na redução de **35%** da nota provisória que tenha sido atribuída ao projeto durante a avaliação prévia. Esta redução será aplicada para todo o grupo

> **Important**
>
> Apesar da sugestão de divisão de tarefas, todos os alunos devem entender o funcionamento completo do projeto, incluindo todas as 4 fases. O aluno sorteado para a prova de autoria deve ser capaz de responder qualquer pergunta sobre qualquer parte do projeto, desde a análise léxica até a execução no Arduino.

24.14 13. Entrega

A entrega será um link para o repositório do GitHub contendo:

24.14.1 13.1. Código-fonte

- Programa completo em **Python**, **C**, ou **C++** integrando todas as 4 fases
- Todas as funções especificadas implementadas
- Scripts de compilação e upload para Arduino

24.14.2 13.2. Arquivos de Teste

- 2 arquivos de teste conforme especificado:
    1. Cálculo de fatorial (1 a 12)
    2. Sequência de Fibonacci (até posição 15)
- Casos de teste devem demonstrar o funcionamento completo do compilador

24.14.3 13.3. Arquivos Gerados

O repositório deve conter todos os arquivos gerados na última execução do compilador, antes do envio para avaliação. Para cada arquivo de teste, deve haver:

- Arquivo de tokens (da Fase 1)
- Arquivo com a árvore sintática (da Fase 2)
- Arquivo com a árvore sintática abstrata atribuída (da Fase 3)
- Arquivo com o TAC gerado (antes das otimizações)

- Arquivo com o TAC otimizado
- Arquivo Assembly (`.s`)
- Arquivo HEX (`.hex`) pronto para upload

24.14.4 13.4. Documentação

- **README.md** bem formatado com:
  - Informações institucionais
  - Instruções de compilação e execução de todas as fases
  - Documentação das técnicas de otimização implementadas
  - Documentação das convenções de registradores AVR utilizadas
  - Instruções para compilar e carregar o código no Arduino
  - Exemplos de uso e resultados esperados

24.14.5 13.5. Relatório de Otimizações

Um arquivo markdown contendo:

- Descrição de cada técnica de otimização implementada
- Exemplos de código TAC antes e depois de cada otimização
- Estatísticas: número de instruções TAC antes e depois, número de temporários eliminados, etc.
- Análise do impacto das otimizações no código Assembly gerado

24.14.6 13.6. Formato de Execução

O programa deve ser executado com:

```
./compilador fatorial.txt
```

ou

```
./compilador fibonacci.txt
```

ou ainda

```
./compilador taylor.txt
```

O compilador deve gerar todos os arquivos intermediários e finais, incluindo o arquivo `.hex` pronto para upload no Arduino.

24.15 14. Recursos Adicionais

24.15.1 14.1. Ferramentas Necessárias

- Compilador para a linguagem escolhida (GCC/G++, Python 3.x)
- Toolchain AVR: `avr-gcc`, `avr-libc`, `avrdude`

- Arduino Uno (ou simulador como SimulIDE)
- Monitor serial ou ferramenta de comunicação USB

24.15.2 14.2. Referências Úteis

- AVR Instruction Set Manual
- Arduino Uno Datasheet
- Compiler Design: Three Address Code
- Code Optimization Techniques
- Livro: "Compilers: Principles, Techniques, and Tools" (Dragon Book) - Capítulos 6-8

24.15.3 14.3. Dicas de Implementação

Para Geração de TAC: - Use uma estrutura de dados simples para representar instruções (classe ou struct) - Mantenha contadores globais para temporários e rótulos - Processe a árvore em pós-ordem para gerar código bottom-up

Para Otimização: - Implemente cada otimização como uma passagem separada - Mantenha um mapa de valores conhecidos para constant propagation - Use análise de *liveness* simples para *dead code elimination*

Para Geração de Assembly: - Crie um template de prólogo/epílogo para o programa - Mantenha um mapeamento entre variáveis e registradores - Use a pilha quando ficar sem registradores - Adicione comentários no Assembly para facilitar depuração

Para Validação no Arduino: - Use comunicação serial para debug (`Serial.begin(9600)`, `Serial.println()`) - Teste primeiro com programas simples - Use o monitor serial do Arduino IDE para ver os resultados

24.16 15. Critérios de Sucesso

O projeto será considerado bem-sucedido se:

1. O compilador integrar corretamente todas as 4 fases
2. Todos os arquivos de teste (fatorial, Fibonacci e Taylor) compilarem sem erros
3. O código Assembly gerado executar corretamente no Arduino Uno
4. Os resultados obtidos no Arduino corresponderem aos valores esperados
5. Pelo menos 3 técnicas de otimização estiverem implementadas e documentadas
6. Toda a documentação estiver completa e clara
7. O código estiver bem organizado e comentado
8. Todos os integrantes do grupo conseguirem explicar o funcionamento do projeto