Relatório de Grupo 4Devs

- Luís Henrique Pereira Marques | nº 701887
Ivo Arlindo Vieira Baptista | nº 2100927
• Pedro Pereira Santos | nº 2000809
• David Miguel Raposo Ferreira | nº 2102814

Melhorias efetuadas ao EfolioB

Relatório:

Conforme indicado na correção do eFolio B, efetuamos as seguintes correções:

• AntlrToExpression.Java

A correção feita no código consiste em diversas melhorias e ajustes nos métodos de visitação e avaliação de expressões, principalmente focando no tratamento adequado de expressões binárias e variáveis. Aqui estão alguns pontos principais das correções:

1. **Correção no Retorno de `BinaryExpression`:**
   - Originalmente, o código retornava a expressão como string usando `exp.toString()`.
   - A correção envolveu garantir que, quando a expressão for uma instância de `BinaryExpression`, o método `toString()` da expressão seja chamado corretamente.
2. **Retiramos as Mensagens de Depuração:**
   - Foram retiradas várias chamadas `System.out.println()` para imprimir mensagens de depuração, o que facilita o acompanhamento do fluxo de execução e dos valores sendo processados.
3. **Tratamento de Declarações Implícitas:**
   - A função `visitImplicitDeclaration` foi ajustada para tratar corretamente as variáveis, verificando se elas já existem e atribuindo novos valores ou inicializando-as conforme necessário.
   - Incluiu a lógica de adicionar a variável na lista `Expression.variables` se ela ainda não existir.
4. **Correções Menores em Outras Funções de Visitação:**
   - A função `visitOperationPM` foi ajustada para visitar as subexpressões à esquerda e à direita corretamente.
   - Correções similares foram aplicadas em outras funções como `visitFloatNumber`, `visitIntegerNumber` e `visitVariable`.

As correções melhoraram a robustez do código, assegurando que as expressões sejam avaliadas e manipuladas corretamente dentro do contexto do compilador. Além disso, as mensagens de depuração fornecem uma visão clara do processo de avaliação e identificação de variáveis e expressões, facilitando a depuração e manutenção do código.

Os métodos `visitFor` são essenciais para percorrer e avaliar a árvore sintática gerada pelo parser.

• ExpressionApplication.Java

1. **Verificação de Erros Semânticos:**
   - Adicionou-se uma verificação explícita de erros semânticos com mensagens informativas para indicar se erros foram encontrados ou se o processo está livre de erros. Isso melhora a robustez do código ao garantir que apenas expressões sem erros sejam processadas.
2. **Processamento e Avaliação de Expressões:**
   - O código agora imprime as expressões do programa e as avaliações resultantes, fornecendo uma saída detalhada do que está sendo processado e os resultados dessas avaliações.

As melhorias feitas no código adicionam clareza e robustez, tornando o processo de depuração mais simples e o código mais confiável. A verificação de erros semânticos garante que apenas expressões corretas sejam processadas, e as mensagens de depuração ajudam a rastrear o fluxo do programa. Essas mudanças forão essenciais para manter a integridade e a clareza do código durante o desenvolvimento e a manutenção.

• ExpressionProcessor.Java

As melhorias concentraram-se principalmente na adição de mensagens de depuração para fornecer mais informações durante a execução do programa. Estas alterações facilitaram o rastreamento e a identificação de problemas, permitindo uma melhor compreensão do fluxo do programa, todas foram eliminadas depois.

Especificamente, foram adicionadas mensagens para imprimir a lista de expressões e a expressão atual a ser processada. Além disso, foi incluída uma verificação explícita ao processar uma declaração de variável, indicando claramente quando uma variável estava a ser processada e quais eram os valores das variáveis disponíveis.

No método responsável por obter o resultado da árvore sintática abstrata (AST), foram adicionadas mensagens de depuração para indicar quando uma variável estava a ser processada e para testar o nome da variável e os valores disponíveis.

O impacto destas melhorias foi significativo:

1. **Facilidade de Depuração:**
   - As mensagens de depuração forneceram informações detalhadas sobre o estado interno do programa durante a execução, facilitando a identificação de problemas.
2. **Melhor Compreensão do Fluxo:**
   - Tornou-se mais fácil rastrear quais expressões estavam a ser processadas, como as variáveis estavam a ser manipuladas e como os resultados estavam a ser gerados.
3. **Robustez no Processamento:**
   - Garantiu-se que o processamento estava a ocorrer conforme esperado, ajudando a identificar rapidamente quaisquer inconsistências.

Estas melhorias contribuíram para tornar o código mais transparente e robusto, assegurando um processamento correto das expressões e variáveis e fornecendo uma base sólida para futuras manutenções e expansões do programa.

• **BinaryExpression.java**

As melhorias realizadas concentraram-se em várias áreas principais para aprimorar a clareza e a robustez do código:

1. **Comentário de Código Desnecessário:**
   - Remoção de verificações comentadas que eram redundantes ou desnecessárias, limpando o código e melhorando a legibilidade.
2. **Correção e Simplificação de Lógica:**
   - Ajustes na lógica de verificação de tipos de variáveis e operações binárias, assegurando que a tipagem e as operações entre variáveis sejam tratadas corretamente. Foi feita uma distinção mais clara entre variáveis do tipo inteiro e do tipo float, garantindo que as operações matemáticas fossem aplicadas corretamente com base no tipo das variáveis envolvidas.
3. **Uniformidade na Comparação de Tipos:**
   - Uniformização da forma como os tipos de variáveis são comparados e manipulados. Isso inclui a correção de casos onde o tipo da variável era diretamente comparado com strings fixas, substituindo estas comparações por verificações mais robustas e consistentes.
4. **Melhoria na Gestão de Variáveis:**
   - Melhoria no manuseio de variáveis, incluindo a correta identificação e manipulação de variáveis no lado esquerdo e direito das expressões binárias. Isso assegura que, ao realizar operações entre variáveis, o tipo e o valor de cada variável sejam considerados adequadamente.
5. **Otimização da Avaliação de Expressões:**
   - Otimização do método de avaliação de expressões binárias, garantindo que os tipos das expressões sejam corretamente avaliados antes de aplicar qualquer operação. Isto evita erros durante a execução de operações entre diferentes tipos de dados.
6. **Remoção de Redundâncias:**
   - Eliminação de código redundante e simplificação das operações, tornando o código mais conciso e eficiente.

Estas melhorias garantem que o código está mais limpo, legível e menos propenso a erros, facilitando a manutenção e futuras expansões. Além disso, asseguram que as operações entre variáveis e a avaliação de expressões sejam realizadas de forma correta e eficiente, considerando adequadamente os tipos de dados envolvidos.

• **Ficheiro Optimizer.Java**

No ficheiro `Optimizer.java`, localizado na package `TAC`, foram implementadas várias técnicas de otimização para melhorar a eficiência das instruções de código intermediário (TAC). As principais funcionalidades e melhorias incluem:

1. **Eliminação de Código Morto:**
   - Remove instruções cujas variáveis de resultado nunca são utilizadas, utilizando um conjunto para rastrear variáveis usadas.
2. **Eliminação de Instruções de Cópia Redundantes:**
   - Identifica e remove instruções de cópia redundantes, substituindo argumentos por suas cópias mapeadas e removendo instruções desnecessárias.
3. **Simplificação de Expressões Constantes:**
   - Simplifica expressões constantes, avaliando operações entre constantes e substituindo a expressão pelo resultado.
4. **Desenrolamento de Ciclos (Loop Unrolling):**
   - Expande ciclos `for` simples, substituindo a variável de índice pelo valor atual e repetindo o corpo do loop.
5. **Inlining de Funções:**
   - Substitui chamadas de função pelo corpo da função correspondente, eliminando a sobrecarga de chamadas de função.
6. **Propagação de Constantes:**
   - Substitui variáveis por seus valores constantes conhecidos, utilizando um mapa para rastrear as constantes.
7. **Remoção de Atribuições Redundantes:**
   - Remove instruções de atribuição redundantes onde uma variável é atribuída a si mesma.
8. **Otimização Peephole:**
   - Aplica otimizações locais (peephole) em pequenas janelas de código, por exemplo, substituindo adições de zero por operações nulas (NOP) e removendo instruções NOP.

Estas otimizações melhoram significativamente a eficiência do código TAC, reduzindo o número de instruções, eliminando redundâncias e melhorando o desempenho geral do programa.

• TACGenerator.java

Este já suportava as operações de expressões binárias e foi estendido para tratar expressões unárias, chamadas de funções, condicionais e ciclos.

Neste sentido, as principais melhorias e funcionalidades adicionadas incluem:

1. **Contador de Variáveis Temporárias:**
   - Adicionou-se um contador para gerar nomes únicos para variáveis temporárias, necessárias para armazenar resultados intermediários das operações.
2. **Método Principal de Geração de TAC:**
   - Implementou-se um método público `generate` que inicia o processo de geração de instruções TAC a partir de um nó da AST. Este método cria uma lista de

instruções TAC e chama recursivamente um método privado para preencher esta lista.

3. **Geração Recursiva de Instruções TAC:**
   - O método privado `generate` foi implementado para percorrer recursivamente a AST e gerar as instruções TAC correspondentes. Este método identifica o tipo de nó e gera a instrução apropriada com base no tipo de expressão.
4. **Tradução de Diferentes Tipos de Expressões:**
   - Foram adicionados casos específicos para tratar diferentes tipos de expressões, incluindo:
     - **Números Inteiros e Flutuantes:** Geração de instruções para literais numéricos.
     - **Literals de Strings:** Geração de instruções para literais de strings.
     - **Variáveis:** Geração de instruções para variáveis.
     - **Expressões Binárias e Unárias:** Geração de instruções para operações binárias (como adição e subtração) e unárias (como negação).
     - **Expressões de Lista:** Geração de instruções para listas.
     - **Chamadas de Função:** Geração de instruções para chamadas de função.
     - **Declarações de Variáveis:** Geração de instruções para declarações de variáveis.
     - **Expressões Condicionais (if-elif-else):** Geração de instruções para estruturas condicionais.
     - **Loops While e For:** Geração de instruções para loops `while` e `for`.
     - **Expressões de Retorno:** Geração de instruções para expressões de retorno.
5. **Gestão de Condições e Saltos:**
   - Implementou-se a geração de instruções para condições e saltos, incluindo a criação de rótulos únicos para pontos de salto em estruturas condicionais e loops. Estas instruções garantem que a lógica de controle de fluxo é mantida corretamente na tradução para TAC.
6. **Criação de Instruções TAC:**
   - Para cada tipo de expressão, foram geradas instruções TAC específicas que capturam a lógica da expressão original em um formato intermediário adequado para posterior tradução ou execução.

Estas melhorias permitiram a conversão eficiente de uma árvore sintática abstrata (AST) para um conjunto de instruções de código intermediário (TAC). Este processo é fundamental no pipeline de compilação, facilitando a otimização e a posterior geração de código de máquina ou de bytecode a partir da representação intermediária do programa.

- **P3 (Linguagem Escolhida) – P3Generator.java**

Criação do Ficheiro P3Generator.java
Neste ficheiro efetuamos a tradução do código TAC otimizado para comandos do processador P3.

No ficheiro `P3Generator.java`, localizado na package `TAC`, foram implementadas várias melhorias para traduzir instruções de código intermediário (TAC) para instruções na linguagem de montagem P3. As principais alterações realizadas incluem:

1. **Contador de Rótulos:**
   - Adicionou-se um contador para gerar rótulos únicos necessários em instruções de salto e loops, garantindo que cada rótulo é distinto e evitando conflitos.
2. **Método Principal de Tradução:**
   - Implementou-se o método `translate`, que percorre uma lista de instruções TAC e as converte para instruções P3. Este método lida com diferentes tipos de operações, como atribuições, operações aritméticas, condicionais, saltos e loops, garantindo uma tradução correta e eficiente de cada instrução.
3. **Tradução de Instruções Específicas:**
   - Foram adicionados métodos específicos para traduzir cada tipo de instrução TAC para o formato P3 correspondente. Estes métodos incluem a tradução de atribuições, adições, subtrações, multiplicações, divisões, estruturas condicionais `if` e `ifFalse`, saltos incondicionais, rótulos, loops `while`, `doWhile` e `for`, definições de função e instruções de retorno. Cada método assegura que a instrução TAC é corretamente convertida para uma ou mais instruções P3 equivalentes.
4. **Geração de Rótulos Únicos:**
   - Implementou-se um método para criar rótulos únicos, necessários para as instruções de salto e de loop. Este método utiliza um contador que incrementa a cada nova geração de rótulo, assegurando que cada rótulo é único e evitando sobreposições.

Estas alterações permitem uma conversão automática e eficiente de um conjunto de instruções TAC para o formato P3. Isto facilita a transição entre diferentes níveis de abstração no processo de compilação, melhorando significativamente a capacidade de gerar código de montagem organizado e eficiente a partir de representações intermediárias de programas.

Simulador para os testes ao P3: https://p3js.goncalomb.com/

Todos os ficheiros foram comentados o melhor que conseguíamos para proporcionar um identificar de opções tomadas na criação do código.