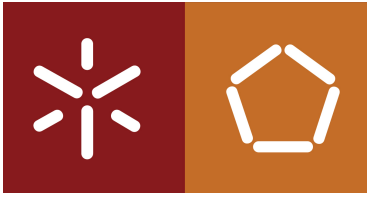

Mestrado em Engenharia Eletrónica Industrial e Computadores

Projeto Integrador em Engenharia Eletrónica Industrial e Computadores

Exploração Prática de um Compilador para Processamento Simplificado de Imagem

Autor:

Tiago Leal Pereira

Professor Orientador:

Adriano Tavares

Junho 2024

Índice

List of Figures

List of Tables

1 Enquadramento e Motivação

Este relatório tem como objetivo analisar o desenvolvimento e a implementação deste projeto, realizado durante o semestre, no âmbito da unidade curricular Projeto Integrador em Eletrónica Industrial e Computadores.

Este projeto oferece uma exploração profunda da construção de compiladores, proporcionando uma compreensão abrangente das várias fases de um compilador, dos seus outputs e das suas interconexões. A DSL que me proponho a desenvolver é concebida para encapsular tarefas fundamentais de processamento de imagens, desde operações básicas como filtragem e redimensionamento até funcionalidades mais avançadas como deteção de contornos ou reconhecimento de texto.

Através da construção de programas personalizados de *lexing* e *parsing*, ganharei experiência prática na decomposição do código fonte da DSL em tokens e na construção de uma árvore sintática abstrata (AST). As etapas subsequentes envolvem a análise semântica para aplicar regras e restrições da DSL, seguida pela geração de código numa linguagem alvo com base na AST validada (árvore sintática colorida). O meu objetivo é produzir um executável que incorpore a representação traduzida da DSL, demonstrando a capacidade da DSL de abstrair as complexidades do processamento de imagens para programadores não familiarizados com os *frameworks* subjacentes.

A motivação para a realização deste trabalho originou-se principalmente da curiosidade em explorar técnicas de processamento de imagem e as bibliotecas disponíveis para esse propósito. Além disso, tive um grande interesse em investigar a área de compiladores e o seu desenvolvimento e programação. Esta oportunidade tornou-se ainda mais relevante, uma vez que estava inscrito na unidade curricular de Linguagens de Sistemas Embebidos onde se aborda o assunto de Compiladores, permitindo uma integração eficaz entre este projeto e os conteúdos académicos e projetos dessa unidade curricular.

2 Objetivos e Resultados Esperados

O trabalho foi subdividido em diferentes objetivos a serem alcançados, de maneira a permitir que o seu acompanhamento seja também mais eficaz. Os principais objetivos delineados foram:

1. Design da *DSL (Domain Specific Language)*[4]. Construir a sintaxe e a sua semântica, especificando os métodos de construção e o tipo de operações que ela é capaz de suportar, sendo sempre o seu objetivo primário criar uma linguagem que simplifique a manipulação e o processamento de imagens.

2. Implementação de um Analisador Léxico(*Scanner*) e um Sintático(*Parser*) para a linguagem anteriormente desenhada.

3. Implementação de um Analisador Semântico de maneira a garantir que o programa cumpra com as regras e restrições semânticas impostas pela prórpria linguagem.

4. Geração de código numa linguagem alvo (*Target Language*) e a consequente produção de um executável.

Espera-se que o resultado final do projeto ofereça uma interface mais intuitiva e acessível, simplificando o processo de manipulação e processamento de imagem. Isso será possível graças à adição de um nível de abstração que permitirá ao programador final, mesmo sem conhecimentos de programação ou das *frameworks* subjacentes, realizar essas modificações desejadas sem ter contacto com as complexidades das bibliotecas de processamento de imagem. Entre as modificações possíveis, incluem-se operações simples, desde redimensionamento, rotação e aplicação de filtros até operações mais complexas, como deteção de contornos e reconhecimento de texto.

2.1 Requisitos

2.1.1 Requisitos funcionais

- A DSL/Metalinguagem deve descrever o fluxo de execução baseado na composição de funções com configurações para deployment sequencial e paralelo
- O compilador deve ser capaz de reconhecer tokens válidos da DSL
- Deve ser capaz de verificar a sintaxe e semântica das frases da DSL
- O compilador deve gerar código executável que permita realizar as transformações na imagem pretendidas

2.1.2 Requisitos não-funcionais

- O projeto deve integrar bibliotecas de processamento de imagem
- Fornecer uma API que facilite o uso de bibliotecas de processamento de imagem através da DSL
- O tempo de compilação deve ser minimizado (tanto do compilador da DSL como do compilador da *Target Language*)

2.2 Restrições

2.2.1 Restrições técnicas

- O software deve ser escrito em C/C++
- Usar a ferramenta gerador-gerador Flex/Lex para gerar o Analisador Léxico
- Usar da ferramenta gerador-gerador Yacc/Bison para gerar o Analisador Sintático

2.2.2 Restrições não-técnicas

- Número de integrantes do projeto, neste caso em específico, uma pessoa
- Tempo limitado para a realização deste projeto, neste caso em específico, um semestre

3 Estado da Arte

O desenvolvimento de DSLs e ferramentas associadas é um campo de extrema importância na engenharia de software. Essas linguagens especializadas permitem a criação de soluções altamente adaptadas e eficientes para diferentes domínios de aplicação, proporcionando uma maneira mais intuitiva e produtiva de expressar problemas e soluções num determinado contexto.

No contexto das metalinguagens e pipelines de funções, destacam-se abordagens poderosas para o desenvolvimento de DSLs. As metalinguagens oferecem uma forma de descrever e definir linguagens específicas de domínio, permitindo aos desenvolvedores criar linguagens com semântica e sintaxe especializadas para resolver problemas específicos. Por outro lado, os pipelines de funções fornecem uma estrutura flexível e modular para compor operações de processamento de dados de forma eficiente e escalável.

Neste contexto, surge a necessidade de explorar a aplicação dessas técnicas no desenvolvimento de uma DSL para processamento de imagem. O processamento de imagem é um domínio altamente especializado, com requisitos e desafios específicos que podem ser melhor abordados através de uma linguagem de programação dedicada.

Ao desenvolver uma DSL para processamento de imagem, é essencial selecionar as ferramentas certas para o trabalho. Isso pode incluir o uso de geradores de compiladores como Flex/Lex e Yacc/Bison para implementar a análise léxica e sintática da linguagem, bem como frameworks como OpenCV para fornecer funcionalidades de processamento de imagem de alto desempenho.

Ao adotar uma abordagem baseada em metalinguagens e pipelines de funções, é possível obter uma série de benefícios significativos. Essas abordagens oferecem uma maneira intuitiva de descrever operações de processamento de imagem, facilitando a expressão de algoritmos complexos de forma concisa e legível. Além disso, os pipelines de funções proporcionam uma maneira modular e reutilizável de compor operações de processamento de imagem, permitindo a construção de fluxos de trabalho flexíveis e escaláveis.

No contexto do desenvolvimento de uma DSL para processamento de imagem, explorar e compreender as tecnologias e práticas existentes é fundamental para garantir o sucesso do projeto. Ao selecionar as ferramentas certas e adotar uma abordagem baseada em metalinguagens e pipelines de funções, é possível criar uma DSL poderosa e eficiente que atenda às necessidades específicas do domínio de processamento de imagem, proporcionando aos desenvolvedores uma maneira mais intuitiva e produtiva de lidar com problemas nesse contexto especializado.

4 Análise do Problema

Ao iniciar o desenvolvimento de uma DSL para processamento de imagem, é crucial realizar uma análise aprofundada do problema em questão. O processamento de imagem é um domínio vasto e complexo, repleto de desafios e requisitos específicos que devem ser cuidadosamente considerados ao projetar uma solução adequada.

Nesta fase inicial, é essencial compreender as necessidades e objetivos dos programadores finais da DSL. Isso envolve identificar os tipos de tarefas de processamento de imagem que serão realizadas, as restrições de desempenho e eficiência que devem ser atendidas e os fluxos de trabalho típicos que serão executados. Além disso, é importante considerar o nível de abstração desejado pelos utilizadores e a complexidade das operações de processamento de imagem que serão suportadas pela DSL.

Ao analisar o problema, também é crucial considerar as tecnologias e práticas existentes no campo do processamento de imagem. Isso inclui a avaliação de bibliotecas e frameworks populares, como OpenCV, e a compreensão das técnicas e algoritmos de processamento de imagem mais utilizados na indústria. Essa análise ajuda a identificar padrões e melhores práticas que podem ser incorporados à DSL, garantindo que ela seja robusta, eficiente e compatível com as expectativas dos utilizadores.

Além disso, ao considerar a implementação da DSL, é importante avaliar as diferentes abordagens disponíveis, como o uso de metalinguagens e pipelines de funções. As metalinguagens oferecem uma maneira poderosa de definir a sintaxe e semântica da linguagem, permitindo uma expressão precisa e concisa das operações de processamento de imagem. Por outro lado, os pipelines de funções fornecem uma estrutura flexível e modular para compor operações de processamento de imagem de forma eficiente e escalável.

4.1 Metalinguagens

No contexto da computação, uma metalinguagem é definida como sendo uma linguagem utilizada para descrever outra linguagem, permitindo assim aos programadores as definirem e manipularem. Por exemplo, é tradicionalmente utilizado metalinguagens como Backus-Naur-Form(BNF) [1] para definir formalmente a sintaxe e a gramática duma nova linguagem, criando as ferramentas necessárias para ser possível trabalhar com o código escrito nela.

Uma metalinguagem não necessita necessariamente de ser uma linguagem de programação, ou seja não precisa de ser *Turing Complete*, inclusive a BNF é um exemplo disso mesmo. No entanto, existem outros exemplos que podem ser considerados *Turing Complete*, como é o caso do *Lisp*, metalin-

guagem amplamente utilizada e que influenciou outras linguagens tão conhecidos, como o Python, Ruby ou Haskell.

Neste projeto, a metalinguagem utilizada é composta por dois componentes. Como ferramenta para definir os *tokens* da DSL foi utilizado o gerador-gerador *Flex/Lex*, já no que toca à gramática da minha DSL, foi utilizado o gerador-gerador *Yacc/Bison*, que permitiu descrever a sintaxe da minha linguagem formal usando uma notação baseada em BNF. Após a construção da AST ainda no ficheiro *yacc*, foi implementada uma função em C++ para fazer a travessia dessa mesma árvore e, finalmente, gerar o código OpenCV C++ correspondente. Esta utilização da linguagem C++ pode ser vista como uma forma de meta-programação, pois é utilizado para manipular estruturas definidas por outra linguagem (neste caso, a minha DSL).

A escolha de utilizar as ferramentas Flex e Bison partiu de diversos fatores, nomeadamente, a sua vasta utilização em diferentes setores. São amplamente utilizadas, por exemplo, em sistemas de compiladores, sendo parte essencial do GNU Compiler Collection (GCC), entre outros projetos de grande relevância. No entanto, houve um outro conjunto de vantagens que, nesta situação, pesaram ainda mais na decisão de escolher estas ferramentas ao invés de outras como o ANTLR, nomeadamente:

- Flex e Bison são ferramentas bem estabelecidas e amplamente documentadas, o que facilita a sua integração em projetos complexos e a resolução de problemas.
- Bison utiliza uma abordagem LALR(*Look-Ahead LR*), o que permite a definição de gramáticas mais flexíveis, incluindo gramáticas com *left-recursion*, aumentando a expressividade e simplicidade na definição de regras gramaticais.

As ferramentas Flex e Bison permitem a construção de Analisadores Léxicos e Sintáticos em C. Integrar o Flex e o Bison no projeto, exigiu a instalação e configuração adequada das ferramentas, e a inclusão dos ficheiros gerados (.l e .y) no processo de compilação. Estas dependências foram configuradas adequadamente através da criação de um ficheiro *makefile* que coordenou opções de compilação, caminhos de inclusão e ligação às bibliotecas de tempo de execução necessárias. Por este motivo, a escolha do C/C++ para a implementação facilitou o desenvolvimento, apesar dos desafios iniciais na configuração do ambiente de compilação.

4.2 Pipeline de Funções

O pipeline de funções na DSL foi desenhado para seguir uma arquitetura modular, onde cada função representa uma operação específica a ser realizada na imagem. A sequência de operações é encadeada

de forma que a saída de uma função serve como entrada para a próxima, permitindo a criação de fluxos de processamento complexos a partir de operações básicas.

As funções disponíveis na DSL abrangem uma ampla gama de operações de processamento de imagem, incluindo, mas não se limitando a:

- **Filtros de imagem:** Aplicação de filtros como blur, sharpen, edge detection, etc.
- **Transformações geométricas:** Redimensionamento, rotação, translação, etc.
- **Ajustes de cores:** Modificação de brilho, contraste, saturação, etc.

Cada função é definida de forma a receber uma imagem como entrada e retornar uma imagem como saída, mantendo a consistência e previsibilidade do *pipeline*.

A construção do *pipeline* é realizada através de uma sintaxe intuitiva que permite ao programador especificar a sequência de operações de maneira declarativa. Por exemplo, um *pipeline* que aplica uma binarização seguida de um espelhamento terminando com um redimensionamento poderia ser definido como:

```
1 load(imagem1, "Images/ESRG.png")
2 imagem1 >> binarization() >> flip() >> resize(800,600)
```

Neste exemplo, a função *load* carrega a imagem inicial, que é então passada sequencialmente pelas funções *binarization*, *flip* e por fim *resize*.

A utilização de um pipeline de funções na DSL de processamento de imagem oferece vários benefícios:

- **Modularidade:** Cada função é isolada, facilitando a manutenção e atualização de operações específicas sem afetar o restante pipeline.
- **Flexibilidade:** A capacidade de encadear funções permite a criação de fluxos de processamento personalizados para atender a diferentes necessidades e cenários de uso.
- **Facilidade de Uso:** A sintaxe declarativa e intuitiva da DSL facilita o desenvolvimento de pipelines complexos sem exigir conhecimentos profundos de programação.

5 Design da solução

Na presente secção, explorarei os principais aspetos do design do compilador, começando pelo design da DSL passando por todas as fases de compilação tanto do *frontend* como do *backend*. Abordarei as técnicas e ferramentas utilizadas no desenvolvimento do compilador, desde a seleção de algoritmos de análise léxica e sintática até à implementação de otimizações de código e geração de código.

5.1 Design da DSL

O design da DSL foi cuidadosamente elaborado para proporcionar uma interface intuitiva e eficiente, adotando um paradigma funcional que facilita a expressão de operações comuns de manipulação e análise de imagens. Esta linguagem especializada visa simplificar a construção de algoritmos de processamento de imagem, tornando o desenvolvimento e a manutenção do código mais acessíveis. Abaixo, detalho algumas das operações suportadas pela DSL e as suas funcionalidades:

- **Operações Básicas de Manipulação de Imagem**:
  - **`imagem2 = binarization(imagem1):`** Realiza a binarização de `imagem1`, convertendo-a numa imagem binária e guarda-a em `imagem2`.
  - **`imagem2 = flip(imagem1):`** Inverte `imagem1` verticalmente e guarda-a em `imagem2`.
  - **`imagem2 = resize(1600,1080,imagem1):`** Redimensiona `imagem1` para as dimensões especificadas (1600x1080) e guarda-a em `imagem2`.
  - **`show(imagem1):`** Exibe `imagem1`.
- **Reconhecimento de Texto**:
  - **`textRecognition(textImg, myVar):`** Executa o reconhecimento de texto em `textImg` e a *string* em `myVar`.
  - **`print(myVar):`** Imprime o conteúdo de `myVar`, mostrando o texto reconhecido.
- **Detecção de contornos e Desfocagem:**
  - **`imagem2 = countor(imagem1):`** Detecta contornos em `image1` e guarda-a em `imagem2`.
  - **`imagem2 = gaussianBlur(3,4,imagem1):`** Aplica um desfoque Gaussiano em `imagem1` com os parâmetros especificados (Kernel Size = 3x3 e Sigma = 4) e guarda-a em `imagem2`.

- **Operações de Limiar Binário**:
  - **`imagem2 = binaryThreshold(127,imagem1):`** Aplica um limiar binário a `imagem1` com o valor de limiar 127 e guarda-a em `imagem2`.
  - **`imagem2 = binaryInvThreshold(7,imagem1):`** Aplica um limiar binário invertido a `imagem1` com o valor de limiar 7 e guarda-a em `imagem2`.
  - **`imagem2 = otsuThreshold(8,imagem1):`** Aplica a técnica de limiarização de Otsu a `imagem1` e guarda-a em `imagem2`.
- **Operações Aritméticas entre Imagens**:
  - **`imagem3 = imagem1 + imagem2:`** Soma `imagem1` e `imagem2` e guarda-a em `imagem3`.
  - **`imagem3 = imagem1 - imagem2:`** Subtrai `imagem2` de `imagem1` e guarda-a em `imagem3`.
  - **`imagem3 = imagem1 * imagem2:`** Multiplica `imagem1` por `imagem2` e guarda-a em `imagem3`.
- **Manipulação de Arrays de Imagens**:
  - **`_my_array = [imagem1, imagem2, textImg]:`** Cria um array que contém `imagem1`, `imagem2` e `textImg`.
  - **`loop(_my_array, flip()):`** Aplica uma operação (neste caso de *flip*) a todas as imagens do array `_my_array`.
  - **`testing_index = _my_array !! 2:`** Acede o terceiro elemento do array `_my_array` e guarda-a em `testing_index`.
- **Pipelines de Operações**:
  - **`compile >> imagem1 >> binarization() >> flip() >> gaussianBlur(3,2) >> resize(640,400):`** Define uma sequência de operações encadeadas a serem aplicadas a `imagem1`.
- **Macros de Modo de Execução**:
  - `SET_MODE(SEQUENTIAL)`: Define o modo de execução sequencial.
  - `SET_MODE(PARALLEL)`: Define o modo de execução paralelo.

Ao adotar um paradigma funcional, esta DSL permite que as operações sejam tratadas como funções, promovendo uma maior modularidade e facilidade de composição. As funções podem ser encadeadas de forma clara e natural, permitindo a construção de pipelines de processamento de imagem de maneira concisa e legível. Essa abordagem funcional também facilita a paralelização e otimização das operações, melhorando a eficiência do processamento.

Este design da DSL foi concebido para ser altamente expressivo e fácil de usar, permitindo aos utilizadores finais especificar de maneira clara e concisa as operações de processamento de imagem que desejam realizar. A DSL não só facilita a definição de algoritmos complexos, como também promove a reutilização e a composição de operações, aumentando a eficiência e a produtividade no desenvolvimento de aplicações de processamento de imagem.

5.2 Frontend

5.2.1 Análise Léxica

A análise léxica é uma fase crucial no processo de compilação, onde um programa fonte é transformado numa sequência de tokens. Esses tokens representam os elementos básicos da linguagem de programação, como palavras-chave, identificadores, números e símbolos especiais. A análise léxica é responsável por reconhecer e identificar esses tokens, ignorando espaços em branco e comentários.

Para implementar a análise léxica, foi utilizada a ferramenta *Flex* (também conhecido como *Lex*), um gerador de analisadores léxicos. O *Flex* simplifica o processo de desenvolvimento do analisador léxico, permitindo que os desenvolvedores descrevam padrões de expressões regulares para identificar os tokens da linguagem. Com a definição desses padrões, o *Flex* gera automaticamente um programa em C (ou C++) que pode analisar o código-fonte e produzir os tokens correspondentes.

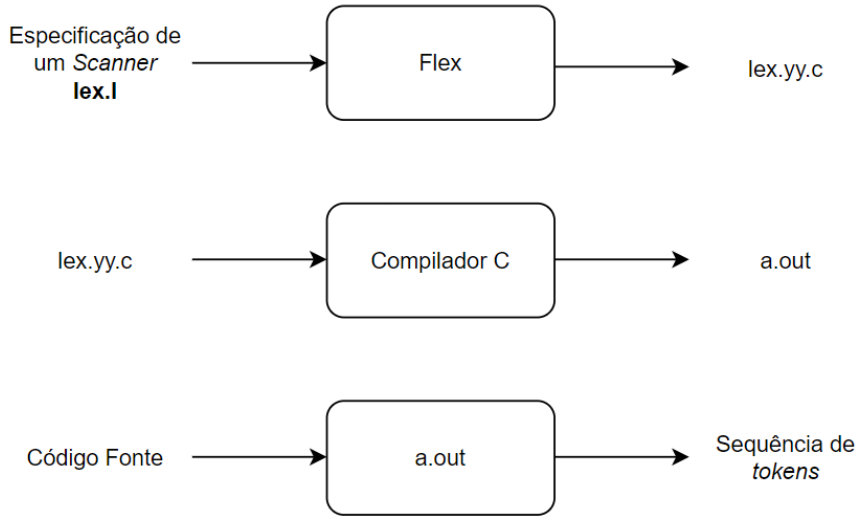

Figure 5.1: Gerador-gerador Flex/Lex

O formato de um ficheiro *Flex* é o seguinte:

```
%{

// Variáveis globais e protótipos de funções

%}

// Secção das definições

%%

// Secção das regras de produção

%%

// Sub-rotinas C auxiliares
```

As expressões regulares são padrões usados para corresponder combinações de caracteres em cadeias de texto. São definidos com metacaracteres para identificar tipos de *tokens*, como identificadores, palavras-chave, operadores e delimitadores, facilitando a análise léxica. Abaixo encontram-se as expressões regulares definidas para a DSL.

Token	**Expressão Regular**
number	`[0-9]`
integer	`{number}+`
array	`_[a-zA-Z0-9_]+`
identifier	`[a-z_][a-zA-Z0-9_]*`
comment	`{[ ^{}]*}`

Table 5.1: Expressões Regulares

5.2.2 Análise Sintática

A análise sintática é a segunda fase fundamental no processo de compilação, após a análise léxica. Enquanto a análise léxica lida com a identificação de tokens individuais, a análise sintática encarrega-se da estrutura gramatical do código-fonte, verificando se a sequência de tokens formam frases válidas de acordo com a gramática da linguagem de programação.

Para implementar a análise sintática, foi utilizada a ferramenta *Yacc* (também conhecido como *Bison*), um gerador de analisadores sintáticos. A ferramenta *Yacc* permite aos desenvolvedores especificar a gramática da linguagem de programação usando uma notação semelhante à *Backus-Naur Form* (BNF), descrevendo as regras de produção que definem como os diferentes elementos da linguagem podem ser combinados.

Com base nessas especificações, *Yacc* gera um analisador sintático em C (ou C++) que pode ler a sequência de tokens produzida pela análise léxica e verificar se a mesma segue as regras gramaticais estabelecidas. Se a sequência de tokens não estiver em conformidade com a gramática, o analisador sintático gera mensagens de erro para indicar onde ocorreu o problema na estrutura do código-fonte.

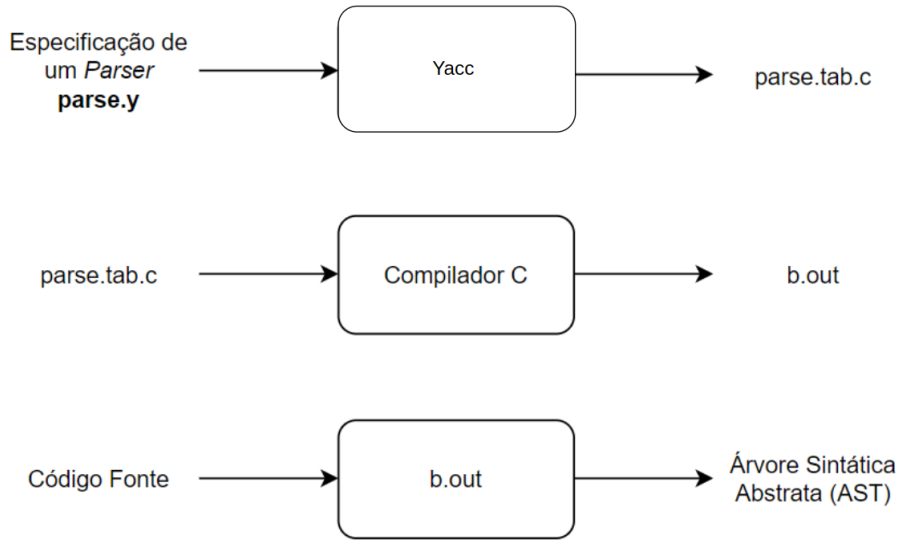

Figure 5.2: Gerador-gerador Yacc/Bison

O formato de um ficheiro *Yacc* é muito semelhante ao de um ficheiro *Flex* e tem a seguinte forma:

```
%{

// Variáveis globais e protótipos de funções

%}

// Secção das definições das expressões regulares

%%

// Secção das regras

%%

// Rotinas C auxiliares
```

As regras de produção são um aspecto fundamental da análise sintática no design de compiladores, definindo a estrutura gramatical de uma linguagem de programação. Expressas em notação formal como BNF, delineiam como as construções sintáticas são formadas a partir de elementos básicos. Essas regras estabelecem uma hierarquia na estrutura do código-fonte, possibilitando uma análise sintática precisa e eficiente. As regras gramaticais especificam como variáveis sintáticas podem ser expandidas para formar frases.

A AST é usada para representar a estrutura sintática do código-fonte de forma hierárquica e abstrata. A estrutura *Node* representa os nós na AST. Inclui uma *string* associada ao nome da operação a ser executada, um inteiro que armazena o valor associado ao nó, um array de ponteiros para os nós filhos e, por fim, outro inteiro que armazena o número de filhos do nó atual.

Campo	Descrição
`label`	rótulo do nó
`value`	valor associado ao nó
`children`	vetor de apontadores para os nós filhos
`num_children`	número de nós filhos

Table 5.2: Definição da estrutura Node

São apresentadas a seguir algumas subárvores da AST, com o objetivo de demonstrar os nós filhos criados sempre que uma operação é chamada.

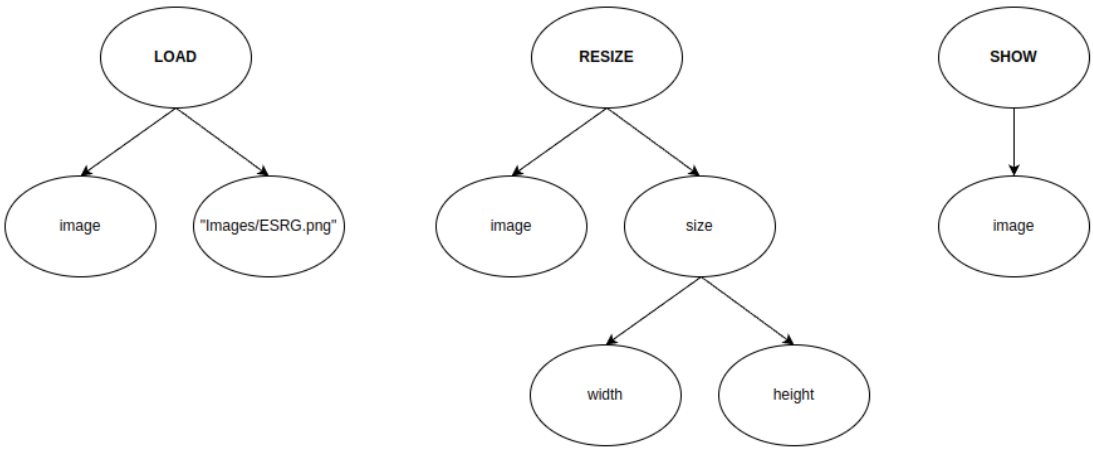

Figure 5.3: Sub-árvores das operações *load*, *resize* e *show*

Como é possível visualizar na figura acima, quando a operação *load* é chamada no código fonte da DSL, o analisador sintático cria o nó `LOAD` com dois nós filhos: um contendo o identificador onde a imagem carregada será guardada e outro contendo o caminho da imagem a ser carregada. No caso da operação `RESIZE`, são criados dois nós filhos: um com o identificador onde a imagem redimensionada será guardada e outro com o tamanho a ser redimensionado, que, por sua vez, cria dois filhos para armazenar os valores de altura e largura. Por fim, no caso da operação `SHOW`, apenas um nó filho é criado, que será a imagem a ser exibida.

5.2.3 Análise Semântica

A análise semântica é uma etapa fundamental no design do compilador para a DSL. Enquanto a análise sintática garante que o código-fonte esteja estruturado corretamente de acordo com a gramática da linguagem, a análise semântica assegura que o código faça sentido e cumpra as regras semânticas da DSL.

No contexto do compilador para a DSL de processamento de imagem, a análise semântica realiza várias verificações cruciais:

- **Verificação de Carregamento de Imagem:** Antes de qualquer operação ser realizada numa imagem, é necessário garantir que a imagem foi previamente carregada. O analisador semântico verifica se cada imagem que recebe uma operação foi efetivamente carregada, evitando operações inválidas em imagens inexistentes.

- **Type Checking:** A verificação de tipos é essencial para assegurar que as operações de processamento de imagem recebem os tipos de dados corretos. O analisador verifica que os argumentos passados para cada operação são do tipo esperado, prevenindo erros de tipo que poderiam causar falhas na execução.

- **Verificação do Número de Argumentos:** Cada operação de processamento de imagem exige um número específico de argumentos. O analisador semântico verifica se o número de argumentos fornecidos em cada chamada de operação está correto, emitindo erros caso haja argumentos em falta ou em excesso.

- **Verificação de Índices:** A DSL permite o uso de arrays de imagens, e é importante garantir que os índices utilizados para aceder a estas imagens estejam dentro da gama inicializada. O analisador semântico verifica que os índices usados estão dentro dos limites válidos dos arrays de imagens, evitando acessos fora dos limites que poderiam resultar em comportamentos inesperados ou erros de execução.

Os erros semânticos detetados durante esta fase são reportados ao programador com mensagens detalhadas, indicando a natureza e a localização dos problemas no código-fonte. Isto não só ajuda a garantir a precisão do código, mas também facilita a identificação e correção de erros por parte do programador. Para esse efeito será feita a travessia da árvore sintática em *post-order*, ou seja, o nó pai é processado depois de todos os seus filhos terem sido verificados.

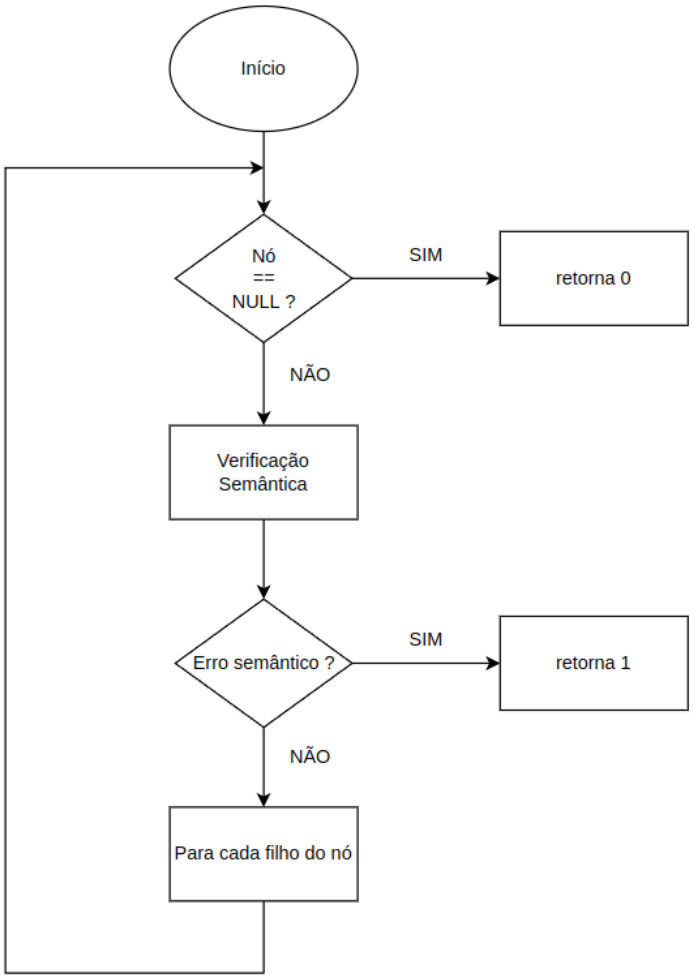

Figure 5.4: Fluxograma de alto nível da travessia da AST

A travessia em *post-order* revela-se mais eficiente em comparação a outros tipos de travessia para a análise semântica, uma vez que os nós pais da AST representam as operações de processamento de imagem, enquanto os respetivos filhos representam os parâmetros dessas operações, incluindo as imagens às quais estas operações serão aplicadas.

5.3 Backend

5.3.1 Geração de Código

A geração de código é a etapa final no processo de compilação, onde o código-fonte analisado e validado é transformado em código executável. No contexto do compilador para a DSL de processamento de imagem, a geração de código foi projetada para ser híbrida. Isto significa que o compilador pode gerar código C++ utilizando a biblioteca OpenCV para execução sequencial, ou pode gerar código C++ utilizando OpenCV em conjunto com PThreads [3] para execução em paralelo.

Na configuração sequencial, o compilador traduz as operações definidas na DSL diretamente em chamadas de funções da biblioteca OpenCV. Cada operação de processamento de imagem especificada pelo programador na DSL é mapeada para uma função correspondente em OpenCV, garantindo que as operações sejam realizadas numa ordem sequencial, tal como especificado no código-fonte. Este modo é ideal para tarefas que não requerem processamento paralelo ou quando a simplicidade de execução sequencial é preferível.

No que toca à configuração paralela, para tirar proveito dos sistemas multiprocessadores modernos, o compilador pode também gerar código paralelo utilizando PThreads em conjunto com OpenCV. Neste modo, o compilador identifica as operações de processamento de imagem que podem ser executadas em paralelo e gera código C++ que utiliza PThreads para distribuir estas operações por múltiplos threads. Cada thread pode executar uma operação de imagem independentemente, permitindo um processamento mais rápido e eficiente, especialmente para operações que podem ser realizadas simultaneamente sem dependências entre si.

O processo de geração de código no nosso compilador envolve várias etapas:

- **Tradução de Operações:** Cada operação de processamento de imagem na DSL é traduzida para a função correspondente em OpenCV, gerando código C++ que realiza a operação desejada.
- **Gestão de Fluxo de Controlo:** O compilador gera estruturas de controlo adequadas em C++ para garantir que as operações são executadas na ordem correta.
- **Configuração de PThreads:** Para a geração de código paralelo, o compilador configura PThreads, criando e gerindo as *threads* conforme necessário. Cada thread é atribuído a uma operação de imagem, e o compilador garante a sincronização adequada entre *threads* para evitar *race conditions* e garantir a integridade dos dados.

- **Integração com OpenCV:** Tanto no modo sequencial como no paralelo, o compilador assegura que todas as operações utilizam as funções e estruturas de dados da biblioteca OpenCV, permitindo um processamento eficiente e robusto de imagens.

A abordagem híbrida oferece flexibilidade ao programador, permitindo-lhe escolher o modo de execução mais adequado às suas necessidades. A execução sequencial é simples e direta, enquanto a execução paralela oferece ganhos significativos de desempenho para tarefas intensivas em processamento.

A seguir encontra-se um fluxograma que representa a travessia da AST para a geração de código.

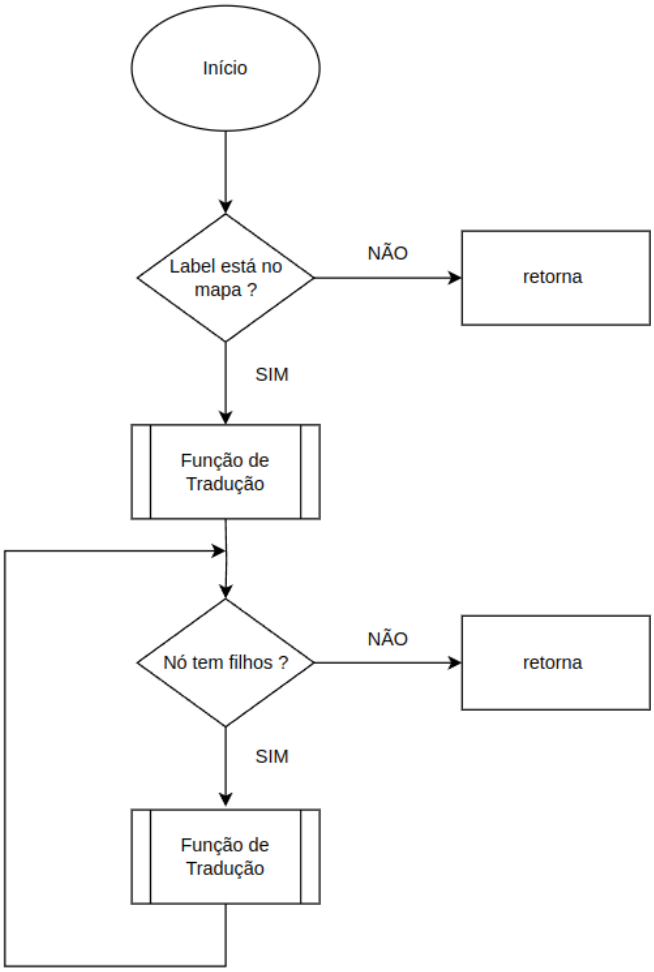

Figure 5.5: Fluxograma da tradução dos nós da AST

5.3.2 Compilação do Código Gerado para Executável

Após a geração do código para a DSL de processamento de imagem, segue-se uma última etapa, a compilação desse código para criar um executável funcional. Nesta etapa é criado um arquivo temporário para armazenar o código gerado, é compilado utilizando o compilador GCC e produz o executável correspondente. Esta etapa automatiza o processo de compilação do código gerado, tornando o desenvolvimento e a implementação de DSLs personalizadas mais eficientes e acessíveis para os utilizadores finais.

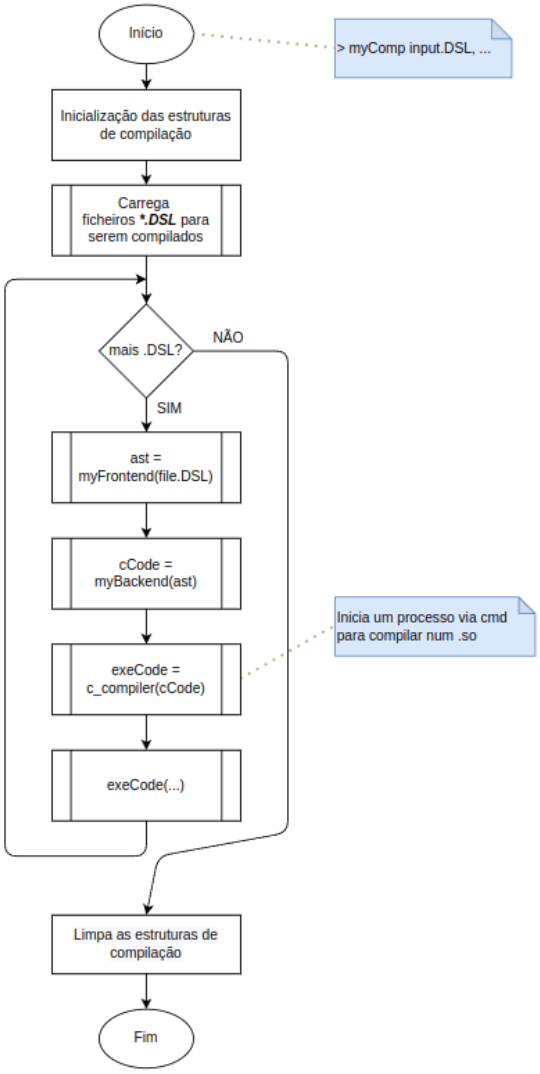

Figure 5.6: Fluxo de Execução do Compilador

6 Implementação da solução

Nesta secção, analisarei os principais aspetos da implementação do compilador, percorrendo todas as fases da compilação, desde o frontend até ao backend. Tratarei das ferramentas utilizadas no desenvolvimento do compilador, da seleção de algoritmos e do código implementado.

6.1 Frontend

6.1.1 Análise Léxica

Na seção de variáveis e protótipos de funções, foi definida a variável *line_number*, inicializada com o valor inteiro "1", que será incrementada sempre que o token "\n" for encontrado, o que representa uma mudança de linha.

```
%{
#include "../Frontend/Scanner/scanner.h"
int line_number=1;
%}
```

Como foi dito na secção de *design* foram definidas as expressões regulares, como se pode verificar abaixo, de salientar que para não se implementar uma versão para rotina `yywrap()` foi usada a opção `noyywrap`.

```
%option noyywrap

number        [0-9]
integer       {number}+
array         _[a-zA-Z0-9_]+
identifier    [a-z_][a-zA-Z0-9_]*
path          \"(\\.|[^\\"])*\"
whitespace    [ \t]+
comment       \{[^{}]*\}
```

Na próxima secção foram definidas as regras que correspondem aos tokens que representam os elementos básicos da DSL, palavras-chave, identificadores, números, etc.

```
%%

"resize"                { return RESIZE; }
"rotate"                { return ROTATE; }
"flip"                  { return FLIP; }
"load"                  { return LOAD; }
"show"                  { return SHOW; }

"gaussianBlur"          { return GAUSSIAN_BLUR; }
"bilateralBlur"         { return BILATERAL_BLUR; }
"medianBlur"            { return MEDIAN_BLUR; }
"binaryThreshold"       { return BINARY_THRESHOLD; }
"binaryInvThreshold"    { return BINARY_INV_THRESHOLD; }
"otsuThreshold"         { return OTSU_THRESHOLD; }
"binarization"          { return BINARIZATION; }
"countor"               { return COUNTOR; }

"textRecognition"       { return RECOGNIZE_TEXT; }
"print"                 { return PRINT; }

"loop"                  { return LOOP; }
"compile"               { return COMPILE; }

"timer"                 { return TIMER; }
"on"                    { return ON; }
"off"                   { return OFF; }

"SET_MODE"              { return SET_MODE; }
"SEQUENTIAL"            { return SEQUENTIAL_MODE; }
"PARALLEL"              { return PARALLEL_MODE; }
"MAP_REDUCE"            { return PARALLEL_CHUNKED_MODE; }

"="                     { return ASSIGNMENT; }
"+"                     { return PLUS; }
```

```
35 "-"                          { return MINUS; }
36 "*"                          { return TIMES; }
37 ","                          { return COMMA; }
38 "("                          { return LPAREN; }
39 ")"                          { return RPAREN; }
40 "["                          { return LBRACKET; }
41 "]"                          { return RBRACKET; }
42 "!!"                         { return DOUBLE_EXCLAMATION; }
43 ">>"                         { return DOUBLE_GREATER; }
44
45 {array}                      { yylval.stringVal = strdup(yytext); return ARRAY; }
46 {identifier}                 { yylval.stringVal = strdup(yytext); return
   ↪  IDENTIFIER; }
47 {integer}                    { yylval.intVal = atoi(yytext); return INT; }
48 {path}                       { yylval.stringVal = strdup(yytext); return PATH; }
49 {whitespace}                 ; /* skip whitespace */
50 {comment}                    ; // ignore comments enclosed in curly braces
51 \n                           { line_number++; /*printf("\n");*/ }
52 <<EOF>>                      { return YY_NULL; }
53 .                            { fprintf(stderr, "Error: [%s] is an invalid
   ↪  token\n", yytext); return ERROR; }
54
55 %%
```

Os `yyval` neste código são usados para armazenar os valores associados aos tokens reconhecidos pelo analisador léxico, antes de passá-los para o analisador sintático. Estes valores são atribuídos aos tokens para que o analisador sintático possa utilizar as informações extraídas do código-fonte durante o processo de análise sintática.

Já na secção das rotinas auxiliares foram definidas duas funções, `int getToken(FILE *inputFile)` que lê um *token* do ficheiro de entrada *inputFile* e `int scanner(FILE *inputFile)` que implementa o ciclo principal do analisador léxico. É responsável por ler *tokens* do ficheiro de entrada um por um até o fim do ficheiro.

```
1  int getToken(FILE *inputFile)
2  {
3    static int firstTime = TRUE;
4    int currentToken;
5    if (firstTime)
6    {
7      firstTime = FALSE;
8      if (!inputFile)
9      {
10       fprintf(stderr, "Error: Input file not opened.\n");
11       return YY_NULL;
12     }
13     yyin = inputFile;
14   }
15   currentToken = yylex();
16   return currentToken;
17 }
18
19
20 int scanner(FILE *inputFile)
21 {
22   int currentToken;
23   while (1)
24   {
25     currentToken = getToken(inputFile);
26     if (currentToken == YY_NULL)
27       break;
28   }
29   return 0;
30 }
```

6.1.2 Análise Sintática

A estrutura Node representa os nós na AST. Ela inclui uma *string* associada ao nome da operação a ser executada, um inteiro que armazena o valor associado ao nó, um array de ponteiros para os nós filhos e, por fim, outro inteiro que armazena o número de filhos do nó atual.

```
typedef struct Node
{
    char *label;
    int value;
    struct Node **children;
    int num_children;
} Node;
```

A função `createNode` é responsável por criar e inicializar um novo nó na árvore sintática. Inicialmente, reserva memória para um novo nó e verifica se a alocação foi bem-sucedida. Em seguida, duplica a *string* do rótulo e atribui ao campo *label* do nó, define o número de filhos e o valor associado ao nó. Além disso, a função aloca memória para o array de apontadores para os filhos e verifica se essa alocação foi bem-sucedida. Caso contrário, libera a memória alocada para o nó e retorna `NULL`. Se tudo ocorrer como esperado, a função retorna o nó criado.

Por outro lado, a função *addChild* é responsável por adicionar um nó filho a um nó pai na estrutura da árvore sintática. Atribui simplesmente o nó filho ao índice especificado no array de filhos do nó pai.

Estas são as funções essenciais para construir e manipular a árvore sintática abstrata durante o processo de análise sintática do compilador da DSL.

```
Node* createNode(const char* label, int num_children, int value)
{
    Node *node = (Node*)malloc(sizeof(Node));
    if(node != NULL)
    {
        node->label = strdup(label);
        node->num_children = num_children;
        node->value = value;
        node->children = (Node**)malloc(num_children * sizeof(Node*));
```

```
        if (node->children == NULL)
        {
            free(node);
            return NULL;
        }
        return node;
    }
    return NULL;
}

void addChild(Node *parent, Node *child, int index)
{
    parent->children[index] = child;
}
```

Na seção de variáveis e protótipos de funções do ficheiro *Yacc*, foi inicializado um apontador para a estrutura `Node`, que representa o nó raiz da AST.

```
%{
#include "../Frontend/Parser/parser.h"
struct Node * root;
%}
```

O seguinte código define os *tokens* e os tipos de dados que serão utilizados pelo parser gerado pelo *Bison*. Especifica as unidades léxicas que o analisador léxico reconhecerá e como esses *tokens* devem ser tratados pelo analisador sintático em termos de valores semânticos.

A diretiva *%union* define um tipo de união que agrupa diferentes tipos de valores que os *tokens* e símbolos não-terminais podem ter. No caso, `stringVal` um apontador para uma *string (char *)*, usado para valores de identificadores e *path's*, `intVal` um valor inteiro (*int*), usado para valores inteiros e por fim `node` um apontador para a estrutura Node, que representa nós na AST.

Através das diretivas *%type*, é possível especificar quais tipos da união são associados a quais *tokens* ou símbolos não-terminais.

Este é um passo crucial na construção de um compilador, pois permite que o parser interprete corretamente a estrutura do código-fonte e construa a AST necessária para a análise semântica e geração

de código subsequentes.

```
%token IDENTIFIER
%token PATH
%token INT
%token ARRAY

...

%token SET_MODE
%token SEQUENTIAL_MODE
%token PARALLEL_MODE

%union
{
    char *stringVal;
    int intVal;
    struct Node * node;
}


%type <stringVal> IDENTIFIER PATH ARRAY
%type <intVal> INT
%type <node> command command_list  identifier_list expression expression_list
↪  func nested_func func_loop func_list mode on_off
```

A seguir encontram-se as regras de produção definidas para a gramática da DSL:

- *program* é a primeira regra e define um `program` como uma *command_list*. Quando o analisador sintático encontra um *program*, interpreta isso como uma lista de comandos, onde a ação *{ root = $1; }* associa o valor resultante da análise da *command_list* à variável *root*, que representa a raiz da árvore de sintaxe abstrata do programa.

command_list pode ser de interpretado de duas formas:

- *command_list : command*

  Esta regra indica que uma *command_list* pode consistir num único *command*, onde a ação

associada cria um novo nó na AST com o rótulo "*Command*" e adiciona o comando como seu filho.

- *command_list : command_list command*

  Esta regra indica que uma *command_list* pode ser uma sequência de uma *command_list* existente seguida por um *command*, onde a ação associada cria um novo nó na AST com o rótulo "*CommandList*" e adiciona a *command_list* existente e o novo *command* como filhos desse nó.

```
%%

program : command_list { root = $1; };

command_list : command
             {
                $$ = createNode("Command", 1, 0);
                addChild($$, $1, 0);
             }
             | command_list command
             {
                 $$ = createNode("CommandList", 2, 0);
                 addChild($$, $1, 0);
                 addChild($$, $2, 1);
             }
             ;

```

A seguir encontram-se as regras de produção que definem o que é um *command*. Cada regra define diferentes operações que podem ser realizadas na linguagem. A estrutura das regras de produção está descrita abaixo:

```
command : SHOW LPAREN IDENTIFIER RPAREN
        {
              $$ = createNode("Show", 1, 0);
              addChild($$, createNode($3, 0, 0), 0);
```

```
5         }
6         | LOAD LPAREN IDENTIFIER COMMA PATH RPAREN
7         {
8               $$ = createNode("Load", 2, 0);
9               addChild($$, createNode($3, 0, 0), 0);
10              addChild($$, createNode($5, 0, 0), 1);
11        }
12        | IDENTIFIER ASSIGNMENT expression
13        {
14              $$ = createNode("Assignment", 2, 0);
15              addChild($$, createNode($1, 0, 0), 0);
16              addChild($$, $3, 1);
17        }
18
19                                  ...
20
21        | COMPILE DOUBLE_GREATER IDENTIFIER DOUBLE_GREATER func_list
22        {
23            $$ = createNode("Compile", 2, 0);
24            addChild($$, createNode($3, 0, 0), 0);
25            addChild($$, $5, 1);
26        }
27        ;
```

Como a linguagem segue o paradigma de operações não destrutivas, em que as operações são aplicadas em cópias das imagens originais, preservando-as intactas, as frases da gramática que correspondem à manipulação simples de imagens seguem a seguinte sintaxe: `imagem1 = binarizacao(imagem2)`. Para tal, como mencionado anteriormente, foi implementada a regra de produção `IDENTIFIER ASSIGNMENT expression`, onde o símbolo não terminal `expression` está definido no código abaixo:

```
expression : RESIZE LPAREN INT COMMA INT COMMA IDENTIFIER RPAREN
        {
            $$ = createNode("Resize", 3, 0);
            addChild($$, createNode("Width", 0, $3), 0);
            addChild($$, createNode("Height", 0, $5), 1);
            addChild($$, createNode($7, 0, 0), 2);
        }
        | ROTATE LPAREN INT COMMA IDENTIFIER RPAREN
        {
            $$ = createNode("Rotate", 2, 0);
            addChild($$, createNode("Angle", 0, $3), 0);
            addChild($$, createNode($5, 0, 0), 1);
        }
        | FLIP LPAREN IDENTIFIER RPAREN
        {
            $$ = createNode("Flip", 1, 0);
            addChild($$, createNode($3, 0, 0), 0);
        }

                                    ...

        | BINARIZATION LPAREN IDENTIFIER RPAREN
        {
            $$ = createNode("Binarization", 1, 0);
            addChild($$, createNode($3, 0, 0), 0);
        }
        | COUNTOR LPAREN IDENTIFIER RPAREN
        {
            $$ = createNode("Countor", 1, 0);
            addChild($$, createNode($3, 0, 0), 0);
        }
        ;
```

Por fim foram implementadas as funções `yyerror` que é a função responsável pelo tratamento de erros usada na análise sintática e a função `parser` que é responsável por iniciar o processo de análise sintática.

```
1  void yyerror(const char *s)
2  {
3    fprintf(stderr, "Error: %s in line %d \n\n", s, line_number);
4  }
5
6  int parser(FILE *inputFile)
7  {
8      if (!inputFile)
9      {
10         printf("Error: Input file not opened.\n");
11         return 1;
12     }
13     yyparse();
14     return 0;
15 }
```

6.1.3 Análise Semântica

Para implementar a análise semântica, foi utilizado um algoritmo *ad hoc* que consiste num *array* de estruturas do tipo *SemanticCheck*, constituída por dois campos: um campo que armazena a operação a ser realizada e outro campo com um apontador para funções que recebe um nó da AST como parâmetro e retorna um inteiro.

```
typedef struct
{
    const char *label;
    int (*checkFunc)(Node *);
} SemanticCheck;

SemanticCheck semanticChecks[] = {
    {"Load", checkLoad},
    {"Assignment", checkAssignment},
    {"Show", checkShow},
    {"Print", checkPrint},
    {"RecognizeText", checkRecognizeText},
    {"Subtraction", checkArithmeticOperation},
    {"Addition", checkArithmeticOperation},
    {"Multiplication", checkArithmeticOperation},
    {"Resize", checkResize},
    {"Rotate", checkRotate},
    {"Flip", checkUnaryOperation},
    {"Binarization", checkUnaryOperation},
    {"Countor", checkUnaryOperation},
    {"Indexing", checkIndexing},
    {"GaussianBlur", checkMultiOperation},
    {"BilateralBlur", checkMultiOperation},
    {"BinaryThreshold", checkBinaryOperation},
    {"BinaryInverseThreshold", checkBinaryOperation},
    {"OtsuThreshold", checkBinaryOperation}
};
```

Como é possível ver no código acima, foram implementadas várias funções, uma para cada operação suportada pela DSL. Como forma de exemplo, será demonstrada abaixo a função responsável por verificar se uma dada imagem foi previamente carregada antes de ser utilizada na operação `show`.

```
                                    ...

// function to check if an image is loaded
int isImageLoaded(char *identifier)
{
    for (int i = 0; i < num_loaded_variables; i++)
    {
        if (strcmp(loaded_variables[i].identifier, identifier) == 0)
            return loaded_variables[i].loaded;
    }
    return 0; // returns 0 if the variable is not found or not loaded
}
                                    ...
int checkShow(Node *node)
{
    char *identifier = node->children[0]->label;
    if (!isImageLoaded(identifier))
    {
        printf("variable %s not loaded\n", identifier);
        return 1; // error detected
    }
    return 0; // no error
}
```

Por fim, em relação à análise semântica, é apresentada a função responsável por esta etapa, na qual é realizada uma travessia em *post-order*. Nesta abordagem, o nó pai é processado apenas após todos os seus filhos terem sido verificados.

De notar que os atributos verificados durante a travessia em *post-order* são, por definição, atributos sintetizados.

```
                                    ...
// semantic analysis function
int analyzeSemantics(Node *node)
{
    if (node == NULL)
        return 0; // return 0 in case of null node

    // loop through semantic checks and apply the appropriate ones based on
    ↪ node label
    for (int i = 0; i < sizeof(semanticChecks) / sizeof(semanticChecks[0]);
    ↪ i++)
    {
        if (strcmp(node->label, semanticChecks[i].label) == 0)
        {
            // if there's an error, stop semantic analysis
            if (semanticChecks[i].checkFunc(node) != 0)
                return 1; // return 1 in case of semantic error
        }
    }

    // recursively analyze children of the current node
    for (int i = 0; i < node->num_children; i++)
    {
        if (analyzeSemantics(node->children[i]) != 0)
            return 1; // return 1 if any of the children has semantic error
    }

    return 0; // return 0 if there's no semantic error
}
```

6.2 Backend

O código seguinte define a interface criada, *ImageProcessingDsl.h*, com um conjunto de classes e operações para processamento de imagens utilizando as bibliotecas *OpenCV* e *Tesseract OCR*. Está estruturado num namespace chamado *ImageProcessingDsl* e oferece diversas funcionalidades para manipulação e análise de imagens.

As bibliotecas incluídas são essenciais para o processamento de imagem e reconhecimento ótico de caracteres (OCR), como a *OpenCV*, utilizada para várias operações de processamento de imagem, a *Tesseract OCR*, utilizada para reconhecimento de texto em imagens, e a *Leptonica*, que serve como suporte ao *Tesseract* para manipulação de imagens.

São definidas constantes para diferentes tipos de operações de desfoque e limiarização, como *GAUSSIAN_BLUR*, *MEDIAN_BLUR*, *BILATERAL_BLUR* e *BINARY_THRESHOLD*.

O código define várias classes para representar imagens e operações sobre elas. A classe *Image* representa uma imagem e inclui métodos para carregar, redimensionar, inverter, rodar, exibir e operar sobre imagens. A classe *ImageOperation* é uma classe base abstrata para operações de imagem, com um método virtual puro *execute* que deve ser implementado pelas subclasses.

Outras classes específicas incluem *Binarization*, que implementa a operação de binarização; *Blur*, que implementa operações de desfoque, como desfoque gaussiano, mediano e bilateral; *Threshold*, que implementa operações de limiarização, incluindo limiarização binária e usando o método de Otsu; *Countor*, que implementa operações de deteção e desenho de contornos; *TextRecognition*, que implementa reconhecimento de texto usando *Tesseract OCR*; *Dsl*, que aplica operações de imagem de forma simplificada; *BinaryThreshold*, que implementa a operação de limiarização binária e *OtsuThreshold*, que implementa a operação de limiarização usando o método de *Otsu*.

```
1 #ifndef _IMAGEPROCESSINGDSL_H_
2 #define _IMAGEPROCESSINGDSL_H_
3
4 #include <opencv2/opencv.hpp>
5 #include <tesseract/baseapi.h>
6 #include <leptonica/allheaders.h>
7 #include <opencv2/highgui.hpp>
8 #include <string>
9 #include <vector>
```

```
10
11  namespace ImageProcessingDsl
12  {
13      #define GAUSSIAN_BLUR 1
14      #define MEDIAN_BLUR 2
15      #define BILATERAL_BLUR 3
16      #define BINARY_THRESHOLD 4
17
18      class Image;
19      class ImageOperation;
20      class Binarization;
21      class Blur;
22      class Threshold;
23      class Countor;
24      class Dsl;
25      class BinaryThreshold;
26
27      class Image
28      {
29      private:
30          int id;
31          cv::Mat image;
32          std::string path;
33      public:
34          Image();
35          Image(std::string path);
36          Image(cv::Mat img);
37          int getId();
38          Image resizeImage(int width, int height);
39          Image flipImage(int flipCode);
40          Image rotateImage(double angle);
41          cv::Mat getImage() const;
42          std::string getImagePath() const;
43          void setImage(cv::Mat img);
```

```
44          void showImage() const;
45          Image operator+(const Image &other);
46          Image operator-(const Image &other);
47          Image operator*(const Image &other);
48          void loop(Image images[], int numImages, Image (Image::*func)(int),
            ↪  int arg);
49          ~Image();
50      };
51
52      class ImageOperation
53      {
54      public:
55          virtual cv::Mat execute(const Image &inputs) const = 0;
56          virtual ~ImageOperation();
57      };
58
59      class Binarization : public ImageOperation
60      {
61      public:
62          cv::Mat execute(const Image &input) const;
63      };
64
65      class Blur : public ImageOperation
66      {
67      private:
68          cv::Size size;
69          double sigma;
70          int mode;
71          int ksize;
72      public:
73          Blur(int mode, cv::Size size = cv::Size(5, 5), double sigma = 0);
74          Blur(int mode, int ksize);
75          Blur(int mode, int ksize, int sigma);
76          cv::Mat execute(const Image &input) const;
```

```
77          static Image apply(const Image &input, int mode, const cv::Size &size,
            ↪  double sigma);
78      };
79
80      class Threshold : public ImageOperation
81      {
82      private:
83          double threshold;
84          double maxVal;
85          int type;
86      public:
87          Threshold(double threshold = 0, double maxVal = 255, int type =
            ↪  cv::THRESH_BINARY_INV | cv::THRESH_OTSU);
88          cv::Mat execute(const Image &input) const;
89      };
90
91      class Countor : public ImageOperation
92      {
93      private:
94          int mode;
95          int method;
96          cv::Scalar color;
97      public:
98          Countor(int mode = cv::RETR_EXTERNAL, int method =
            ↪  cv::CHAIN_APPROX_SIMPLE, cv::Scalar color = cv::Scalar(0, 0,
            ↪  255));
99          void findContours(const Image &input,
            ↪  std::vector<std::vector<cv::Point>> &contours,
            ↪  std::vector<cv::Vec4i> &hierarchy) const;
100         cv::Mat drawContours(const Image &input, const
            ↪  std::vector<std::vector<cv::Point>> &contours) const;
101         cv::Mat execute(const Image &input) const;
102     };
103
```

```
104     class TextRecognition
105     {
106     public:
107         std::string execute(const cv::Mat &input) const;
108         void printText(const std::string input);
109     };
110 
111     class Dsl
112     {
113     public:
114         Dsl();
115         cv::Mat applyOperation(const Image &input, const ImageOperation
            ↪  &operation) const;
116     };
117 
118     class BinaryThreshold : public ImageOperation
119     {
120     private:
121         double threshold;
122         double maxVal;
123         int type;
124     public:
125         BinaryThreshold(double threshold = 0, double maxVal = 255, int type =
            ↪  cv::THRESH_BINARY_INV | cv::THRESH_OTSU);
126         cv::Mat execute(const Image &input) const;
127     };
128 
129     class BinaryInverseThreshold : public ImageOperation
130     {
131     private:
132         double threshold;
133         double maxVal;
134     public:
135         BinaryInverseThreshold(double threshold = 0, double maxVal = 255);
```

```
136         cv::Mat execute(const Image &input) const override;
137     };
138 
139     class OtsuThreshold : public ImageOperation
140     {
141     private:
142         double maxValue;
143     public:
144         OtsuThreshold(double maxValue = 255);
145         cv::Mat execute(const Image &input) const override;
146     };
147 }
148 
149 #endif
```

6.2.1 Geração de Código

A geração de código do compilador da DSL para processamento de imagem é híbrida, permitindo ao programador escolher entre processamento sequencial ou paralelo. Esta escolha é feita através de duas *macros* definidas, `SET_MODE(SEQUENTIAL)` e `SET_MODE(PARALLEL)` que funcionam como diretivas para o compilador e são definidas da seguinte forma:

```
1  SET_MODE(SEQUENTIAL)
2  
3      load (image,"Images/ESRG.png")
4      image1 = resize(1920,1080,image)
5      show (image1)
6  
7  SET_MODE(PARALLEL)
8  
9      load (imagem,"Images/ESRG.png")
10     imagem1 = resize(1920,1080,imagem)
11     show (imagem1)
```

Como é possível ver no código acima, no mesmo ficheiro é possível fazer deployment sequencial e paralelo, o que torna a DSL mais flexível e adequada para um maior leque de aplicações.

```
void translateNode(Node* node, stringstream& code)
{
    // if the node is null return
    if (node == nullptr)
        return;

    // check if the translation mode is sequential
    if (mode == SEQUENTIAL)
    {
        // find the node label in the sequential translation map
        auto it = translationMapSeq.find(node->label);
        if (it != translationMapSeq.end())
            it->second(node, code);
        // if the node has children process them recursively
        if (node->num_children > 0 && node->children != nullptr)
        {
            for (int i = 0; i < node->num_children; ++i)
            {
                translateNode(node->children[i], code); // recursively
                ↪  translate each child node
            }
        }
    }
    // check if the translation mode is parallel
    else if (mode == PARALLEL)
    {
        // find the node label in the parallel translation map
        auto it = translationMapPar.find(node->label);
        if (it != translationMapPar.end())
            it->second(node, code);
        // if the node has children process them recursively
```

```
31          if (node->num_children > 0 && node->children != nullptr)
32          {
33              for (int i = 0; i < node->num_children; ++i)
34              {
35                  translateNode(node->children[i], code); // recursively
                    ↪  translate each child node
36              }
37          }
38      }
39      else
40          cout << "\n\nerror -> no mode selected <- \n\n"; // if no valid
            ↪  translation mode is selected print an error message
41  }
```

A travessia da AST para a geração de código é realizada em *pre-order*. Isto significa que o nó atual é processado antes dos seus filhos, reduzindo assim a sobrecarga na travessia da AST visto que os mapas (unordered_map) mapeiam as *strings* que representam as operações(nós pais) e depois traduzem-nas utilizando as funções correspondentes, como será demonstrado nas secções seguintes.

a) Geração de Código Sequencial

Para a geração de código sequencial em C++ foi definido um mapa chamado `translationMapSeq`, que mapeia *strings* (nomes de operações) para funções de tradução correspondentes. Durante a tradução de uma estrutura de dados (*Node* node*) para código (*stringstream& code*), o código verifica o label (*node->label*) para determinar qual operação deve ser realizada. Uma vez determinada a operação através do label, a função lambda correspondente é chamada, recebendo como parâmetros o `Node* node` atual e a `stringstream& code`, onde o código gerado é acumulado. Essas funções lambdas encapsulam a lógica específica de tradução para cada operação, permitindo uma implementação modular e extensível do sistema de tradução.

```
1  using TranslationFunctionSeq = void (*)(Node*, stringstream&);
```

TranslationFunctionSeq é um alias para um apontador de função que recebe dois parâmetros: um apontador para `Node` e uma referência para *stringstream*, e retorna *void*.

```
unordered_map<string, TranslationFunctionSeq> translationMapSeq = {
    // Mapeamento de operações para funções lambda correspondentes
};
```

translationMapSeq é um mapa não ordenado (*unordered_map*) onde as chaves são *strings* e os valores são do tipo *TranslationFunctionSeq* (apontadores para funções com a assinatura definida anteriormente).

Dentro das chaves do *unordered_map*, estão definidos vários pares de chave-valor. Cada chave é uma *string* que representa uma operação específica. Cada valor é uma função lambda que corresponde à tradução dessa operação para código C++ específico. A seguir apresento alguns exemplos do mapeamento desenvolvido:

```
{
    "Load",
    [](Node* node, stringstream& code)
    {
        code << "ImageProcessingDsl::Image " << node->children[0]->label <<
        ↪  "(" << node->children[1]->label << ");\n";
        code << "\n";
    }
}
```

Quando o *label* "*Load*" é encontrado, a função lambda associada é chamada. Esta função lambda cria uma nova instância de *ImageProcessingDsl::Image* com base nos labels dos nós filhos do nó atual, onde analisando o nó "*Load*" da AST:

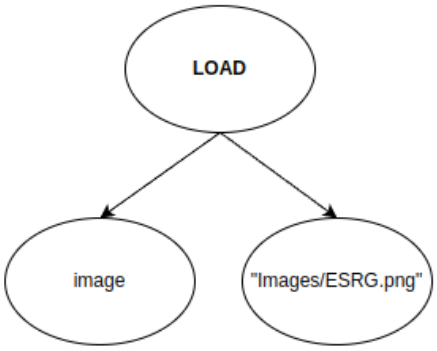

Figure 6.7: Sub-árvore da operação *Load*

Podemos retirar que os labels dos nós filhos (*node->children[0]->label e node->children[1]->label*) dizem respeito ao nome da imagem criada e ao seu respetivo *path*, logo o código gerado neste caso seria:

```
ImageProcessingDsl::Image image("Images/ESRG.png");
```

Já quando o *label "Show"* é encontrado, a função lambda associada é chamada.

```
{
    "Show",
    [](Node* node, stringstream& code)
    {
        code << node->children[0]->label << ".showImage();\n";
        code << "\n";
    }
}
```

Para a operação "*Show*", a função lambda simplesmente chama o método *showImage()* no objeto *Image* correspondente, que no caso, analisando a sub-árvore:

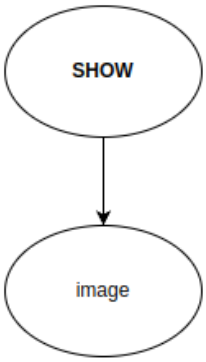

Figure 6.8: Sub-árvore da operação *Show*

O código gerado neste caso seria:

```
image.showImage();
```

b) Geração de Código Paralelo

Além da geração de código em C++ utilizando OpenCV de forma puramente sequencial, foi também implementada uma geração de código com processamento paralelo utilizando PThreads[2]. Para isso foi definido um novo mapa chamado *translationMapPar*, que mapeia *strings* (nomes de operações) para funções de tradução correspondente com processamento paralelo.

```
using TranslationFunctionPar = void (*)(Node*, stringstream&);
unordered_map<string, TranslationFunctionPar> translationMapPar = {
    // Mapeamento de operações para funções lambda correspondentes
};
```

Como forma de comparação, será demonstrado como ocorre a tradução para processamento paralelo. Quando o label "*Show*" é encontrado, cria uma função *lambda* *showImage* na imagem definida

no nó filho. Depois, cria uma *thread* para executar essa *lambda* e espera que a *thread* termine antes de continuar. Cada *thread* e lambda gerada recebe um identificador único para evitar conflitos.

```
{
    "Show",
    [](Node* node, stringstream& code)
    {
        code << endl;
        code << "auto showImage" << numThreads << " = [&" <<
        ↪  node->children[0]->label << "]() -> void* {\n";
        code << node->children[0]->label << ".showImage();\n";
        code << "    return nullptr;\n";
        code << "};\n";
        code << "pthread_t show_thread" << numThreads << ";\n";
        code << "pthread_create(&show_thread" << numThreads << ", nullptr,
        ↪  [](void* arg) { return (*static_cast<decltype(showImage" <<
        ↪  numThreads << ")*>(arg))(); }, &showImage" << numThreads <<
        ↪  ");\n";
        code << "pthread_join(show_thread" << numThreads << ", nullptr);\n";
        numThreads++;
    }
}
```

A definição da função lambda interna *showImage* dentro do próprio bloco de código facilita a sua definição e execução no mesmo *scope*. Isto permite uma maior clareza e organização do código, pois a função é definida exatamente onde é usada, sem a necessidade de declarações separadas.

O código gerado neste caso seria o seguinte:

```
auto showImage1 = [&image1]() -> void* {
    image1.showImage();
```

```
    return nullptr;
};
pthread_t show_thread1;
pthread_create(&show_thread1, nullptr, [](void* arg) { return
↪  (*static_cast<decltype(showImage1)*>(arg))(); }, &showImage1);
pthread_join(show_thread1, nullptr);
```

Este código define e executa uma operação de mostrar uma imagem (*showImage*) de forma paralela usando *PThreads*, permitindo que a operação seja realizada numa *thread* separada, melhorando a eficiência e o desempenho em cenários onde múltiplas operações paralelas são necessárias.

6.3 Compilação do Código Gerado para Executável

Para a compilação do código gerado para um executável, foi implementada uma função específica. Esta função, chamada `compileCode`, é responsável por compilar o código C++ gerado pelo compilador da DSL para processamento de imagem.

A função `compileCode` realiza as seguintes operações:

1. **Criação de um ficheiro temporário**:
    - A função começa por criar um ficheiro temporário chamado `Output/tempCode.cpp`.
    - Se o ficheiro for aberto com sucesso, o código gerado é escrito no ficheiro temporário.
    - Caso a abertura do ficheiro falhe, é exibida uma mensagem de erro e a função retorna um código de erro.
2. **Comando de compilação**:
    - A função define o comando de compilação utilizando o compilador `g++`.
    - O comando inclui o ficheiro temporário *`tempCode.cpp`* e o ficheiro *`ImageProcessingDsl.cpp`*.
    - Adicionalmente, são especificadas as *flags* e bibliotecas necessárias, utilizando *`pkg-config`* para as *flags* e bibliotecas do OpenCV e ligando as bibliotecas *`tesseract`* e *`lept`*.
3. **Execução da compilação**:
    - O comando de compilação é executado utilizando a função `system`.
    - Se a compilação for bem-sucedida, é exibida uma mensagem de sucesso.
    - Caso contrário, é exibida uma mensagem de erro e a função retorna um código de erro.
4. **Resultado da compilação**:
    - A função retorna 0 se a compilação for bem-sucedida, ou 1 se ocorrer algum erro durante o processo.

Segue-se o código completo da função `compileCode`:

```
int compileCode(const std::string& cCode)
{
    std::ofstream tempFile("Output/tempCode.cpp");
    if (tempFile.is_open())
    {
        tempFile << cCode << std::endl;
        tempFile.close();
    }
    else
    {
        std::cerr << "Error: Failed to open tempCode.cpp for writing" <<
        ↪  std::endl;
        return 1;
    }

    std::string compileCommand = "g++ Output/tempCode.cpp
    ↪  Backend/ImageProcessingDsl.cpp -o Output/output ";
    compileCommand += "$(pkg-config --cflags --libs opencv4) ";
    compileCommand += "-ltesseract -llept";

    int compileResult = system(compileCommand.c_str());

    if (compileResult == 0)
        std::cout << "Compilation successful!\n" << std::endl;
    else
    {
        std::cerr << "Compilation failed!" << std::endl;
        return 1;
    }

    return 0;
}

```

7 Verificação

7.1 Makefile

Para simplificar o processo de compilação e organização do código, foi criado um *Makefile* com o seguinte conteúdo:

```
all: setup lexer parser src
setup:
    mkdir -p Output
lexer: setup
    flex -o Output/lex.yy.c Frontend/Scanner/LexScanner.l
parser: setup
    bison -dv -o Output/parser.tab.c Frontend/Parser/parser.y
src: setup lexer parser
    g++ -c -o Output/genCode.o Backend/genCode.cpp
    ↪ -I/usr/local/include/opencv4 -I/usr/include/tesseract
    ↪ -I/usr/include/clang -L/usr/local/lib -lopencv_core -lopencv_highgui
    ↪ -lopencv_imgproc -lopencv_imgcodecs -ltesseract -pthread
    g++ -o Output/myComp Utils/utils.c Output/parser.tab.c Output/lex.yy.c
    ↪ Frontend/Semantic/semantic.c main.c Output/genCode.o -lstdc++
    ↪ -I/usr/local/include/opencv4 -I/usr/include/tesseract -L/usr/local/lib
    ↪ -lopencv_core -lopencv_highgui -lopencv_imgproc -lopencv_imgcodecs
    ↪ -ltesseract
clean:
    rm -rf Output
prog: all
    ./Output/myComp input.txt
    ./Output/output
run: clean prog
```

Este Makefile automatiza várias etapas do processo de construção, incluindo a criação de diretórios, a geração de ficheiros de análise léxica e sintática, a compilação dos ficheiros fonte, e a criação do executável final. Além disso, inclui alvos para limpar os ficheiros gerados e para executar o programa com um ficheiro de entrada especificado, que no caso se trata do código fonte da DSL.

7.2 Stream de tokens

Como forma de verificar a análise léxica, foi escrito um código-fonte de teste.

```
SET_MODE(SEQUENTIAL)

    load (image1,"Images/countor.jpg")
    load (image2,"Images/ESRG.png")
    load (image3,"Images/test1.png")

    _my_array = [image1,image2,image3]

    testing_index = _my_array !! 1

    image5 = resize(800,600,testing_index)

    image6 = binarization(image5)

    show (image6)
```

Para o código de entrada definido acima, a sequência de tokens gerados pelo analisador léxico é apresentada abaixo:

```
SET_MODE LPAREN SEQUENTIAL_MODE RPAREN

LOAD LPAREN IDENTIFIER COMMA PATH RPAREN
LOAD LPAREN IDENTIFIER COMMA PATH RPAREN
LOAD LPAREN IDENTIFIER COMMA PATH RPAREN

ARRAY ASSIGNMENT LBRACKET IDENTIFIER COMMA IDENTIFIER COMMA IDENTIFIER RBRACKET

IDENTIFIER ASSIGNMENT ARRAY DOUBLE_EXCLAMATION INT

IDENTIFIER ASSIGNMENT RESIZE LPAREN INT COMMA INT COMMA IDENTIFIER RPAREN

IDENTIFIER ASSIGNMENT BINARIZATION LPAREN IDENTIFIER RPAREN

SHOW LPAREN IDENTIFIER RPAREN


END OF FILE
```

Figure 7.9: Stream de tokens

7.3 Geração da AST

Para o código-fonte definido acima, a AST gerada pelo analisador sintático é a seguir apresentada em representação textual:

```
Mode
        Sequential
Load
        image1
        "Images/countor.jpg"
Load
        image2
        "Images/ESRG.png"
Load
        image3
        "Images/test1.png"
Array
        _my_array
        IdentifierList
                image1
                image2
                image3
Assignment
        testing_index
        Indexing
                _my_array
                Index
Assignment
        image5
        Resize
                Width
                Height
                testing_inde
Assignment
        image6
        Binarization
                image5
Show
        image6
```

Figure 7.10: Representação textual da AST

Nesta representação, os filhos dos respetivos nós pais estão indentados com um *tab* à frente, para facilitar a visualização da hierarquia da AST.

De acordo com a imagem, é possível verificar o correto funcionamento do analisador sintático.

7.4 Geração da código sequencial

Para o código de entrada especificado, o código gerado com processamento sequencial é o seguinte:

Output > tempCode.cpp > ...

```cpp
#include "../Backend/ImageProcessingDsl.h"
#include <iostream>
#include <pthread.h>
#include <chrono>
#include <vector>

int main() {
ImageProcessingDsl::Binarization binarization;

ImageProcessingDsl::Image image1("Images/countor.jpg");
ImageProcessingDsl::Image image2("Images/ESRG.png");
ImageProcessingDsl::Image image3("Images/test1.png");

std::vector<ImageProcessingDsl::Image> _my_array = {image1 ,image2 ,image3};
ImageProcessingDsl::Image testing_index = _my_array[1];

ImageProcessingDsl::Image image5 = testing_index.resizeImage(800, 600);

ImageProcessingDsl::Image image6 = binarization.execute(image5);

image6.showImage();

return 0;
}
```

Figure 7.11: Código gerado pelo Compilador da DSL de processamento de imagem

7.5 Geração da código paralelo

Para que o compilador gere código em paralelo, foi necessário alterar a macro na primeira linha do código de entrada para:

```
SET_MODE(PARALLEL)
```

O código gerado com processamento paralelo é:

Output > tempCode.cpp > ...

```cpp
#include "../Backend/ImageProcessingDsl.h"
#include <iostream>
#include <pthread.h>
#include <chrono>
#include <vector>

int main() {
ImageProcessingDsl::Binarization binarization;

ImageProcessingDsl::Image image1("Images/countor.jpg");
ImageProcessingDsl::Image image2("Images/ESRG.png");
ImageProcessingDsl::Image image3("Images/test1.png");

std::vector<ImageProcessingDsl::Image> _my_array = {image1 ,image2 ,image3};

ImageProcessingDsl::Image testing_index;
testing_index=_my_array[1];

ImageProcessingDsl::Image image5;
auto resizeImage0 = [&testing_index, &image5]() -> void* {
    image5=testing_index.resizeImage(800, 600);
    return nullptr;
};
pthread_t resize_thread0;
pthread_create(&resize_thread0, nullptr, [](void* arg) { return (*static_cast<decltype(resizeImage0)*>(arg))(); }, &resizeImage0);
pthread_join(resize_thread0, nullptr);

ImageProcessingDsl::Image image6;
auto binarizeImage1 = [&image5, &image6, &binarization]() -> void* {
    image6=binarization.execute(image5);
    return nullptr;
};
pthread_t binarize_thread1;
pthread_create(&binarize_thread1, nullptr, [](void* arg) { return (*static_cast<decltype(binarizeImage1)*>(arg))(); }, &binarizeImage1);
pthread_join(binarize_thread1, nullptr);

auto showImage2 = [&image6]() -> void* {
image6.showImage();
    return nullptr;
};
pthread_t show_thread2;
pthread_create(&show_thread2, nullptr, [](void* arg) { return (*static_cast<decltype(showImage2)*>(arg))(); }, &showImage2);
pthread_join(show_thread2, nullptr);

return 0;
}
```

Figure 7.12: Código gerado pelo Compilador da DSL de processamento de imagem

7.6 Testes e resultados

Nesta secção, vou apresentar as capacidades da DSL e do compilador correspondente, destacando algumas das suas funcionalidades e características distintas, além dos resultados obtidos com cada código-fonte.

Como mencionado na secção de Design da Solução, onde descrevi a sintaxe e a semântica da linguagem, também foram demonstradas as operações suportadas e as funcionalidades da mesma.

7.6.1 Detecção de contorno

O primeiro teste visa demonstrar a capacidade de detecção de contornos em uma imagem usando apenas um comando da DSL.

Para a seguinte imagem original com o nome ”counter_detection_img.png” foi executado o seguinte código.

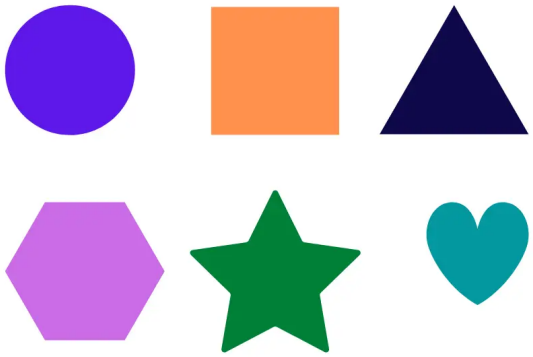

Figure 7.13: Imagem original "counter_detection_img.png"

```
load (image1,"Images/counter_detection_img.png")

image2 = countor(image1)

show (image2)
```

O resultado foi:

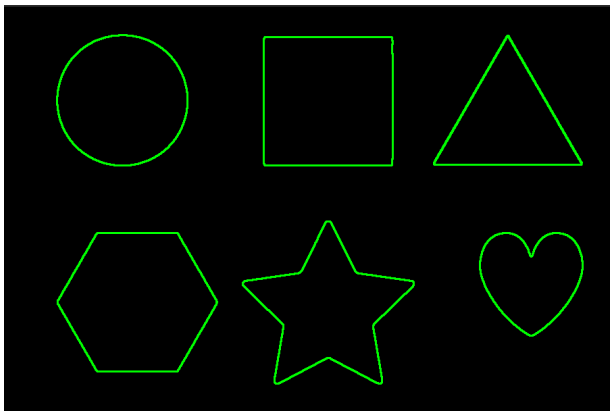

Figure 7.14: Resultado da imagem "counter_detection_img.png modificada"

7.6.2 Reconhecimento de texto

O próximo teste tem como objetivo demonstrar a capacidade de reconhecimento de texto proporcionada pela DSL.

A imagem que contém o texto a ser reconhecido é apresentada abaixo:

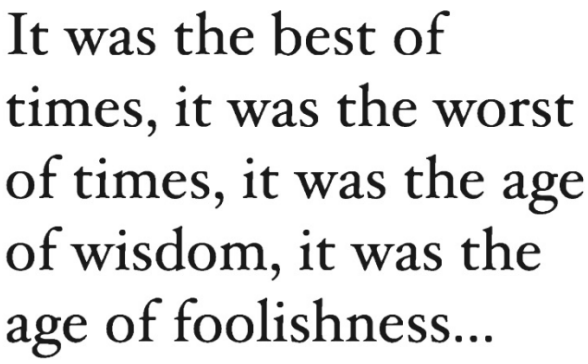

Figure 7.15: Texto presente na imagem "text.png"

```
1 load (image,"Images/text.png")
2
3 textRecognition (image, myText)
4
5 print (myText)
```

O texto reconhecido é então impresso na consola, como é possível verificar:

```
Recognized Text: It was the best of
times, it was the worst
of times, it was the age
of wisdom, it was the
age of foolishness...
```

Figure 7.16: Texto reconhecido na imagem "text.png"

O mesmo teste foi realizado com uma imagem que contém cores e texto em diferentes regiões.

Figure 7.17: Texto presente na imagem "text2.jpg"

Com o mesmo código:

```
load (image2,"Images/text2.jpg")

textRecognition (image2, myText2)

print (myText2)
```

O resultado foi como esperado:

```
Recognized Text: This is SAMPLE TEXT

Text is at different regions
```

Figure 7.18: Texto reconhecido na imagem "text2.jpg"

7.6.3 Manipulação de arrays de imagens e ciclos

Além do acesso a elementos específicos num array de imagens, a linguagem também suporta uma função *loop* que executa uma operação específica em todos os elementos de um *array* de imagens.

Para facilitar a demonstração, consideremos um array de imagens com três elementos e desejamos aplicar binarização, redimensionar e posteriormente mostrar ambas. O código seria o seguinte:

```
load (image1,"Images/um.png")
load (image2,"Images/ESRG.png")
load (image3,"Images/check.jpg")

_my_array = [image1,image2,image3]

loop(_my_array,binarization())

loop(_my_array,resize(640,480))

loop(_my_array,show())
```

Tendo em conta as imagens originais:

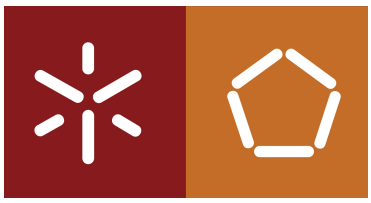

Figure 7.19: Imagem original "um.png"

Figure 7.20: Imagem original "ESRG.png"

Figure 7.21: Imagem original "check.jpg"

O resultado é, como esperado:

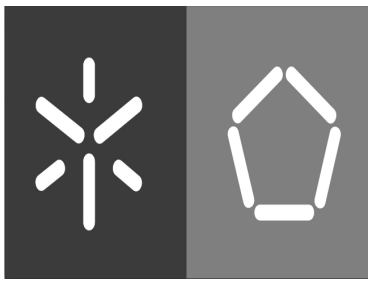

Figure 7.22: Imagem final "um.png"

Figure 7.23: Imagem final "ESRG.png"

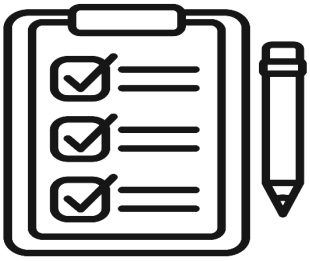

Figure 7.24: Imagem final "check.jpg"

7.6.4 Pipeline de operações

A linguagem, como foi anteriormente mencionado, também suporta a execução de um pipeline de operações, que é uma sequência de operações encadeadas a serem aplicadas a uma imagem.

Pode ser definida da segunite forma:

```
1 load (imagem,"Images/um.png")
2
3 compile >> imagem >> flip() >> resize(1280,768) >> rotate(180) >>
  ↪  binarization() >> gaussianBlur(3,4)
```

O código acima executa, de forma sequencial, as seguintes operações na imagem: espelhamento, redimensionamento, rotação de 180 graus, binarização e por fim, aplica um blur gaussiano com kernel de 3x3 e sigma de 4.

Abaixo é possível ver a imagem original e o resultado após as transformações.

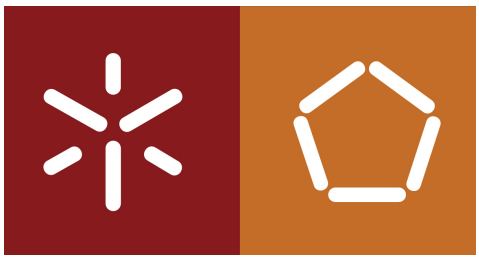

Figure 7.25: Imagem original

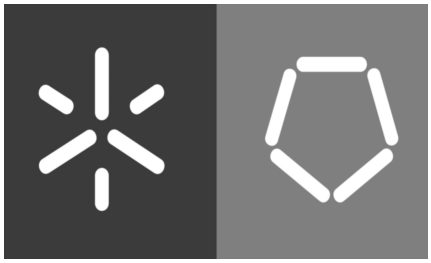

Figure 7.26: Imagem final

7.6.5 Desempenho: Processamento Sequencial vs. Paralelo

Como forma de medir o desempenho do código gerado na configuração sequencial e na configuração paralela, foi implementada uma nova função na DSL que serve para medir o tempo entre dois instantes de execução, apenas com o objetivo de estudar a performance do código gerado nas duas configurações. Segue-se um exemplo do uso dessa nova funcionalidade.

```
load (image,"Images/um.png")

SET_MODE(SEQUENTIAL)
    timer(on)
        image_seq = gaussianBlur(1,2,image)
    timer(off)

SET_MODE(PARALLEL)
    timer(on)
        image_par = gaussianBlur(1,2,image)
    timer(off)
```

Para o código acima, foram verificados estes tempos.

```
Sequential execution time: 788 us
Parallel execution time: 987 us
```

Figure 7.27: Tempos relativos a uma única operação

Como é possível verificar, o tempo da execução sequencial é menor que o tempo da execução paralela, o que é o esperado, dado que se trata apenas de uma operação da DSL a ser executada. O tempo relativo à criação da *thread* e o *overhead* associado, essencialmente na transferência da imagem, fazem com que o tempo da execução sequencial seja menor. No entanto, isso já não se verifica com um número significativo de operações, como é possível ver abaixo.

```
1  load (image,"Images/um.png")
2  load (textImg,"Images/text.png")
3  SET_MODE(SEQUENTIAL)
4      timer(on)
5          imagem1 = binarization(image)
6          imagem2 = flip(imagem1)
7          imagem3 = resize(1600,1080,imagem2)
8          imagem4 = gaussianBlur(1,2,imagem3)
9          textRecognition (textImg, stringText1)
10         imagem_seq = countor(image)
11     timer(off)
12 SET_MODE(PARALLEL)
13     timer(on)
14         imagem5 = binarization(image)
15         imagem6 = flip(imagem5)
16         imagem7 = resize(1600,1080,imagem6)
17         imagem8 = gaussianBlur(1,2,imagem7)
18         textRecognition (textImg, stringText2)
19         imagem_par = countor(image)
20     timer(off)
```

```
Parallel execution time: 142060 us
Sequential execution time: 147619 us
```

Figure 7.28: Tempos relativos a várias operações

8 Conclusão e Trabalho Futuro

Todos os objetivos delineados para este projeto foram cumpridos com sucesso. Além disso, foi realizado trabalho adicional significativo no desenvolvimento do processamento paralelo, expandindo as capacidades da DSL e do compilador além das expectativas iniciais.

Ao longo do projeto, foram desenvolvidas diversas componentes técnicas em várias áreas. Especificamente, houve avanços notáveis nas áreas de compiladores, DSLs e bibliotecas orientadas ao processamento de imagem. As ferramentas utilizadas foram consideradas opções válidas e atuais dentro do estado da arte.

Além das competências técnicas, o projeto também proporcionou o desenvolvimento de competências transversais (*soft skills*), como a capacidade de apresentar e discutir ideias de forma eficaz, aprimorada através de apresentações e reuniões realizadas durante o desenvolvimento do projeto.

Como trabalho futuro, propõem-se várias melhorias e extensões ao projeto:

- **Implementação de Monad Transformers**: A implementação de *monad transformers* poderá melhorar a flexibilidade e a composição das operações da DSL.
- **Geração de Código para C com Backend para LLVM**: Desenvolver a capacidade de gerar código C, utilizando o LLVM como *backend*, permitirá otimizações mais avançadas e um maior desempenho.
- **Uso da Ferramenta Vitis HLS para Deployment em FPGA**: Utilizar a ferramenta Vitis HLS permitirá o *deployment* para FPGAs, proporcionando uma aceleração significativa das operações de processamento de imagem.
- **Uso de CUDA para Deployment em GPU**: A implementação de suporte para CUDA permitirá o *deployment* em GPUs, tirando partido da capacidade de processamento paralelo das GPUs para acelerar ainda mais as operações de processamento de imagem.

Estas melhorias e extensões contribuirão para aumentar a eficiência e a aplicabilidade da DSL, proporcionando uma ferramenta ainda mais poderosa e versátil para a manipulação e processamento de imagens.

Bibliography

[1] John Backus et al. “Revised Report on the Algorithmic Language ALGOL 60”. In: *Communications of the ACM* 6.1 (Jan. 1963). [Online; accessed March 3, 2023], pp. 1–17. ISSN: 0001-0782. DOI: 10.1145/366062.366075. URL: https://dl.acm.org/doi/10.1145/366062.366075.

[2] David R. Butenhof. *POSIX Threads Programming*. POSIX Standard. 1997. URL: https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html.

[3] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. ISBN: 978-0-201-63392-4.

[4] Wikipedia. *Domain Specific Language*. https://en.wikipedia.org/wiki/Domain-specific_language. 2024.

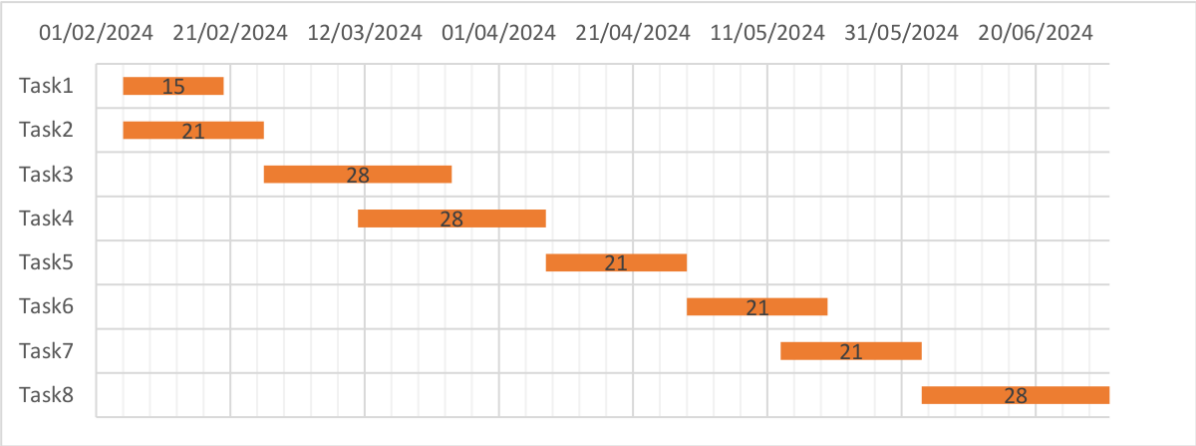

Figure 8.29: Diagrama de Gantt