deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

Compilador de Linguagem IML

Compiladores

Duarte Gabriel Castro Branco - 119253 (33.3%)
Henrique Miguel Senra Reveles - 119786 (33.3%)
Diogo Coelho Aires de Nascimento - 120031 (33.3%)

2024/2025

Conteúdo

1. Introdução

1.1 Introdução Geral do Compilador

Este relatório contém informação relacionada à nossa implementação de uma linguagem para tratamento de imagens em tons de cinza.

A linguagem iml é uma linguagem de programação compilada que gera programas na linguagem Python, com foco na manipulação de imagens em tons de cinza (formato PGM).

Estas imagens são representadas como matrizes de valores reais no intervalo [0, 1]. Cada valor individual corresponde a um pixel da imagem, onde 0 corresponde ao preto e 1 ao branco. Uma característica distintiva do tipo `image` é a sua natureza saturante, mantendo os valores dos pixels sempre dentro do intervalo [0, 1].

A linguagem iml oferece diversas operações especializadas para imagens:

- Operações pixel a pixel prefixadas por um ponto (como `.+`, `.*`)
- Operadores unários para inversão de imagem (`.-`)
- Operadores de escala (`-*`, `|*`, `+*`)
- Operadores de flip (`-`, `|`, `+`)
- Operações morfológicas (`erode`, `dilate`, `open`, `close`)

A linguagem secundária iiml é uma versão interpretada simplificada da linguagem principal, permitindo criar imagens e colocar figuras geométricas simples (retângulo, círculo, cruz e mais), além de operações aritméticas e entrada/saída de dados. Esta linguagem secundária é implementada em Python.

1.2 Características Mínimas

Estas características do Compilador são requeridas para o seu funcionamento incondicional. Estas funcionalidades incluem:

- Instruções para carregar e salvar imagens em tons de cinzento (.pgm)
- Implementação dos tipos de dados image, string, number e percentage
- Operações morfológicas, escala, flip e aritméticas pixel a pixel
- Expressões aritméticas standard para tipos numéricos e concatenação de texto
- Instruções de escrita e leitura no standard input/output
- Operadores de conversão entre tipos de dados
- Instruções para desenhar uma imagem
- Execução da linguagem secundária (iiml) interpretada pela linguagem destino
- Verificação semântica do sistema de tipos

Para a linguagem interpretada iiml:

- Instrução para a criação de uma imagem
- Instrução para desenhar figuras geométricas

1.3 Características Desejáveis

Estas características do Compilador são complementares, sendo o seu funcionamento, enquanto essencial, não requerido sem a implementação das mínimas. Estas funcionalidades incluem:

- Expressões booleanas (predicados) com relações de ordem e operadores booleanos
- Operadores `any pixel` e `all pixel` para verificar valores em imagens ou listas booleanas
- Operador `count pixel in` para contar pixels de determinada intensidade
- Instrução condicional `if-then-else-done`
- Implementação de listas como novo tipo de dados, com adição e remoção de elementos
- Matrizes como listas de listas com operador de indexação recursivo
- Armazenamento de lista de imagens em formato GIF
- Instruções iterativas `for` e `until`
- Operações morfológicas adicionais: `top hat` e `black hat`

1.4 Características Avançadas

Estas características do Compilador não são requeridas nem essenciais para o funcionamento do compilador, e apenas servem para adicionar profundidade e complexidade às suas capacidades, cujo incluem:

- Funções e variáveis locais
- Tabela de símbolos para resolução de contextos de declaração
- Tratamento de erros de runtime (como "File not found")
- Suporte para imagens coloridas (RGB) com todas as operações definidas

2. Implementação das Características Mínimas

2.1 Arquitetura do Compilador

O nosso projeto tem dois componentes principais:

- **iml**: O compilador da linguagem principal que gera código Python
- **iiml**: O interpretador da linguagem secundária implementado em Python

O projeto segue uma estrutura modular com os seguintes componentes:

```
/comp2425-iml-a4
|
|-- build.sh                    # Script de compilação do código
|-- compile.sh                  # Aplicar o compilador em ficheiros
|-- run.sh                      # Compilar e executar ficheiro
|
|-- src/
|   |-- iml/                    # Compilador IML (Java)
|   |   |-- iml.g4              # Gramática ANTLR da linguagem IML
|       |-- imlMain.java        # Ponto de entrada do compilador
|   |   |-- CodeGenVisitor.java # Geração de código Python
|   |   |-- TypeInferenceVisitor# Verificação de tipos
|   |   |-- ...
|   |
|   |-- iiml/                   # Interpretador IIML (Python)
|       |-- iiml.g4             # Gramática ANTLR da linguagem IIML
|       |-- iimlMain.py         # Ponto de entrada do interpretador
|       |-- IimlEvalVisitor.py  # Implementação do interpretador
|       |-- run_iiml.py         # Script para executar programas IIML
|
|-- examples/                   # Exemplos de programas IML e IIML
|-- output/                     # Output de execuções; ficheiros py/imagens
|-- doc/                        # Este relatório
|-- ...
```

Listing 2.1: Estrutura do Projeto

2.2 Gramática das Linguagens

2.2.1 Gramática iml

A linguagem iml possui uma gramática que suporta declarações de variáveis, estruturas de controle, operações de manipulação de imagens e operações aritméticas. A gramática foi implementada em ANTLR4, definindo regras para:

```
// Estrutura do programa
program
    : statement* EOF
    ;

// Declarações e estruturas de controle
statement
    : variableDeclaration
    | assignment
    | ifStatement
    | forStatement
    | untilStatement
    | storeStatement
    | outputStatement
    | COMMENT
    ;

// Tipos de dados
type
    : 'image'
    | 'number'
    | 'string'
    | 'percentage'
    | 'list' 'of' type
    ;

// Operações específicas de imagem
imageOperation
    : 'load' 'from' expression                      #loadOp
    | 'run' 'from' expression                       #runOp
    | expression 'erode' 'by' expression            #erodeOp
    | expression 'dilate' 'by' expression           #dilateOp
    | expression 'open' 'by' expression             #openOp
    | expression 'close' 'by' expression            #closeOp
    ...
```

Listing 2.2: Trechos da Gramática IML

2.2.2 Gramática iiml

A linguagem iiml é mais simples, focada na criação e manipulação básica de imagens:

```
program
    : statement* EOF
    ;

statement
    : variableDeclaration
    | assignment
    | imageStatement
    | placeStatement
    | outputStatement
    | forStatement
    | COMMENT
    ;

imageStatement
    : 'image' 'size' expression 'by' expression 'background' expression
    ;

placeStatement
    : 'place' shape (shapeSize) 'at' expression expression 'with' '
       intensity' expression
    ;

shapeSize
    : 'radius' expression
    | 'width' expression 'height' expression
    ;

shape
    : 'circle'
    | 'rect'
    | 'cross'
    | 'plus'
    ;
```

Listing 2.3: Trechos da Gramática IIML

2.3 Implementação do Compilador iml

O compilador iml traduz código da linguagem iml para Python, utilizando principalmente os seguintes componentes:

- **TypeInferenceVisitor**: Para verificação e inferência de tipos
- **ValidationListener**: Para validação semântica
- **CodeGenVisitor**: Para geração de código Python

2.3.1 Verificação de Tipos

A verificação de tipos é implementada pelo TypeInferenceVisitor, que percorre a árvore sintática para inferir e verificar tipos. Por exemplo:

```java
@Override
public Type visitVariableDeclaration(imlParser.
    VariableDeclarationContext ctx) {
    Type declaredType = visit(ctx.type());
    Type valueType = visit(ctx.expression());

    if (!typeCompatible(declaredType, valueType)) {
        throw new RuntimeException("Type mismatch in variable
            declaration. Expected "
            + declaredType + " but got " + valueType);
    }

    // Add to symbol table
    symbolTable.put(ctx.ID().getText(), declaredType);
    return declaredType;
}
```

Listing 2.4: Trecho do Visitor de Tipos

2.3.2 Geração de Código

A geração de código é implementada pelo CodeGenVisitor, convertendo a árvore sintática em código Python equivalente:

```java
@Override
public String visitImageOperation(imlParser.ImageOperationContext ctx)
    {
    if (ctx.loadOp() != null) {
        String path = visit(ctx.loadOp().expression());
        return String.format("skimage.io.imread(%s, as_gray=True) /
            255.0", path);
    }
    else if (ctx.erodeOp() != null) {
        String image = visit(ctx.erodeOp().expression(0));
        String kernel = visit(ctx.erodeOp().expression(1));
        return String.format("np.clip(morphology.erosion(%s, %s), 0, 1)
            ", image, kernel);
    }
    // ...outros casos
}
```

Listing 2.5: Trecho do Gerador de Código

Estes códigos Python são escritos para a pasta output/.

2.4 Exemplos Avançados da Linguagem IML

2.4.1 Operações Morfológicas em IML

A linguagem IML suporta diversas operações morfológicas para processamento de imagens:

```
// Carregar imagem de entrada
image img1 is load from "input.pgm"

// Criar um kernel para operações morfológicas
list of list of number kernel is [[0, 1, 0], [1, 1, 1], [0, 1, 0]]

// Aplicar operações morfológicas
image eroded is img1 erode by kernel
image dilated is img1 dilate by kernel
image opened is img1 open by kernel
image closed is img1 close by kernel

// Operações morfológicas avançadas
image tophat is img1 top hat by kernel
image blackhat is img1 black hat by kernel

// Salvar resultados
eroded store into "eroded.pgm"
dilated store into "dilated.pgm"
```

Listing 2.6: Exemplo de Operações Morfológicas em IML

2.4.2 Controle de Fluxo em IML

A linguagem IML oferece estruturas de controle de fluxo como if-then-else e loops:

```
// Declarar variáveis
number threshold is 0.5
image img is load from "input.pgm"

// Estrutura condicional if-then-else
if any pixel img > threshold then
    output "Imagem contém pixels acima do limiar"
    image binarized is img > threshold
    binarized store into "binarized.pgm"
else
    output "Imagem não contém pixels acima do limiar"
done

// Estrutura de repetição for
list of string filenames is ["img1.pgm", "img2.pgm", "img3.pgm"]
list of image images is []

for string filename within filenames do
    image current is load from filename
    images append current
done

// Estrutura until
number i is 0
until i > 5 do
    output "Processando iteração " + i
    i is i + 1
done
```

Listing 2.7: Exemplo de Estruturas de Controle em IML

2.4.3 Manipulação de Listas e Matrizes em IML

IML suporta operações com listas e matrizes (listas de listas):

```
// Criar uma lista simples
list of number values is [0.1, 0.5, 0.8, 0.3]

// Acessar elementos da lista
number first is values[0]
number last is values[3]

// Criar uma matriz (lista de listas)
list of list of number matrix is [
    [1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]
]

// Acessar elementos da matriz
number diagonal1 is matrix[0][0]
number diagonal2 is matrix[1][1]
number diagonal3 is matrix[2][2]

// Operações com listas
output "Média dos valores: " + (values[0] + values[1] + values[2] +
    values[3]) / 4

// Criar um GIF a partir de uma lista de imagens
list of image frames is []
for number i within [1, 2, 3, 4, 5] do
    image frame is load from "frame" + i + ".pgm"
    frames append frame
done

// Salvar como GIF animado
frames store into "animation.gif"
```

Listing 2.8: Exemplo de Manipulação de Listas em IML

2.4.4 Funções em IML

IML suporta a definição e uso de funções:

```
// Definição de função para redimensionar uma imagem
function resize(image input, number scale) returns image is
    image output is input *scale
    return output
end

// Definição de função para aplicar um filtro de média
function average_filter(image input, number size) returns image is
    list of list of number kernel is []

    // Criar kernel de média
    for number i within [0, 1, 2] do
        list of number row is []
        for number j within [0, 1, 2] do
            row append 1/9
        done
        kernel append row
    done

    // Aplicar convolução
    image result is input open by kernel
    return result
end

// Uso das funções definidas
image original is load from "input.pgm"
image small is resize(original, 0.5)
image filtered is average_filter(original, 3)

// Salvar resultados
small store into "small.pgm"
filtered store into "filtered.pgm"
```

Listing 2.9: Exemplo de Funções em IML

2.5 Implementação do Interpretador iiml

O interpretador iiml foi implementado em Python como um visitor ANTLR4 que processa diretamente a árvore sintática e executa operações. Principais componentes:

- **IimlEvalVisitor**: Visitor principal que implementa a lógica para cada construção da linguagem
- **Gerenciamento de Variáveis**: Usando um dicionário Python para armazenar variáveis
- **Gestão de imagens**: Utilizando NumPy e PIL para representação e manipulação

2.5.1 Estrutura do Visitor

O visitor principal implementa métodos para cada regra da gramática:

```python
class IimlEvalVisitor(iimlVisitor):
    def __init__(self):
        super().__init__()
        self.vars = {}  # Armazena as variáveis do programa

    def visitProgram(self, ctx:iimlParser.ProgramContext):
        for stmt in ctx.statement():
            self.visit(stmt)
        return self.vars

    def visitVariableDeclaration(self, ctx:iimlParser.
        VariableDeclarationContext):
        var_type = ctx.type_().getText()
        var_name = ctx.ID().getText()
        value = self.visit(ctx.expression())
        self.vars[var_name] = value
        return value

    # Outros métodos do visitor...
```

Listing 2.10: Estrutura do IimlEvalVisitor

2.5.2 Implementação de Manipulação de Imagens

O interpretador IIML implementa operações de imagem usando NumPy para processamento eficiente:

```python
def visitImageStatement(self, ctx:iimlParser.ImageStatementContext):
    width = int(self.visit(ctx.expression(0)))
    height = int(self.visit(ctx.expression(1)))
    background = float(self.visit(ctx.expression(2)))

    # Criar matriz NumPy com as dimensões corretas (altura, largura)
    img = np.full((height, width), background, dtype=float)
    self.vars["image"] = img
    return img
```

Listing 2.11: Método para Criar Imagem

2.5.3 Visualização e Desenho de Formas

Uma parte fundamental da linguagem IIML é a capacidade de desenhar formas geométricas:

```python
def visitPlaceStatement(self, ctx:iimlParser.PlaceStatementContext):
    if "image" not in self.vars:
        raise RuntimeError("No active image to draw on")

    shape = ctx.shape().getText()
    img = self.vars["image"]
    height, width = img.shape  # Note: arrays NumPy são (altura, largura)

    # Inicializar variáveis com valores padrão
    shape_width = 0
    shape_height = 0
    radius = 0
    size_type = "radius"

    # Obter tamanho da forma (raio ou largura/altura)
    if ctx.shapeSize().getChild(0).getText() == "radius":
        radius = int(self.visit(ctx.shapeSize().expression(0)))
        size_type = "radius"
    else:
        shape_width = int(self.visit(ctx.shapeSize().expression(0)))
        shape_height = int(self.visit(ctx.shapeSize().expression(1)))
        radius = max(shape_width, shape_height) // 2
        size_type = "width_height"

    # Posição central e intensidade
    centerX = int(self.visit(ctx.expression(0)))
    centerY = int(self.visit(ctx.expression(1)))
    intensity = float(self.visit(ctx.expression(2)))

    # Desenhar diferentes formas
    if shape == "circle":
        y, x = np.ogrid[:height, :width]
        mask = (x - centerX)**2 + (y - centerY)**2 <= radius**2
        img[mask] = intensity
    elif shape == "rect":
        # Implementação de retângulo...
    elif shape == "cross":
        # Implementação de cruz...
    elif shape == "plus":
        # Implementação de plus...

    return img
```

Listing 2.12: Método para Desenhar Formas

2.5.4 Laços e Estruturas de Controle

A linguagem IIML também suporta estruturas de controle como loops for:

```python
def visitForStatement(self, ctx:iimlParser.ForStatementContext):
    # Obter a coleção iterável
    collection = self.visit(ctx.forControl().expression())

    # Obter o nome da variável de iteração
    var_name = ctx.forControl().ID().getText()

    # Determinar o tipo da variável se especificado
    var_type = None
    if ctx.forControl().type_():
        var_type = ctx.forControl().type_().getText()

    # Armazenar o valor original da variável se existir
    original_value = self.vars.get(var_name, None)

    # Iterar sobre a coleção
    for item in collection:
        # Atribuir o item atual à variável de iteração
        self.vars[var_name] = item

        # Executar o corpo do loop (uma única instrução)
        self.visit(ctx.statement())

    # Restaurar o valor original ou remover se não existia antes
    if original_value is not None:
        self.vars[var_name] = original_value
    else:
        self.vars.pop(var_name, None)
```

Listing 2.13: Implementação do Loop For

2.6 Tratamento de Imagens

2.6.1 Biblioteca de Manipulação de Imagens

Para a implementação do interpretador IIML, utilizamos NumPy e a biblioteca PIL (Python Imaging Library) para representação e manipulação de imagens em tons de cinza:

- **NumPy**: Para representar imagens como arrays 2D e realizar operações matriciais eficientes
- **PIL**: Para carregamento, visualização e salvamento de imagens em formato PGM

2.6.2 Implementação das Operações de Imagem

O interpretador IIML implementa várias operações de imagem:

```python
# Salvar imagem em formato PGM
img_uint8 = np.clip(img_draw * 255, 0, 255).astype(np.uint8)
Image.fromarray(img_uint8).save(output_path)
print(f"Image saved to {output_path}")
```

Listing 2.14: Salvamento de Imagem

Para desenhar formas, implementamos algoritmos eficientes usando operações vetorizadas do NumPy:

```python
y, x = np.ogrid[:height, :width]
mask = (x - centerX)**2 + (y - centerY)**2 <= radius**2
img[mask] = intensity
```

Listing 2.15: Desenho de Círculo Eficiente

2.7 Integração entre IML e IIML

A linguagem IML principal pode executar programas IIML através do operador `run from`:

```
@Override
public String visitRunOp(imlParser.RunOpContext ctx) {
    String iimlCode = visit(ctx.expression());
    return String.format(
        "run_iiml_interpreter(%s)",
        iimlCode
    );
}
```

Listing 2.16: Execução de IIML a partir de IML

O código Python gerado inclui uma função para executar o interpretador IIML:

```python
def run_iiml_interpreter(code):
    from io import StringIO
    from iiml_runtime import interpret_iiml

    # Executar o código IIML e retornar a imagem resultante
    result = interpret_iiml(code)
    return result
```

Listing 2.17: Função de Execução do IIML

3. Conclusões e Trabalhos Futuros

3.1 Resultados Obtidos

Implementamos com sucesso:

- Compilador IML que gera código Python para manipulação de imagens
- Interpretador IIML em Python para criar e manipular imagens
- Suporte para todas as características mínimas requeridas
- Algumas características desejáveis, como estruturas de controle e loops

3.2 Limitações e Desafios

Alguns desafios enfrentados:

- Conversão do interpretador IIML de Java para Python
- Implementação eficiente de operações morfológicas
- Garantir consistência de tipos entre as linguagens
- Compatibilidade entre os diferentes formatos de representação de imagem

3.3 Trabalhos Futuros

Possíveis melhorias futuras:

- Implementar características avançadas (funções, tratamento de erros)
- Melhorar a eficiência das operações morfológicas complexas
- Adicionar suporte para imagens coloridas (RGB)
- Desenvolver uma interface gráfica para visualização e depuração

3.4 Observações Finais

A implementação do compilador para a linguagem IML e do interpretador para IIML constituiu uma experiência enriquecedora para a nossa formação académica e profissional. Este projeto permitiu-nos consolidar conhecimentos teóricos adquiridos ao longo da unidade curricular de Compiladores e aplicá-los num contexto prático.

Ao longo do desenvolvimento deste trabalho, enfrentámos diversos desafios que contribuíram significativamente para o desenvolvimento das nossas competências técnicas. A definição formal de uma linguagem através de gramáticas ANTLR, a implementação de análise semântica e a geração de código são processos complexos que exigiram um profundo entendimento dos conceitos fundamentais de compiladores.

O foco na manipulação de imagens também nos permitiu explorar a aplicação prática de compiladores em domínios específicos, evidenciando como estas podem simplificar tarefas complexas através de abstrações adequadas ao problema.

Em suma, o desenvolvimento do compilador IML e do interpretador IIML proporcionou-nos não apenas conhecimentos técnicos profundos sobre compiladores, mas também uma apreciação mais ampla do papel fundamental que estas ferramentas desempenham no ecossistema da computação moderna.