Compilador para a Linguagem Lang

Relatório de Implementação

Desenvolvido por:
Maria Cecília Romão Santos (202165557C)
Maria Luísa Riolino Guimarães (202165563C)

1 Introdução

Este relatório documenta o processo de desenvolvimento e as decisões de projeto tomadas na implementação de um compilador para a linguagem *Lang*, como trabalho final da disciplina DCC045 - Teoria dos Compiladores. O projeto abrange todas as fases clássicas de um compilador, desde a análise léxica e sintática até a geração de código de alto nível (source-to-source) e baixo nível.

As seções a seguir descrevem a arquitetura e as estratégias adotadas para cada módulo do compilador, incluindo o analisador sintático, a estrutura da Árvore de Sintaxe Abstrata (AST), o interpretador, o analisador semântico e o gerador de código para Python e Java Virtual Machine (JVM).

2 Estrutura do Projeto e Compilação

Esta seção detalha os pré-requisitos de software, a organização dos diretórios do projeto e as instruções para compilar e executar o compilador a partir da linha de comando.

2.1 Requisitos

Para compilar e executar o projeto, os seguintes softwares devem estar instalados no ambiente:

- Java Development Kit (JDK) 17 ou superior
- Apache Maven 3.8 ou superior
- GNU Make

2.2 Estrutura de Pastas do Projeto

O projeto foi estruturado seguindo as convenções do Maven, com uma organização modular para separar as diferentes fases do compilador. A gramática ANTLR está localizada em src/main/antlr4, enquanto o código-fonte Java reside em src/main/java. A estrutura dos pacotes foi projetada para garantir uma clara separação de responsabilidades:

```
lang-compiler/
 | --- Makefile
 | --- pom.xml
 | --- outputs/              -- Diretório para arquivos gerados (.py, .j)
 | --- src/main/
    | --- antlr4/
    | --- .../parser/Lang.g4
    | --- java/br/ufjf/lang/compiler/
        | --- analyzer/       -- Analisador Semântico
        | --- ast/            -- Árvore de Sintaxe Abstrata (AST)
        | --- cli/            -- Lógica da Linha de Comando (Main)
        | --- generator/      -- Geradores de Código (S2S, Jasmin)
        | --- interpreter/    -- Interpretador e Valores de Runtime
        | --- parser/         -- Arquivos gerados pelo ANTLR
```

2.3 Instruções de Compilação e Execução

O projeto utiliza um Makefile para simplificar os comandos do Maven. Para compilar todo o projeto, incluindo a geração das classes do ANTLR e a compilação do código Java, execute os seguintes comandos na raiz do projeto:

```
make clean
make generate
make compile
```

Após a compilação, o compilador pode ser executado em diferentes modos, conforme as diretivas especificadas no enunciado do trabalho.

- **Análise Sintática (-syn**): Verifica se o arquivo .lan é sintaticamente válido.

  ```
  make run-syn file=caminho/para/seu/arquivo.lan
  ```

- **Interpretação (-i):** Executa o programa .lan e imprime a saída no terminal.

  ```
  make run-i file=caminho/para/seu/arquivo.lan
  ```

- **Análise Semântica (-t):** Verifica se o programa é semanticamente válido (tipos, escopos, etc.).

  ```
  make run-t file=caminho/para/seu/arquivo.lan
  ```

- **Geração Source-to-Source (-src):** Traduz o programa .lan para um arquivo Python (.py).

  ```
  make run-src file=caminho/para/seu/arquivo.lan
  ```

- **Geração de Código de Baixo Nível (-gen):** Traduz o programa .lan para JVM

  ```
  make run-gen file=caminho/para/seu/arquivo.lan
  ```

3 Analisador Léxico e Sintático (Diretiva -syn)

A primeira fase do compilador é responsável por validar a estrutura do código-fonte. A ferramenta escolhida para esta etapa foi o **ANTLR v4**. Esta decisão foi motivada pela sua capacidade de gerar analisadores eficientes a partir de uma única especificação gramatical, integrando as fases léxica e sintática de forma robusta e simplificando a gestão de precedência e associatividade de operadores.

3.1 Definição da Gramática

Foi criada uma gramática formal (arquivo Lang.g4), na qual as regras léxicas foram definidas para todos os tokens, como identificadores (ID), nomes de tipos (TYID) e literais (INT, FLOAT, CHAR).

Uma atenção especial foi dada à regra de literais de caractere para suportar tanto escapes padrão (por exemplo, '\n') quanto a notação de código ASCII de três dígitos ('\ddd'). As regras sintáticas foram estruturadas para refletir a precedência e a associatividade dos operadores, conforme a Tabela 1 do documento de especificação da linguagem.

3.2 Validação e Saída

Para a diretiva **-syn**, o requisito é validar se o programa de entrada é sintaticamente correto, com a saída sendo *accept* ou *reject*. A estratégia implementada foi adicionar um *ErrorListener* customizado ao parser do ANTLR. Em caso de qualquer erro sintático, este listener imprime *reject* e lança uma exceção para encerrar a execução. Se o parsing for concluído sem que o listener seja acionado, o programa imprime *accept*.

4 Árvore de Sintaxe Abstrata (AST)

Nesta implementação, cada fase do compilador é independente e desacoplada dos detalhes da árvore de parsing do ANTLR. Para isso, foi implementada uma hierarquia de classes customizada para formar uma Árvore de Sintaxe Abstrata (AST).

A conversão da árvore do ANTLR para a nossa AST é realizada por uma classe dedicada, AstBuilderVisitor, que utiliza o padrão de projeto *Visitor*. Esta separação simplificou de forma significativa a implementação das fases subsequentes, como a interpretação e a análise semântica.

As classes base abstratas (ExprBase, LValueBase) foram criadas para centralizar a declaração de campos compartilhados, como o campo type usado na análise semântica. Isso resolveu bugs de *Field Hiding* em Java e garantiu que todas as fases do compilador operassem sobre a mesma instância de dados na AST.

5 Interpretador (Diretiva -i)

A estratégia de interpretação escolhida foi a de um **visitante da AST** (tree-walking interpreter). Uma classe Interpreter percorre a AST de forma recursiva e executa diretamente a semântica de cada nó.

A arquitetura desta etapa do compilador possui como principais características:

- **Separação entre AST e Valores de Runtime:** Foi feita uma distinção clara entre as classes da AST (pacote ast) e as classes que representam os valores manipulados durante a execução (declaradas ao início do arquivo Interpreter.java). A hierarquia Value (ValueInt, ValueFloat, ValueArray, etc.) modela os valores dinâmicos, conforme a gramática de valores da especificação. Esta separação evitou ambiguidades e tornou o código mais limpo.
- **Gerenciamento de Escopo:** Para modelar o escopo léxico aninhado da linguagem, foi implementada uma Tabela de Símbolos de tempo de execução baseada em uma Pilha de Maps (`Stack<Map<String, Value>>`). A cada entrada em um novo escopo (chamada de função ou bloco {}), um novo HashMap é empilhado (*push*). Ao sair, o mapa do topo é removido (*pop*). A busca por variáveis percorre a pilha de cima para baixo, garantindo a visibilidade correta das variáveis conforme a especificação.

6 Analisador Semântico (Diretiva -t)

A arquitetura do analisador semântico também segue o padrão *Visitor* da AST. Sua função é aplicar as regras da semântica estática da linguagem, verificando erros de tipo, escopo e fluxo de controle.

As decisões de projeto mais importantes foram:

- **Análise em Múltiplos Passes:** Foi adotada uma estratégia de dois passes. O **primeiro passe** percorre as declarações de topo (*data* e *fun*) e popula tabelas globais com as definições de tipos e as assinaturas de todas as funções. Isso resolve o problema de referências futuras, como usar uma função ou tipo antes de sua declaração no código, e permite a validação da unicidade de nomes.
- **"Decoração" da AST:** Uma decisão arquitetural fundamental foi fazer com que o analisador semântico "decorasse" a AST. À medida que o tipo de cada expressão e *l-value* é verificado, a informação de *Type* é armazenada em um campo diretamente no nó correspondente da árvore. Essa AST "enriquecida" torna-se uma representação intermediária que é posteriormente utilizada nas fases de geração de código, que consomem essa informação diretamente.
- **Inferência de Tipos:** Conforme os exemplos práticos na especificação da linguagem, foi implementada a declaração de variáveis por inferência de tipo na primeira atribuição. A lógica no analisador diferencia entre uma atribuição a uma variável existente (onde os tipos devem ser compatíveis) e a uma nova variável (onde o tipo é inferido e armazenado na tabela de símbolos).

- **Análise de Fluxo de Controle:** Foi implementada uma verificação para garantir que todas as funções com tipo de retorno declarado terminem com uma instrução return em todos os caminhos de execução possíveis. A implementação também emite um *warning* para código inalcançável após uma instrução return, melhorando a qualidade do feedback do compilador.

7 Gerador Source-to-Source (Diretiva -src)

Para a geração de código de alto nível, a linguagem alvo escolhida foi **Python**, devido à sua sintaxe limpa e ao mapeamento direto de estruturas da linguagem *Lang*. A implementação também se baseia em um visitante da AST, a classe **S2SGenerator**.

As principais decisões de projeto foram:

- **Uso da AST Decorada:** Para que esta etapa funcione de maneira simplificada, o gerador de código *source-to-source* confia inteiramente na AST decorada produzida pelo analisador semântico (a análise semântica é obrigatoriamente executada antes da etapa de *codeGen*). Isso simplifica a tradução, pois o gerador pode usar a informação de tipo para tomar decisões, como escolher entre a divisão de inteiros (//) e a de ponto flutuante (/) do Python, garantindo a correção semântica do código gerado.
- **Tradução de Estruturas:** As estruturas de *Lang* foram mapeadas para seus equivalentes em Python: `data` vira `class`; funções continuam sendo funções; `iterate` vira `for ... in range(...)` ou `for ... in ...;` e chamadas de função com múltiplos retornos são traduzidas para o desempacotamento de tuplas do Python `(a, b = func())`.
- **Tratamento de Palavras-Chave:** Foi implementada uma técnica de *name mangling* para evitar conflitos com as palavras-chave do Python. Se um identificador em *Lang* (ex: or) for uma palavra-chave em Python, ele é traduzido com um *underscore* no final (ex: or_), garantindo que o código gerado seja sempre sintaticamente válido.

8 Gerador de Código de Baixo Nível (Diretiva -gen)

A fase final do compilador, conforme especificado no enunciado do trabalho, é a geração de código de baixo nível para a Java Virtual Machine (JVM), utilizando a sintaxe do assembler **Jasmin**. Esta etapa é a que mais se aproxima de um compilador tradicional, traduzindo as construções de alto nível da linguagem *Lang* para um conjunto de instruções de uma máquina de pilha.

A estratégia de implementação, assim como nas fases anteriores, baseia-se no padrão *Visitor* sobre a AST. Uma classe `JasminGenerator` percorre a árvore, que já foi validada e decorada pelo analisador semântico, e emite o código Jasmin correspondente.

8.1 Decisões de Projeto e Arquitetura

- **Confiança na AST Decorada:** A decisão mais crítica para esta fase foi depender inteiramente da AST "enriquecida" pelo analisador semântico. O gerador de código para Jasmin pode operar de forma "ingênua" e direta, pois confia que o programa de entrada é semanticamente válido. A informação de tipo, armazenada em cada nó de expressão, é fundamental para decidir qual instrução da JVM gerar (ex: `iadd` para soma de inteiros vs. `fadd` para soma de floats).
- **Mapeamento para a Máquina de Pilha:** A principal complexidade desta fase é traduzir as expressões de *Lang*, que são baseadas em árvores, para a arquitetura de uma máquina de pilha como a JVM. A estratégia adotada foi a seguinte: para uma expressão binária como `a + b`, o gerador primeiro emite o código para avaliar `a` (empilhando seu valor), depois o código para avaliar `b` (empilhando seu valor) e, finalmente, emite a instrução que consome os dois valores do topo da pilha e empilha o resultado (ex: `iadd`).

- **Gerenciamento de Variáveis:** As variáveis locais de *Lang* são mapeadas para o array de variáveis locais da JVM. O gerador de código mantém um mapa que associa cada nome de variável a um índice nesse array (ex: `0` para 'this', `1` para o primeiro parâmetro, e assim por diante). As operações de leitura e escrita de variáveis são traduzidas para as instruções `load` e `store` da JVM (ex: `iload_1`, `istore_2`).
- **Tradução de Estruturas de Controle:** Estruturas de controle de fluxo como `if` e `iterate` foram traduzidas usando as instruções de desvio condicional e incondicional da JVM. Isso requer a geração de *labels* (rótulos) únicos para marcar os pontos de início e fim dos blocos de código. Por exemplo, um `if (cond) then {...} else {...}` é traduzido para:
  1. Código que avalia `cond` e empilha 0 ou 1.
  2. Uma instrução como `ifeq else_label` (se for falso, pule para o bloco else).
  3. O código do bloco `then`.
  4. Uma instrução `goto end_if_label` para pular o bloco else.
  5. A `else_label`.
  6. O código do bloco `else`.
  7. A `end_if_label`.
- **Mapeamento de Tipos de Dados:** Os tipos de dados registro (`data`) de *Lang* são naturalmente mapeados para classes Java. Para cada `data` definido, o gerador produz um arquivo `.j` separado que descreve uma classe com os campos correspondentes. A expressão `new` de *Lang* é traduzida para a instrução `new` da JVM, seguida pela chamada ao construtor `<init>`.

9 Conclusão

O desenvolvimento do compilador para a linguagem *Lang* seguiu uma abordagem modular e incremental, com uma clara separação de responsabilidades entre as fases. As decisões de projeto, como o uso de uma AST customizada, a estratégia de decoração da árvore pelo analisador semântico e a implementação de visitantes para cada fase, resultaram em um código organizado e robusto. Cada fase foi validada com o conjunto de testes disponibilizado, garantindo a conformidade com a especificação da linguagem e a correção das funcionalidades implementadas.

Os principais desafios enfrentados são intrínsecos à complexidade do próprio processo de construção de um compilador. A cada nova diretiva implementada, surgiam inconsistências que exigiam ajustes e, por vezes, refatorações completas nas etapas anteriores. Este processo iterativo de implementação, depuração e refatoração foi a maior fonte de aprendizado. Ele demonstrou na prática a interdependência entre as fases de um compilador e a importância de uma arquitetura sólida e desacoplada, o que nos permitiu compreender como funcionam os diferentes componentes de um compilador e os desafios que este tipo de desenvolvimento apresenta.