Compilador para Kotlin-Parte 2

0. Identificação

Grupo: T02_G05

Nomes: André Filipe Carvalho Guedes Borges Adrêgo
Bruno Alexandre Magalhães Costa

1. Introdução

Este projeto implementa as fases finais de um compilador para um subconjunto da linguagem **Kotlin**, focando nos seguintes aspectos:

1. **Construção da Tabela de Símbolos:** Gerenciar informações semânticas, como tipos e mutabilidade de variáveis.
2. **Geração de Código Intermediário:** Traduzir a Árvore Sintática Abstrata (AST) para um formato intermediário usando instruções de **Três Endereços (TAC)**.
3. **Geração de Código MIPS:** Converter o TAC em código MIPS, permitindo execução em simuladores como **MARS**.

2. Estrutura do Projeto

O projeto é organizado em módulos, cada um correspondendo a uma etapa específica do processo de compilação. Essa estrutura modular facilita a manutenção, o entendimento e a extensão do compilador. A seguir, detalhamos cada módulo e sua responsabilidade:

1. **Lexer (Analisador Léxico):**
    - Responsável por dividir o código-fonte em tokens.
    - Reconhece identificadores, palavras-chave, operadores, literais e símbolos de pontuação.
    - Produz uma sequência de tokens que serve como entrada para o parser.
2. **Parser (Analisador Sintático):**
    - Recebe a sequência de tokens do lexer e constrói a **Árvore Sintática Abstrata (AST)**.
    - Garante que o código segue as regras gramaticais da linguagem.
    - Define a estrutura geral do programa, incluindo comandos, expressões e blocos.
3. **Semantics (Análise Semântica):**
    - Implementa a verificação semântica usando uma **Tabela de Símbolos**.
    - Verifica:

        - Declaração e uso de variáveis.
        - Compatibilidade de tipos em operações e atribuições.
    - Armazena informações sobre os identificadores (tipo, escopo e mutabilidade).
4. **CodeGen (Gerador de Código Intermediário):**
    - Produz um conjunto de instruções intermediárias no formato de **Três Endereços (TAC)**.
    - Transforma expressões, atribuições, condicionais e loops em uma representação abstrata que é independente da arquitetura.
    - Cria rótulos e temporários para facilitar a tradução para o código de máquina.
5. **MipsGen (Gerador de Código MIPS):**
    - Converte as instruções intermediárias (TAC) em código MIPS.
    - Implementa operações aritméticas, lógicas, condicionais e controle de fluxo (saltos e rótulos).
    - Gera código para manipulação de entrada e saída (ex. leitura de valores e impressão).
    - Produz o código MIPS final, que pode ser executado em simuladores como MARS.
6. **Arquivos de Configuração e Testes:**
    - Contém exemplos de código-fonte na linguagem de entrada, usados para testar as etapas do compilador.
    - Scripts para validar o pipeline completo, do código-fonte até a execução do código MIPS.

Relação entre os Módulos

- O **Lexer** gera tokens para o **Parser**.
- O **Parser** produz a AST para o módulo **Semantics**, que realiza verificações e gera a tabela de símbolos.
- A AST e a tabela de símbolos são usadas pelo **CodeGen** para criar o código intermediário.
- Finalmente, o **MipsGen** transforma o código intermediário em código MIPS.

3. Semantics

A parte de Semantics do projeto refere-se à verificação semântica do código-fonte durante o processo de compilação. O objetivo principal é garantir que o código está semanticamente correto, isto é, que ele segue as regras de tipo e que todas as variáveis e expressões são utilizadas corretamente de acordo com o contexto e tipo definido.

Objetivo da Semântica

A verificação semântica serve para identificar erros que não são capturados pela análise sintática. Durante esta fase, o compilador verifica a consistência dos tipos, atribuições, declarações de variáveis e outras operações lógicas do programa.

Componentes e Funções Principais

1. Tabela de Símbolos (Symbol Table):
    - A tabela de símbolos é uma estrutura que armazena informações sobre todas as variáveis e funções do programa, como:
        - Nome da variável
        - Tipo de dado (Int, Float, Boolean, etc.)
        - Se é uma variável mutável ou imutável (val vs var)
    - A tabela de símbolos é atualizada durante a análise do código é usada para checar se uma variável foi corretamente declarada antes de ser utilizada.
2. Funções de Verificação:
    - checkExpr: A função responsável por validar as expressões dentro do código. Ela verifica se as operações entre as variáveis são do tipo correto. Por exemplo:
        - Operações aritméticas (como soma, subtração, multiplicação, etc.) devem ocorrer entre tipos compatíveis (ex: dois inteiros ou dois floats).
        - Operações lógicas (como &&, ||, !) devem ocorrer entre valores booleanos.
        - Operações de comparação (como ==, !=, >, <, etc.) verificam se os operandos são do mesmo tipo e se são compatíveis com a operação.
    - checkArithmetic: Verifica se operações aritméticas entre dois operandos têm tipos compatíveis e, se sim, retorna o tipo do resultado. Caso contrário, gera um erro de tipo.
    - checkComparison: Similar à função de verificação de operações aritméticas, mas para operações de comparação. Verifica se os tipos de ambos os operandos são compatíveis para uma operação de comparação.
    - validateExpr: Verifica se uma expressão tem o tipo esperado. Por exemplo, se a expressão for do tipo IntType, a função verificará se a expressão é de fato um inteiro.
3. Declaração e Atribuição:
    - As declarações e atribuições de variáveis são verificadas para garantir que não há erros de tipo e que a variável está sendo usada corretamente de acordo com sua declaração.
    - extendEnv: Esta função é responsável por atualizar a tabela de símbolos com novas variáveis e suas informações de tipo e mutabilidade. Ela é chamada quando novas variáveis são declaradas no programa.
4. Declarações de Funções e Tipos:
    - O compilador também verifica se funções são chamadas com os tipos corretos de parâmetros e se as variáveis dentro das funções são declaradas corretamente.
    - Tipos de variáveis: Verifica se variáveis são declaradas com um tipo correto e se a inicialização delas é compatível com o tipo declarado.
5. Erros Semânticos:
    - O compilador lança erros semânticos quando:
        - Uma variável é usada sem ser previamente declarada.
        - Uma variável é usada fora de seu escopo.

        - Há uma tentativa de atribuir um tipo incompatível a uma variável (por exemplo, tentar atribuir um valor de tipo Float a uma variável de tipo Int).
        - As funções são chamadas com parâmetros do tipo errado.
    - A detecção de erros semânticos garante que o programa gerado esteja sem falhas de tipo, evitando que o código rode de maneira indesejada.

4. Geração de Código Intermediário

A geração de código intermediário (ICG) é uma etapa importante no processo de compilação. Ela serve como uma ponte entre a análise semântica do código-fonte e a geração de código para a máquina alvo (neste caso, o MIPS). O objetivo dessa etapa é gerar um código intermediário de três endereços, que é uma representação do programa mais próxima da linguagem de máquina, mas ainda independente da arquitetura.

Objetivo da Geração de Código Intermediário

O objetivo principal da geração de código intermediário é transformar a árvore sintática abstrata (AST) em uma forma de código mais simples, que pode ser otimizada e traduzida para o código de máquina (MIPS, no caso). O código intermediário facilita a transformação e otimização do programa antes de ser convertido para o código final, permitindo uma análise mais eficiente e maior controle sobre a geração de código.

Estrutura do Código Intermediário

O código intermediário gerado utiliza **instruções de três endereços**, que são instruções onde cada operação envolve três operandos: dois para a operação e um para o resultado. Essas instruções geralmente se assemelham a operações aritméticas e lógicas, como ADD, SUB, MUL, DIV, e operações de controle de fluxo como IF e GOTO.

Componentes do Código Intermediário

Abaixo estão alguns dos principais componentes envolvidos na geração do código intermediário.

1. **Instruções**:
    - **Atribuições** (MOVE, MOVEI, MOVER, etc.): Usadas para mover valores entre variáveis temporárias.
    - **Operações aritméticas e lógicas** (ADD, SUB, MUL, etc.): Realizam operações entre operandos.
    - **Controle de fluxo** (JUMP, COND, LABEL): Controlam o fluxo do programa, incluindo saltos incondicionais e condicionais.
    - **Instruções de comparação** (BEQ, BNE, BLT, etc.): Usadas para operações condicionais baseadas em comparação de valores.
2. **Representação de Instruções**:
    - As instruções intermediárias são representadas de forma mais abstrata, o que permite otimizações posteriores e uma fácil conversão para código final.

Funções do Gerador de Código Intermediário

O gerador de código intermediário recebe a AST (Árvore Sintática Abstrata) gerada durante a análise sintática e traduz essas expressões em instruções intermediárias.

5. Geração de Código MIPS

A geração de código MIPS é uma das etapas finais do processo de compilação, onde o código intermediário gerado anteriormente é traduzido para o código de máquina específico da arquitetura MIPS. O código MIPS gerado pode ser executado diretamente em um simulador como o **MARS** ou em um processador físico, dependendo do ambiente de execução.

O **MIPS** (Microprocessor without Interlocked Pipeline Stages) é uma arquitetura de conjunto de instruções (ISA) amplamente usada em ensino e em sistemas de processamento de baixo custo devido à sua simplicidade e eficiência.

Objetivo da Geração de Código MIPS

O objetivo principal da geração de código MIPS é traduzir o código intermediário gerado para instruções que possam ser compreendidas pelo processador MIPS. As instruções MIPS operam diretamente sobre registradores e incluem operações aritméticas, lógicas, de controle de fluxo e acesso à memória.

Componentes do Código MIPS

As instruções do MIPS operam sobre registradores e utilizam um conjunto de operações aritméticas e lógicas. O código MIPS gerado pode incluir:

1. **Atribuições**:
    - move: Copia valores de um registrador para outro.
    - li: Carrega um valor imediato em um registrador.
2. **Operações Aritméticas**:
    - add, sub, mul, div, rem: Operações básicas entre registradores.
3. **Controle de Fluxo**:
    - j: Salto incondicional (goto).
    - beq, bne, blt, bgt, ble, bge: Saltos condicionais baseados em comparação.
4. **Comparações**:
    - Compara valores nos registradores e realiza saltos dependendo da comparação.

Funções de Geração de Código MIPS

O processo de tradução de código intermediário para MIPS envolve a conversão de operações e instruções intermediárias para a sintaxe do MIPS. As instruções de três endereços, como MOVE, ADD, e JUMP, são convertidas para instruções de MIPS.

Conclusão

O projeto de compilação desenvolvido permite transformar um subconjunto da linguagem Kotlin em código executável, passando por várias fases essenciais do processo de compilação. Desde a análise léxica e sintática até a geração de código intermediário e, finalmente, a tradução para código MIPS, o trabalho engloba os principais conceitos de um compilador. Em resumo, este projeto não só demonstrou o funcionamento básico de um compilador, mas também forneceu uma compreensão profunda das fases envolvidas na tradução de programas de alto nível para código de máquina.