Image Manipulation Language

Carlos Vieira*

April 27, 2019

1 Introdução

O objetivo desse documento é descrever o trabalho final desenvolvido para a disciplina de Linguagens de Domínio Específico (MPES0019 - Tópicos Avançados em Engenharia de Software I), ministrada pelo Prof. Dr. Sérgio Queiroz de Medeiros, dentro do Programa de Pós-graduação em Engenharia de Software (PPgSW) da Universidade Federal do Rio Grande do Norte (UFRN).

O trabalho tinha como objetivo geral projetar e implementar uma linguagem de domínio específico. A proposta especifíca desse projeto se deve a dois fatores: primeiramente, muitas pessoas que trabalham com edição e manipulação de imagens não tem conhecimento de linguagens de programação de propósito geral que possam usar para esse propósito. Além disso, a alternativa de usar linguagens de scripting em programas de edição de imagem (e.g. AppleScript ou VBScript com photoshop), além de necessitar de compreensão de características da linguagem irrelevantes para o processo de edição, é extremamente acoplada ao programa em uso.

Por outro lado, uma linguagem de domínio específico (DSL - *Domain Specific Language*) voltada a esse propósito, além de poder ser mais simples de aprender, poderia ser mais clara e legível, e independente de qual ferramente de edição é usada na sua implementação. Por isso, propus o desenvolvimento da *Image Manipulation Language* (IML), com essas vantagens em mente. O projeto dessa linguagem pode ser melhor compreendido através da sua sintaxe e da sua semântica, descritas nas sessões 3.1 e 3, respectivamente.

Além do projeto da linguagem, foi necessário implementá-la. Isso envolveu, mais especificamente, o desenvolvimento de um parser descendente recursivo, discutido na sessão 4; e geração de código em outra linguagem (no caso dessa implementação, Python), com o auxílio de *Abstract Syntax Trees* (AST), discutida na sessão 5. Na sessão 6 oferecemos alguns exemplos, e discutimos os resultados assim como possíveis melhorias futuras para a implementação. No apêndice A, oferecemos instruções simples de como utilizar essa implementação.

2 Sintaxe da IML

A sintaxe da IML é descrita abaixo. O único elemento não presente nessa, por claridade, são comentários: uma linha, quando iniciada pelo caractere '#', é ignorada.

*vieiramecarlos@gmail.com

2.1 Programa

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{command}* \\
\textit{command} &\rightarrow \textit{stmt} \mid \textit{img-expr} \\
\textit{stmt} &\rightarrow \textit{assignment} \mid \textit{export} \mid \textit{print} \mid \textit{for-each} \\
\textit{assignment} &\rightarrow \textit{id}\ \texttt{`='}\ \textit{expression} \\
\textit{export} &\rightarrow \texttt{`save'}\ \textit{expression}\ \texttt{`as'}\ \textit{expression} \\
\textit{print} &\rightarrow \texttt{`print'}\ \textit{expression} \\
\textit{for-each} &\rightarrow \texttt{`for'}\ \texttt{`all'}?\ \textit{id}\ \texttt{`in'}\ \textit{expression}\texttt{`\{'}\ \textit{command}*\ \texttt{`\}'}
\end{aligned}
$$

2.2 Expressões de imagem

$$
\begin{aligned}
\textit{img-expr} &\rightarrow \textit{geom-expr} \mid \textit{color-expr} \\
\textit{geom-expr} &\rightarrow \textit{rotate-expr} \mid \textit{resize-expr} \mid \textit{crop-expr} \mid \textit{flip-expr} \\
\textit{color-expr} &\rightarrow \textit{modify-expr} \\
\textit{rotate-expr} &\rightarrow \texttt{`rotate'}\ \textit{expression}\ \texttt{`by'}\ \textit{expression} \\
\textit{flip-expr} &\rightarrow \texttt{`flip'}\ \textit{expression}\ (\texttt{`vertically'} \mid \texttt{`horizontally'}) \\
\textit{resize-expr} &\rightarrow \texttt{`resize'}\ \textit{expression}\ (\texttt{`to'} \mid \texttt{`by'})\ \textit{expression} \\
\textit{crop-expr} &\rightarrow \texttt{`crop'}\ \textit{expression}\ \texttt{`from'}\ \textit{expression} \\
\textit{modify-expr} &\rightarrow \texttt{`modify'}\ \textit{expression}\ \textit{enhancement}\ \texttt{`by'}\ \textit{expression} \\
\textit{enhancement} &\rightarrow \texttt{`sharpness'} \mid \texttt{`brightness'} \mid \texttt{`contrast'} \mid \texttt{`color'}
\end{aligned}
$$

2.3 Expressões

$$
\begin{aligned}
\textit{expression} &\rightarrow \textit{arith-expr} \\
\textit{arith-expr} &\rightarrow \textit{term} \mid \textit{arith-expr}\ (\texttt{`+'} \mid \texttt{`-'})\ \textit{term} \\
\textit{term} &\rightarrow \textit{factor} \mid \textit{term}\ (\texttt{`*'} \mid \texttt{`/'})\ \textit{factor} \\
\textit{factor} &\rightarrow \textit{atom} \mid \texttt{`-'}\ \textit{factor} \\
\textit{atom} &\rightarrow \textit{primary} \mid \textit{primary}\ \texttt{`('}\ \textit{channel}\ \texttt{`)'} \\
\textit{channel} &\rightarrow \texttt{`R'} \mid \texttt{`G'} \mid \texttt{`B'} \\
\textit{primary} &\rightarrow \textit{id} \mid \texttt{`('}\ \textit{expression}\ \texttt{`)'} \mid \textit{scalar} \mid \textit{import-expr} \mid \textit{img-expr} \mid \textit{shape-expr}
\end{aligned}
$$

2.4 Expressões primitivas

$$
\begin{aligned}
\textit{import-expr} &\rightarrow \texttt{`image' `in'}\ \textit{expression} \\
\textit{shape-expr} &\rightarrow \textit{dimensions} \mid \textit{section} \\
\textit{dimensions} &\rightarrow \texttt{`('}\textit{expression}\texttt{`,'}\textit{expression}\texttt{`)'} \\
\textit{section} &\rightarrow \texttt{`('}\textit{expression}\texttt{`,'}\textit{expression}\texttt{`,'}\textit{expression}\texttt{`,'}\textit{expression}\texttt{`)'} \\
\textit{scalar} &\rightarrow \textit{float} \mid \textit{integer} \mid \textit{path} \\
\textit{id} &\rightarrow [a\text{-}zA\text{-}Z]+ \\
\textit{integer} &\rightarrow [0\text{-}9]+ \\
\textit{float} &\rightarrow \textit{integer}(\texttt{`.'}\textit{integer})? \\
\textit{path} &\rightarrow \texttt{"}[\hat{}\texttt{"}]*\texttt{"}
\end{aligned}
$$

3 Semântica da IML

3.1 Programa

Um programa na linguagem IML é essencialmente uma sequência de comandos. Um comando pode ser da forma de uma produção *stmt* ou de uma produção *img-expr* (ver sessão). No primeiro caso, teremos um de quatro comandos: uma atribuição "*id* `=` *expr*", que atribui o valor de *expr* a uma variável referida como *id*; uma exportação "`save` *expr* `as` *expr*", que salva a imagem correpondente a primeira expressão, no caminho correspondente a segunda; uma impressão "`print` *expr*", que mostra a expressão temporariamente em um visualizador, caso seja uma imagem, ou a imprime na saída padrão, caso contrário; ou um laço.

Esse laço é iniciado da forma "`for` *id* `in` *expr*", onde a expressão deve corresponder a um caminho (ver sessão 3.2), e mais especificamente, a um diretório. Dessa forma, *id* recebe o valor do caminho de um arquivo diferente nesse diretório a cada iteração. Adicionar a palavra chave opcional `all` após `for` significa que essa iteração será recursiva dentro do diretório dado.

Caso um comando não seja equivalente a uma produção de *stmt*, ele pode ser uma de *img-expr*, o que representa uma operação sobre uma imagem (ver sessão 3.3). Nesse caso, a expressão sobre a qual se está fazendo essa operação deve ser um *id*, que irá receber o resultado da operação sobre si mesma.

3.2 Tipos e expressões

Há seis tipos de dados na IML:

- Image, ou imagem;
- Integer, ou Int, um inteiro;
- Float, um número de ponto-flutuante;

- Path, ou caminho, uma *string* correspondente a um caminho para um arquivo ou diretório;
- Dimensions, ou Dims, as dimensões (em pixels) de uma imagem, consiste de uma tupla (largura, altura);
- Section, uma sessão retangular de uma imagem, consisite de uma tupla contendo, respectivamente, os limites (em pixels) esquerdo, superior, direito, e inferior da sessão.

Note que para propósitos de Sections, a imagem tem sua origem no canto superior esquerdo. Qualquer expressão em IML deve necessariamente ser de um desses seis tipos. Uma expressão pode ser obtida através de operações sobre outras expressões (ver sessões 3.3 e 3.4), ou através de uma das produções primitivas, apresentadas na sessão 2.4. Variáveis com um tipo apropriado podem ser usadas em lugar de uma expressão através de seu identificador (`id`), contanto que já tenham sido atribuidas um valor (caso contrário, um erro de semântica estática será emitido durante a compilação).

3.3 Operações sobre imagens

Operações sobre imagens são sentenças de IML que produzem uma imagem a partir de outra, e podem ser usadas como expressões, representando o valor da imagem produzida, ou como comandos, alterando a imagem original (ver sessão 3.1). Elas são cinco:

- `r`otate, rotaciona uma imagen no sentido antihorário pelo número dado (em graus);
- `f`lip, inverte uma imagem verticalmente ou horizontalmente;
- `r`esize, redimensiona uma imagem para as dimensões dadas, ou pelo fator dado;
- `c`rop, recorta uma sessão dada de uma imagem, produzindo uma nova imagem desse recorte;
- `m`odify, modifica um atributo de cor de uma imagem por um dado fator.

O atributo de cor, mencionado em `modify`, pode ser um de: *sharpness*, *brightness*, *contrast*, ou *color*.

3.4 Operações unárias e binárias

Além das operações sobre imagens, há duas operações unárias, e quatro binárias, que podem operar sobre imagens e outros tipos. Descrevemos aqui quais dessas operações são válidas, e então o que resulta de cada uma delas. Quanto às operações unárias, há duas: a do `-` unário, que é válido somente para tipos numéricos (Integer e Float), e inverte o sinal do operando; e o operador de acesso a canal, que pode ser da forma `(R)`, `(G)`, ou `(B)`, e é válido somente para imagens, produzindo uma imagem contendo somente o canal respectivo do operando (vermelho, verde, ou azul, respectivamente).

Quanto às operações binárias, há quatro: `+`, `-`, `*`, e `/`. Aparecem na tabela 3.4 somente as operações válidas, para um operando esquerdo correspondente ao tipo da linha, e um direito ao da coluna. Além disso, são marcadas as operações não comutativas, ou seja, que produzem um resultado distinto dependendo da ordem dos operandos.

	Image	Int	Float	Path	Dims	Section
Image	+, -, *, /	+, *, /	+, *, /	+, *	+, *	-
Int	+, *	+, -, *, /	+, -, *, /	+, *	+, *	
Float	+, *	+, -, *, /	+, -, *, /			
Path				+		
Dims		+, *, /	+, *, /		+, -, *, /	*
Section	-	+, *, /	+, *, /		*	+, -, *, /

Quando ambos os operandos são imagens, temos os seguintes comportamentos: para `+`, a segunda imagem é simplesmente sobreposta sobre a primeira; para `-`, a difereça absoluta é obtida entre as duas imagens, pixel a pixel; para `*`, as duas imagens são compostas, ou seja, sobrepostas com igual contribuição para a imagem resultante; e para `/`, temos o mesmo comportamento de `*`, mas entre a primeira imagem e o inverso (em relação a cor) da segunda. Quando ambos os operandos são de tipos númericos, temos o comportamento usual de cada operação, e quando ambos os operandos são caminhos, `+` produz a concatenação desses.

Quando ambos os operandos são iguais e de tipo Dims ou Section, a operação é realizada numericamente, elemento a elemento, produzindo uma expressão de mesmo tipo. Temos um comportamento similar (operação realizada elemento a elemento) quando um dos operandos é de tipo Dims ou Section, e outro é de tipo númerico. Quando um dos operandos é de tipo Dims e o outro Section, a sessão é vista como uma proporção das dimensões, e `*` produz a sessão apropriadamente dimensionada. Quando um dos operandos é uma imagem e o outro uma sessão, temos a operação `-`, que remove uma sessão da imagem, ou remove toda a imagem exceto aquela parte dentro da sessão, dependendo da ordem.

4 Parser

O parser para essa linguagem (assim como o restante do compilador), foi escrito em C++, sem auxílio de ferramentas para geração de parsers. Antes de realizado o parsing em si, a entrada é agrupada em tokens, ignorando espaços em branco, que carregam informações do conteúdo (ou texto), tipo, e localização (linha e coluna). Esses tokens são então passados para o parser em si, que é um parser descendente recursivo (*top-down*) LL(1).

O parser não apenas verifica a corretude sintática do programa, mas também constrói uma árvore sintática abstrata (AST - *Abstract Syntax Tree*) que facilitará o restante do processo de compilação, em especial a geração de código. Erros sintáticos são reportados na saída padrão, com informações de posição (linha, coluna) e forma esperada, e interrompem o processo de parsing.

5 Geração de Código

Após a análise léxica e a contrução da AST pelo parser, passamos a geração de código. Essa implementação gera código equivalente em Python a partir de um programa válido

em IML. Esse processo envolve duas travessias pela AST. Na primeira, são determinados os tipos de expressões compostas e de variáveis, assim como verificados problemas de semântica estática, como operações indefinidas ou uso de variáveis não inicializadas. Esses problemas, se encontrados, são assim como no parser reportados na saída padrão, e interrompem o processo de geração de código.

Na segunda, passamos a geração de código em si. Para isso, são necessárias algumas funções auxiliares (em Python) definidas no início de cada programa gerado. Essas funções servem somente para facilitar o processo de compilação, e gerar um código um pouco menor e mais claro. Uma vez definidas essa funções (brevemente descritas abaixo), o código equivalente é gerado para cada nó da árvore e acumulado, e então escrito para um arquivo de saída pré-definido (ver apêndice A). Note que para as operações de manipulação de imagens, são usadas várias funções providas pelo pacote Pillow de manipulação de imagens para Python, do qual o código gerado depende.

- `_resize`: redimensionar imagem dada por dimensão dada, usada pela operação de imagem de mesmo nome (ver sessão 3.3).
- `_crop`: obter sessão dada contida em imagem dada, usada pela operação de imagem de mesmo nome.
- `_add`: realizar operação `+` entre duas imagens.
- `_sub`: realizar operação `-` entre duas imagens.
- `_mult`: realizar operação `*` entre duas imagens.
- `_div`: realizar operação `/` entre duas imagens.
- `_inv`: inverter cor de imagem, usada por `_div`.
- `_rm_sec`: remove sessão de imagem, usada pela operação `-` entre imagem e sessão.
- `_scale`: redimensionar sessão, usada pela operação `-` entre tipos Dims e Section.

Note que toda função padrão se inicia com um *underscore* (_), para evitar conflitos com variáveis definidas pelo usuário de IML.

6 Resultados

Abaixo, podemos ver um exemplo de programa válido em IML, e o programa equivalente gerado por essa implementação em Python (comentários adicionados posteriormente, e funções padrões omitidas). Considerando isso, e o que foi apresentado até então nesse documento, podemos considerar que todos os requisitos do projeto foram atendidos, e que o projeto está coerente com nossa motivação inicial. O principal ponto de melhoria para esse projeto seria de melhor suporte para operações entre imagens de dimensões distintas, ou erros mais descritivos (próprios da linguagem) para operações inválidas desse tipo.

Além disso, há vários pontos de extensão atraentes, como uma forma de obter as dimensões de uma imagem, estruturas de controle condicional, e refinamento de estruturas de repetição. Esses pontos foram considerados ao longo do desenvenvolvimento do projeto,

mas desenvolvimento e teste de todas as operações inicialmente propostas foram dadas como prioritárias, tendo em vista nossa motivação inicial.

```
img = image in "img.png"
frame = image in "frame.png"

flip img vertically
frame = modify img * frame contrast by 3
print frame - img

p = "my/imgs/dir/"
for img in p {
    save (image in img) + frame as p+"/new/"+img
}
```

```
# after standard functions

img = Image.open("img.png").convert("RGBA")
frame = Image.open("frame.png").convert("RGBA")

img = img.transpose(Image.FLIP_TOP_BOTTOM)
frame = ImageEnhance.Contrast(_mult(img, frame)).enhance(3)
_sub(frame, img).show()

p = "my/imgs/dir/"
for img in (_f for _f in _listdir(p) if _isfile(p+_f)):
    _add(Image.open(img).convert("RGBA"), frame).save(((p + "/new/") + img))
```

A Uso

Para compilação do projeto, é necessário um compilador compatível com o padrão c++-17. Para execução dos códigos produzidos, é necessesário um interpretador Python com versão igual ou superior a 3.5, além de uma versão igual ou superior a 6.0 do pacote Pillow instalada. A compilação do projeto é feita através do comando `make`, no diretório raíz do projeto, o que produzirá um executável `bin/iml`.

Para executá-lo, então, faça `./bin/iml <in> <out>`, onde `<in>` é o arquivo de entrada, um programa na linguagem IML, e `<out>` onde será escrito o código resultante em Python, no caso de compilação bem-sucedida. No caso de erros, estes são reportados na saída padrão, e nenhum código será gerado.