Implementação de Linguagem de Domínio Específico

Linguagens Formais e Autômatos – Trabalho Final

Prof. Jefferson O. Andrade

Instituto Federal do Espírito Santo – Campus Serra

2019/1

Sumário

1 Objetivo

Para a execução deste trabalho o grupo deverá propor, definir, e implementar uma **Linguagem de Domínio Específico** (*Domain Specific Language* – DSL).

A escolha da DSL ficará à cargo do grupo, podendo ser destinada para qualquer propósito. Serão aceitas tanto linguagens originais propostas pelo grupo, como implementações de linguagens ou subconjuntos de linguagens já existentes.

2 Partes do Trabalho

O trabalho será dividido em 2 partes:

1. Relatório.
2. Implementação.

2.1 Relatório

O relatório deverá conter as seguintes seções:

Introdução Descrição sucinta da DSL escolhida, indicando se é uma DSL original ou se é uma implementação de uma DSL já existente. Deve ser apresentado o propósito da DSL.

Definição da DSL Deve apresentar uma descrição da sintaxe da DSL, este descrição deve conter **obrigatoriamente** a gramática que foi implementada em notação EBNF e os diagramas de sintaxe gerados pelo ANTLR. Também devem ser dados alguns exemplos de "código" da linguagem ilustrando as construções sintáticas.

Definição da AST Descrição sucinta da estrutura de dados criada para reapresentar os códigos da linguagem (*Abstract Syntax Tree* – AST). Esta seção deve conter o diagrama de classes e uma descrição de cada classe. Não é necessário descrever cada atributo e cada método. A descrição pode ser feita em alto nível com o essencial para a compreensão da implementação.

2.2 Implementação

O grupo poderá utilizar qualquer ferramente de geração de reconhecedores de linguagem que desejar, por exemplo: yacc, JavaCC, PyParsing, ANTLR, Parsec. Alternativamente, se o grupo desejar, podem fazer a implementação de um parser descendente recursivo "do zero". Em qualquer dos casos a escolha do grupo deverá ser descrita e justificada no relatório.[1]

A implementação do programa que processará os arquivos fontes da DSL proposta pelo grupo deve seguir um dentre dois fluxo de trabalho possíveis para o processamento da linguagem e geração das respectivas saídas. Caso o grupo utilize uma ferramenta que gere um analisador léxico, ou caso o grupo decida implementar um analisador léxico, deverá ser seguido o seguinte fluxo:

[1] A justificativa pode ser simplesmente algo como: "Analisamos ferramentas X, Y e Z e concluímos que seria mais fácil usar Y porque blah, blah, blah."

O arquivo fonte será lido pelo *lexer* e gerará uma *stream* de tokens. O *parser* lerá a stream de tokens e irá gerar uma estrutura de dados (*Abstract Syntax Tree* – AST) que irá representar as definições lidas do arquivo fonte. Em seguida o *walker* irá receber o AST e, à partir deste, gerar o arquivo de saída.

Caso a ferramenta utilizada não gere um analisador léxico separadamente, ou caso o grupo decida codificar um parser descendente recursivo que já integre as duas etapas (análise léxica e sintática) em um só código, o fluxo utilizado deve ser o seguinte:

Neste fluxo, o arquivo fonte será lido pelo *parser* que também fará a análise léxica, sem o passo intermediário de geração dos *tokens*, e irá gerar a AST. Em seguida o *walker* irá receber o AST e gerar o arquivo de saída.

3 Requisitos da DSL

A DSL proposta pode ser voltada para qualquer aplicação que o grupo considere interessante, entretanto ela terá que atender a alguns critérios mínimos:

- Deve haver ao menos um mecanismo de nomeação, i.e., uma forma de atribuir nomes a valores. Por exemplo: variáveis, ou objetos.
- Deve haver ao menos um mecanismo de abstração, i.e., uma forma de dar nomes a conceitos complexos e utilizar esses conceitos posteriormente através destes nomes. Por exemplo: classes, procedimentos, ou funções.
- Deve haver ao menos uma construção sintática para seleção. Mais detalhes na Subseção 3.1.
- Deve haver ao menos uma construção sintática para repetição. Mais detalhes na Subseção 3.2.

3.1 Estrutura de Seleção

Vários tipos de estruturas de seleção e/ou decisão podem ser escolhidos. Os mais comuns são os do tipo `if... then... else...`, e suas variações que incluem construções como `elsif` ou `elif`. Há também construções como o `cond` de *Common Lisp* e *Scheme*, mostrado abaixo:

```
(cond
  ((> x 0) (run-simulation x))
  ((eq x 0) (show-warning))
  ((< x 0) (panic-protocol x)))
```

Uma outra possibilidade é que a DSL implemente seleção de modo declarativo, como fazem, por exemplo, *Haskell* e *Prolog*. Nessas linguagens podemos ter, respectivamente, funções e regras declarados de acordo com casos, sem a necessidade de uma sintaxe explícita para seleção.[2] Por exemplo, considere a definição da função `sayMe` em Haskell dada abaixo:

```
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Nesta definição, de acordo com o valor inteiro passado como argumento, a função retorna uma string diferente. Entretanto, não há a necessidade de testar explicitamente o argumento da função com um `if`, pois a prórpia definição da função já faz o "teste". Na verdade, é possível definir uma função em Haskell que implemente o `if-then-else` típico de outras linguagens sem necessidade de nenhum elemento adicional, como mostrado abaixo:[3]

```
if True x y = x
if False x y = y
```

Já em Prolog, a definição do predicado análogo à função `sayMe` seria o seguinte:

```
sayMe(1, "One!").
sayMe(2, "Two!").
sayMe(3, "Three!").
sayMe(4, "Four!").
sayMe(5, "Five!").
sayMe(X, "Not between 1 and 5").
```

Note que também não há a necessidade de uma estrutura explícita de seleção pois a seleção é "declarada" na definição dos *fatos*.

A DSL implementada pelo grupo pode utilizar tanto seleção *explícita* quanto seleção *implícita* (no estilo de Haskell e Prolog). Ambas as forma serão aceitas.

3.2 Estrutura de Repetição

Assim como as estruturas de seleção, as estruturas de repetição variam bastante de uma linguagem para outra. Provavelmente a mais comum seja a estrutura do tipo `while-do`, mas também é bastante comum encontrar estruturas do tipo `for` e `repeat-until` ou

[2] Embora tanto em Haskell quanto em Prolog existam construções explícitas de seleção.
[3] Isso funciona porque Haskell utiliza *lazy evaluation*.

`loop-while`. No caso das estruturas de repetição do tipo `for` há ainda a questão de realizar um for sobre um contador ou do tipo *for each*, ou seja, para cada elemento de uma coleção (uma lista ou conjunto).

A título de exemplo, abaixo são mostradas duas estruturas de repetição da linguagem *Common Lisp*, a forma `dolist` que executa um comando para cada valor de uma lista, e a forma `dotimes` que executa um comando para cada valor ente 0 e n, onde n é informado.

```
(dolist (x '(um dois tres)) (print x))
(dotimes (x 10) (print x))
```

Na primeira expressão do exemplo acima (`dolist`), serão impressos os valores "um", "dois" e "tres", em linhas individuais. Na segunda expressão serão impressos em linhas distintas os valores de 0 a 9.

Entretanto, é possível ter repetições sem ter uma sintaxe específica para isso. O mecanismo de recursão permite que tenhamos repetições "de graça". Se uma linguagem permite definições recursivas e também permite algum tipo de seleção, então é possível ter repetições.

Por exemplo, considere a definição da função fatorial, dada em Haskell, abaixo:

```
fact 0 = 1
fact n = n * fat (n - 1)
```

Note que, por causa da chamada recursiva na segunda linha da definição ocorrerá repetição, mesmo que não haja uma estrutura sintática específica para repetição.

Como um segundo exemplo do mesmo conceito, considere o predicado análogo à função `fact`, definido em Prolog abaixo:

```
fact(0, 1).
fact(N, F) :- M is N - 1, fact(M, G), F is N * G.
```

Em ambos os casos, a chamada recursiva faz com que a computação desejada seja repetida e a repetição para quando se atinge o caso base, que é definido na primeira linha das duas definições. Não há a necessidade de uma estrutura explícita para repetição.

A DSL implementada pelo grupo pode utilizar tanto repetição *explícita* quanto repetição *implícita* via recursão. Ambós as forma serão aceitas.

4 Instruções

- O trabalho poderá ser executado por grupos de até 3 integrantes. Os estudantes podem debater ideias entre si, mas a elaboração do código deve ser um trabalho original de cada grupo.

- O trabalho deve ser entregue **exclusivamente** através do sistema AVA do Ifes na forma de um arquivo compactado em formato ZIP ou 7z na tarefa corresponde indicada na página da disciplina.

  ATENÇÃO: Apenas os formatos de compactação ZIP e 7z serão aceitos. Trabalhos entregue em outros formado de compactação tais como RAR ou LHA serão **anulados**.

- O arquivo compactado entregue deve conter um diretório (pasta) com todo o código fonte desenvolvido para o trabalho. A estrutura do deve ser a seguinte:

  ```
  trab-final
  ├── Readme.md
  ├── relatório em pdf
  ├── source
  │   ├── build file
  │   ├── código fonte do
  │   │   trabalho
  │   └── ...
  └── testes
      ├── arquivos de teste
      │   da DSL
      └── ...
  ```

  O arquivo `Readme.md` é um arquivo em formato Markdown contendo ao menos: (a) o nome dos autores do trabalho; (b) a descrição da linguagem e ambiente de programação utilizados na elaboração do trabalho; (c) a descrição geral dos arquivos contidos no trabalho; (d) a forma de construção (*build*), se for necessário; (e) instruções de como executar o programa.

  O diretório `source` deve conter todo o código fonte propriamente dito. A organização interna deste diretório é livre, entretanto, deve haver algum arquivo de *build*, tal como `Makefile`, `build.xml`, script de shell, etc. de acordo com a ferramenta de *build* usada pelo grupo. O arquivo de *build* e como ativá-lo deve estar descrito no arquivo *Readme.md*.[4]

- Os critérios usados para atribuição de nota serão:

  Gramática (10 pontos) Qualidade da gramática da DSL implementada, envolve tanto a descrição EBNF da gramática quanto o grau de sofisticação da gramática escolhida.

  AST (15 pontos) Referente ao nível de complexidade, qualidade no projeto e na implementação das estruturas de dados utilizadas para implementar o *Abstract Syntax Tree* gerado pelo parser.

[4]Se o trabalho for feito em uma linguagem que não necessita de *build*, esta parte não é necessária.

Exemplos (10 pontos) Qualidade dos exemplos apresentados no relatório e para teste do programa. Os exemplos foram explicados? Eles "exercitam" todas as características da linguagem? Eles são criativos?

Qualidade do Relatório (15 pontos) Referente à qualidade geral do relatório. Se apresenta todos os pontos que foram pedidos. Se é de fácil leitura. Se está bem redigido e organizado.

Nomeação (10 pontos) Mecanismo para atribuição de nomes a valores e/ou objetos da linguagem.

Definição (10 pontos) Mecanismo para definição de novas construções na linguagem. Tais como procedimentos, funções, classes, objetos, etc.

Seleção (10 pontos) Mecanismo para controle de fluxo de execução da linguagem que realize seleções. Tais como `if-then-else`, `switch`, `case`, `cond`, etc.

Repetição (10 pontos) Mecanismo para controle de fluxo de execução da linguagem que realize repetições. Tais como `while`, `repeat`, `loop`, etc.

Qualidade do Código (10 pontos) Refere-se à qualidade geral do código: endentação; nomes de funções, métodos e variáveis; estrutura geral do código; atribuição de responsabilidades aos elementos do código; etc.

Além destes critérios também poderão ser computadas penalidades para falhas consideradas muito básicas ou não previstas nos critérios acima. Não foi definido valor máximo para as penalidades.