

**UNIDADE CURRICULAR:** Compilação

**CÓDIGO:** 21018

**DOCENTES:** Jorge Morais e Rúdi Gualter (tutor)

**A preencher pelo estudante**

**NOME, N.º DE ESTUDANTE, TURMA**

* Diogo Moreira, 2104140, turma 2
* José Ribeiro, 1601792, turma 1
* Silvana Oliveira, 2102785, turma 2
* Vítor Frango, 1802925, turma 1
* **EQUIPA: Compile Wizards**

**CURSO:** Licenciatura Engenharia Informática

**DATA DE ENTREGA:** 26 de junho de 2024

**TRABALHO / RESOLUÇÃO:**

Nesta que é a ultima parte do projeto da UC de Compilação, o objetivo principal é a obtenção do código Assembly gerado a partir do código intermediário TAC (Three Address Code).

Nas duas fases anteriores desenvolvemos funcionalidades para gerar a TAC a partir de código-fonte MontPy e a otimização desse TAC para melhorar a eficiência e a performance, agora com o TAC otimizada em mãos o passo seguinte foi traduzir este código intermedio para Assembly.

#### Objetivos desta Fase

1. **Gerar Código Assembly a partir do TAC Otimizado:**.
2. **Manutenção da Correção Semântica:**
3. **Otimização e Eficiência:**

#### Estrutura do Processo

1. **Leitura e Análise do TAC Otimizado:**
2. **Mapeamento para Instruções Assembly:**
3. **Geração do Arquivo Assembly:**

### Breve Análise dos Arquivos Atualizados

### **TACGenerator.java - Alterações:**

1. **generateTACFromSource(InputStream sourceCode)**

**M**étodo que permite gerar a TAC (Three Address Code) a partir de um código-fonte fornecido como um InputStream. Implementa a leitura e conversão do código-fonte MontPy em TAC, este método preenche duas listas, uma com o TAC global e outra com o TAC das funções identificadas no código-fonte.

1. **Novo Método: getGlobalTAC()**

Retorna uma lista contendo o TAC global gerado, fornece acesso ao TAC gerado para o código global permitindo a análise ou a visualização externa.

1. **getFunctionsTAC()**

Retorna um mapa contendo a TAC das funções geradas, onde a chave é o nome da função e o valor é uma lista de instruções TAC, facilita a obtenção da TAC específica para cada função, permitindo uma análise mais detalhada.

1. **Simulação e Exibição de TAC no main**

O main foi atualizado para incluir a simulação de leitura do código-fonte a partir da entrada padrão, gerar o TAC correspondente e exibir tanto o TAC global quanto o TAC das funções.

#### **TACOptimizer.java - Alterações**

#### **optimize(List<String> tac)**

Otimiza uma lista de instruções TAC, este método percorre a lista de instruções TAC e aplica otimizações um exemplo de otimização implementada é a substituição de operadores em instruções de impressão.

### Resumo das Atualizações

* **TACGenerator.java**:
  + Adição de métodos para geração de TAC a partir de código-fonte MontPy.
  + Métodos adicionais para obter TACs gerados.
  + Atualização do método main para incluir a simulação de leitura de código-fonte e exibição dos TACs gerados.
* **TACOptimizer.java**:
  + Introdução de um método para otimização de TACs.
  + Implementação de otimizações de exemplo para instruções TAC.
  + Inclusão de classes internas para representar nós no gráfico de fluxo de controle (CFG).
  + Atualização do método main para demonstrar a funcionalidade de otimização.

Essas atualizações aumentam a funcionalidade dos arquivos, permitindo a geração e otimização mais eficiente de TACs, além de facilitar a análise detalhada dos códigos gerados.

### **Conclusão**

Com a finalização desta terceira fase, completamos o ciclo de tradução do código-fonte MontPy para Assembly. Este processo detalhado garante que o código gerado é eficiente e pronto para ser executado em um ambiente real, cumprindo assim todos os objetivos do projeto de desenvolvimento de um compilador completo.

**Dificuldades Encontradas**

Durante a implementação desta que foi a ultima parte do projeto MontPy, uma das principais dificuldades encontradas foi a gestão eficiente dos registradores. Tendo em conta que na arquitetura x86-64 o número de registradores disponíveis é limitado, distribuir variáveis de forma otimizada entre registradores e memória sem termos erros em output foi um desafio interessante, desta forma e resolver esse problema desenvolvemos a função getRegister.

A função getRegister é responsável por mapear variáveis para registradores de hardware (R0 a R7) ou para posições na memória quando os registradores são insuficientes, a referida função utiliza um mapa (registerMap) para associar variáveis aos registradores correspondentes, utilizando o método computeIfAbsent para inserir novas variáveis de forma eficiente. Esta abordagem permitiu uma gestão dinâmica e adaptável dos recursos limitados de registradores, garantindo que o código Assembly gerado fosse tanto funcional quanto eficiente.

Essa solução foi fundamental para superar a limitação de registradores, permitindo que o compilador gerasse código Assembly otimizado mesmo em cenários com um grande número de variáveis.

**Anexo - Output**

**A screenshot of a computer program

Description automatically generated**

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated