Image Manipulation Language

Carlos Vieira*

April 26, 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 imagen (e.g. AppleScript ou VBScript com photoshop), além de necessitar de compreensão de características da linguagem irrelevantes para o processo de ediçãi, 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 2 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.

*vieiramecarlos@gmail.com

2 Sintaxe de List

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.

$$
\begin{aligned}
program &\rightarrow command* \\
command &\rightarrow stmt \mid img - expr \\
stmt &\rightarrow assignment \mid export \mid print \mid for - each \\
assignment &\rightarrow id \text{ `=' } expression \\
export &\rightarrow \text{`save' } expression \text{ `as' } expression \\
print &\rightarrow \text{`print' } expression \\
for - each &\rightarrow \text{`for' `all'}? \ id \text{ `in' } expression \text{`\{' } command* \text{ `\}'} \\
expression &\rightarrow arith - expr \\
import - expr &\rightarrow \text{`image in' } expression \\
img - expr &\rightarrow geom - expr \mid color - expr \\
geom - expr &\rightarrow rotate - expr \mid resize - expr \mid crop - expr \mid flip - expr \\
color - expr &\rightarrow modify - expr \\
rotate - expr &\rightarrow \text{`rotate' } expression \text{ `by' } expression \\
flip - expr &\rightarrow \text{`flip' } expression \ (\text{`vertically'} \mid \text{`horizontally'}) \\
resize - expr &\rightarrow \text{`resize' } expression \ (\text{`to'} \mid \text{`by'}) \ expression \\
crop - expr &\rightarrow \text{`crop' } expression \text{ `from' } expression \\
modify - expr &\rightarrow \text{`modify' } expressionenhancement \text{ `by' } expression \\
enhancement &\rightarrow \text{`sharpness'} \mid \text{`brightness'} \mid \text{`contrast'} \mid \text{`color'} \\
shape - expr &\rightarrow dimensions \mid section \\
dimensions &\rightarrow \text{`('}expression\text{`,'}expression\text{`)'} \\
section &\rightarrow \text{`('}expression\text{`,'}expression\text{`,'}expression\text{`,'}expression\text{`)'} \\
arith - expr &\rightarrow term \mid arith - expr \ (\text{`+'} \mid \text{`-'}) \ term \\
term &\rightarrow factor \mid term \ (\text{`*'} \mid \text{`/'}) \ factor \\
factor &\rightarrow atom \mid \text{`-' } factor \\
atom &\rightarrow primary \mid primary \text{ `(' } channel \text{ `)'} \\
channel &\rightarrow \text{`R'} \mid \text{`G'} \mid \text{`B'} \\
primary &\rightarrow id \mid \text{`(' } expression \text{ `)'} \mid scalar \mid import - expr \mid img - expr \mid shape - expr \\
scalar &\rightarrow float \mid integer \mid path \\
id &\rightarrow [a - zA - Z] + \\
integer &\rightarrow [0 - 9] + \\
float &\rightarrow integer(\text{`.'}integer)? \\
path &\rightarrow \text{`"'}[\hat{}\text{`"'}] * \text{`"'}
\end{aligned}
$$

3 Semântica de List

Quanto a semântica de um programa na linguagem List, enquanto a maioria dos pontos foram definidos previamente na descrição do projeto, muitos foram deixados como dependentes da implementação. De qualquer forma, descrevemos resumidamente suas características semânticas abaixo.

Primeiramente, um programa na linguagem List é composto de zero ou mais comandos sequenciais. Um comando pode ser de impressão, "`print` *expr*", que imprime o valor de *expr* na saída padrão; ou de atribuição, "*ID* `=` *expr*", que atribui o valor de *expr* à variável com nome *ID*, que pode ou não já ter sido atribuido outro valor.

Uma expressão, assim como uma variável já atribuída, possui um de três tipos: inteiro (valores não-negativos), booleano (`true` ou `false`), ou uma lista, cujos elementos por sua vez podem ter qualquer um desses três tipos. Há duas operações em List que podem compor expressões mais básicas em expressões mais complexas, `.` e `+`. Porém, essas operações funcionam de formas diferentes dependendo dos tipos de seus operandos, e não são definidas para todas as possíveis combinações de tipos.

A operação `+`, quando aplicada a dois inteiros, representa sua soma; e entre dois booleanos, sua disjunção (operador '*or*'). Já quando aplicada a duas listas l_1 e l_2, o resultado será uma lista do mesmo tamanho de l_1, onde cada elemento na posição i é o resultado de aplicar a operação `+` entre os elementos de l_1 e l_2 nessa posição. Caso l_2 seja menor que l_1, para cada i maior que o tamanho de l_2, o elemento na posição i da lista resultante terá o valor do elemento nessa mesma posição em l_1. Caso l_2 seja maior que l_1, os valores de l_2 em posições além do tamanho de l_1 são ignorados, já que a operação só é realizada para cada posição em l_1. Quando aplicado entre um booleano e uma lista, ou um inteiro e uma lista, o resultado será uma nova lista, onde cada elemento é o resultado de aplicar a operação `+` entre esse valor e cada elemento da lista original.

A operação `.`, quando aplicada entre inteiros, representa seu produto; e entre booleanos, sua conjunção (operador '*and*'). Já quando aplicada a duas lista, o resultado é a concatenação dessas. E, similarmente a `+`, se aplicada entre um booleano e uma lista ou entre um inteiro e uma lista, o resultado será uma nova lista, onde cada elemento é o resultado de aplicar a operação `.` entre esse valor e cada elemento da lista original. Note que ambas as operações são indefinidas no caso de um operando ser um booleano e outro um inteiro, o que leva a um erro de semântica estática em tempo de compilação nessa implementação. Além disso, ambas são operações comutativas, exceto quando ambos operandos são listas. 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).

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 na linguagem List. 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. São essas funções que permitem algumas das operações "recursivas" de List, como `+` entre listas, ou `.` entre um inteiro e uma lista que contém listas, por exemplo. 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).

- `_sum`: aplicar `+` a dois elementos de quaisquer tipos, só é chamada por outras funções (e.g. `_list_sum`);
- `_list_sum`: aplicar `+` a duas listas;
- `_scalar_list_sum`: aplicar `+` entre um booleano e uma lista, ou um inteiro e uma lista;
- `_scalar_sum`: aplicar `+` entre dois booleanos ou entre dois inteiros, só é chamada por outras funções (e.g. `_sum`);
- `_dot`: aplicar `.` a dois elementos de quaisquer tipos, só é chamada por outras funções (e.g. `_scalar_list_dot`);
- `_scalar_list_dot`: aplicar `.` entre um booleano e uma lista, ou um inteiro e uma lista;
- `_scalar_dot`: aplicar `.` entre dois booleanos ou entre dois inteiros, só é chamada por outras funções (e.g. `_dot`);

Note que certas funções só são chamadas por outras funções no sentido que não são chamadas diretamente do "*main*", já que o compilador pode chamar uma função mais específica diretamente nesses casos. Por exemplo, para `.` entre dois inteiros `a` e `b`, `_dot(a, b)`, assim como `_scalar_dot(a, b)`, produziriam o resultado desejado, mas se não há listas envolvidas podemos simplesmente traduzir essa operação como `a * b` em Python. Além disso, toda função padrão se inicia com um *underscore* (_), para evitar conflitos com variáveis definidas pelo usuário de List.

6 Resultados

Abaixo, podemos ver um exemplo de programa válido na linguagem List, e o programa equivalente gerado por essa implementação em Python (comentários adicionados posteriormente, e funções padrões omitidas). Outros exemplos estão disponíveis no diretório `examples/`, dentro do diretório raíz do projeto. Considerando isso, e o que foi apresentado até então nesse documento, todos os requisitos do projeto foram atendidos, além de verificação pelo compilador de erros de semântica estática. Um ponto em que a implementação poderia melhorar seria em indicar, no caso de operações indefinidas envolvendo um elemento de uma lista, qual elemento (por posição, ou posições no caso de listas aninhadas) causou o erro.

```
a = [0, 1, 2]
print a + 2
print 3 . a
print a . [3]

b = [4, [3, 2], [], 5]
print b + a
print [] . []
```

```
# after standard functions

a = [0, 1, 2]
print(_scalar_list_sum(2, a))    # gives [2, 3, 4]
print(_scalar_list_dot(3, a))    # gives [0, 3, 6]
print((a + [3]))                 # gives [0, 1, 2, 3]
b = [4, [3, 2], [], 5]
print(_list_sum(b, a))           # gives [4, [4, 3], [], 5]
print(([] + []))                 # gives []
```

A Uso

Para compilação do projeto, é necessário um compilador compatível com o padrão c++-17, e para execução dos códigos produzidos, um interpretador Python com versão igual ou superior a 3.6. A compilação do projeto é feita através do comando `make`, no diretório raíz do projeto, o que produzirá um executável `bin/compiler`.

Para executá-lo, então, faça `./bin/compiler <in> <out>`, onde `<in>` é o arquivo de entrada, um programa na linguagem List, 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.