Instituto Tecnológico de Aeronáutica

Curso de Engenharia da Computação

Disciplina CES-41: Compiladores

Relatório do Laboratório 1

Vitor Pimenta dos Reis Arruda

Professor Fábio Carneiro Mokarzel

São José dos Campos

04/03/2019

Introdução

Este trabalho intenciona a escrita e o teste de analisadores léxicos para linguagens artificiais de computação. Resolvem-se vários problemas retirados de [1], apresentando seu enunciado original, a solução proposta e os testes que a validam.

Para o desenvolvimento, usou-se a linguagem de programação *javascript* com o analisador léxico **Moo** [2]. O modo de uso básico dessa biblioteca é explicado junto aos problemas resolvidos neste trabalho. Nota-se que os problemas pedem explicitamente o uso da ferramenta **Flex**, que não foi utilizada, como já explicado.

Problema 1

Enunciado

Escrever um programa em ***Flex*** reconhecedor de cadeias sobre o alfabeto {0, 1}, tais que o número de dígitos **0** seja par ou o número de dígitos 1 seja par.

Solução proposta

Considere a Codificação 1:

```
1   const lexer1 = moo.compile({
2       valido: /(?:(?:0*10*10*)+|(?:1*01*01*)+|0+|1+)(?![^])/,
3       invalido: { match: /[^]+/, lineBreaks: true }
4   })
```

Codificação 1: Sintaxe Moo para resolver o problema 1

A função **moo.compile()** toma como argumento um objeto (delimitado por **{** na linha 1 e **}** na linha 4). Na linguagem **javascript**, um objeto é um dicionário, isto é, um conjunto de pares chave→valor. No exemplo, as chaves são **valido** e **invalido**, e o valor de cada uma é o que segue o sinal de dois-pontos (**:**).

Para a chave **valido**, forneceu-se uma expressão regular (delimitada pelos sinais **/** na linha 2). Para a biblioteca **Moo**, isso significa que, ao realizar posterior análise léxica de uma cadeia de caracteres, qualquer sequência aceita pela expressão regular será retornada como um **token**, que é um objeto cujas chaves informam sobre a localização em que foi descoberto dentro da sequência de entrada (como linha e coluna), o tipo do **token** (que seria **valido** no caso do exemplo), seu texto (cópia literal da parte capturada da cadeia de entrada) e outros.

Para ilustrar, considere a Figura 1, obtida ao acionar **lexer1** sobre a cadeia **0011000** (que, nota-se, tem quantidade par de dígitos 1, como será discutido adiante).

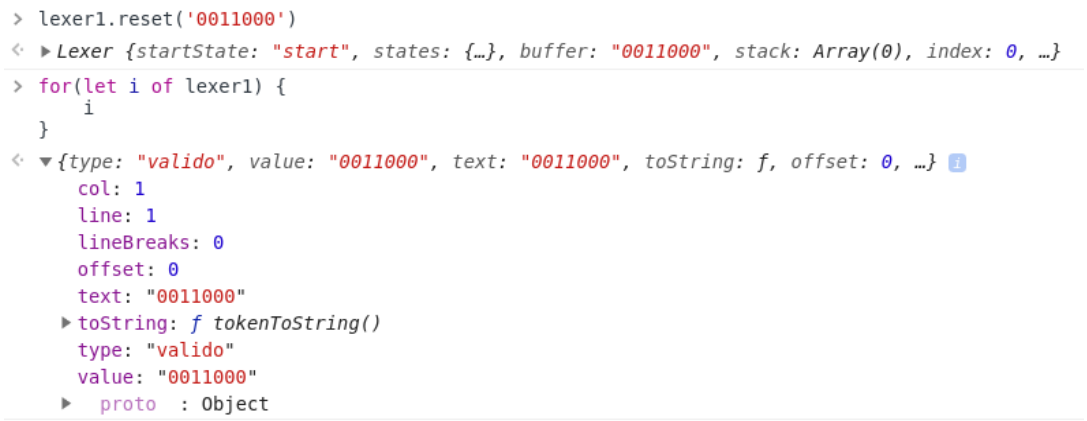

Figura 1: Saída de **lexer1** *sobre cadeia* **0011000**, *evidenciando a estrutura do objeto* **token** *retornado*

Quanto à outra chave (**invalido**) do objeto fornecido ao **Moo**, ela caracteriza outro tipo de **token**. Para preenchê-la, não se forneceu diretamente uma expressão regular (como no caso anterior), mas outro objeto contendo chaves **match** e **lineBreaks**. O efeito, porém, é o mesmo do caso anterior, e a necessidade de se fornecer um objeto em vez de uma expressão regular é meramente técnica: a biblioteca **Moo** exige que se explicite quando a expressão regular fornecida puder aceitar caracteres de mudança de linha (**\n**), daí a necessidade de especificar **lineBreaks:true**. Nota-se, também, que ***Moo*** não aceita expressões regulares que aceitem a cadeia vazia.

Por último, antes de discorrer sobre a lógica do problema em si, dois detalhes de sintaxe das expressões regulares: Os grupos envolvidos pelos sinais **(?:** e **)** são chamados **grupos não capturantes** e possuem o mesmo efeito da parentisação usual (isto é, sem ***?:***) no que concerne à aceitação de cadeias, enquanto os grupos envolvidos por ***(?!*** e ***)*** são chamados **lookahead negativos**, e servem para especificar que a expressão regular anterior a eles só deve aceitar certa cadeia se esta **não for** sucedida pelo que estiver dentro do **lookahead negativo**. Em relação a outros aspectos do formalismo de expressões regulares, por serem similares aos do **Flex**, são assumidos como conhecidos pelo leitor.

Finalmente, explica-se o motivo de a expressão regular fornecida para **valido** resolver ao problema proposto:

- A porção **(?:0*10*10*)+** identifica cadeias com quantidade par *não-nula* de dígitos 1
- A porção **(?:1*01*01*)+** identifica cadeias com quantidade par *não-nula* de dígitos 0
- A porção **0+** identifica cadeias sem senhum dígito 1 (afinal, 0 é par)
- A porção **1+** identifica cadeias sem senhum dígito 0 (afinal, 0 é par)
- O **lookahead negativo (?![^])** determina que, depois da cadeia aceita por qualquer dos pontos anteriores, não deve aparecer nenhum outro caractere (**[^]** é uma expressão regular que aceita qualquer caractere único, inclusive **\n**).

Reconhece-se que, a rigor, a cadeia vazia deveria ser aceita pela solução proposta, pois é solução válida do problema. Porém, como **Moo** não permite o uso de expressões regulares que aceitem a cadeia vazia, não é possível corrigir esse erro unicamente com a biblioteca. Portanto, para que a solução seja completamente correta (avaliando corretamente a cadeia vazia), é necessário fazer essa verificação “por fora” da biblioteca, como na Codificação 2:

```
1   export function predicate1(inputstring: string) {
2       if (inputstring === '') {
3           return true
4       }
5       lexer1.reset(inputstring)
6       return lexer1.next().type === 'valido'
7   }
```

Codificação 2: Função-solução ao problema 1. Dada uma entrada **inputstring**, *ela retorna o valor lógico* **true** *se, e somente se, a entrada for solução do problema*

Testes realizados

Para verificação de **predicate1**, geraram-se as 10000 cadeias mais curtas contendo os caracteres **0**, **1**, **a** e **\n** e aplicou-se cada cadeia a **predicate1**, verificando se os resultados retornados seriam corretos. A Figura 2 ilustra alguns testes manuais e a Figura 3 invoca a rotina de teste automatizada (**test1**), que retorna uma lista vazia (**[]**), indicando que não houve erro em nenhuma das 10000 cadeias testadas (a rotina de teste foi especificada para retornar uma lista contendo as cadeias em que a função proposta errasse).

```
> predicate1('')
<- true
> predicate1('0')
<- true
> predicate1('01')
<- false
> predicate1('010')
<- true
> predicate1('0101')
<- true
> predicate1('01010')
<- true
>
```

Figura 2: Alguns testes manuais da função-solução ao problema 1

```
> test1()
<- ▶ []
>
```

Figura 3: Teste automatizado da função-solução ao problema 1, bem sucedido

Problema 2

Enunciado

Escrever um programa em ***Flex*** reconhecedor de cadeias sobre o alfabeto {0, 1}, tais que o número de dígitos **0** seja ímpar e o número de dígitos **1** seja ímpar.

Solução proposta

Observa-se que as cadeias a serem aceitas neste problema são justamente as que devem ser rejeitadas no problema anterior, considerando somente o alfabeto **{0, 1}.** Porém, considerando que não existem só os caracteres **0** e **1**, essa complementaridade não é mais válida (isto é, há cadeias que devem ser rejeitadas em ambos os problemas, a saber, aquelas em que algum caractere diferente de **{0, 1}** aparece).

Assim, optou-se por uma estratégia diferente mas similar: Identificaram-se o **tokens** a serem rejeitados (chave **invalido** da Codificação 3):

```
1  const lexer2 = moo.compile({
2     invalido: {
3        match: /(?:(?:0*10*10*)+|(?:1*01*01*)+|0+|1+|(?:[01]*[^01][01]*)+)(?![^])/,
4        lineBreaks: true
5     },
6     valido: /.+/
7  })
```

Codificação 3: Sintaxe **Moo** *para resolver o problema 2*

Explica-se a expressão regular utilizada:

- A porção **(?:0*10*10*)+** identifica cadeias com quantidade par *não-nula* de dígitos 1
- A porção **(?:1*01*01*)+** identifica cadeias com quantidade par *não-nula* de dígitos 0
- A porção **0+** identifica cadeias com quantidade par *nula* de dígitos 1
- A porção **1+** identifica cadeias com quantidade par *nula* de dígitos 0
- A porção **(?:[01]*[^01][01]*)+** identifica cadeias que contenham qualquer caractere diferente de **0** e **1** (sejam lá quantos forem esses caracteres).

Assim, o analisador léxico gerado atribuirá o rótulo **invalido** a qualquer **token** que extrapole o alfabeto **{0, 1}** ou que não tenha quantidades ímpares tanto de dígitos **0** quanto de dígitos **1**. Similarmente ao problema 1, foi gerada uma função-solução **predicate2**, que aceita uma cadeia de entrada e retorna o valor lógico **true** se, e somente se, a tal cadeia for solução do problema 2.

Testes realizados

Similarmente ao problema 1, para verificação de **predicate2**, geraram-se as 10000 cadeias mais curtas contendo os caracteres **0**, **1**, **a** e **\n** e aplicou-se cada cadeia a **predicate2**, verificando se os resultados retornados seriam corretos. A Figura 4 ilustra alguns testes manuais e a Figura 5 invoca a rotina de teste automatizada (**test2**), que retorna uma lista vazia (**[]**), indicando que não houve erro em nenhuma das 10000 cadeias testadas (a rotina de teste foi especificada para retornar uma lista contendo as cadeias em que a função proposta errasse).

Figura 4: Alguns testes manuais da função-solução ao problema 2

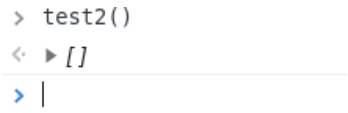

Figura 5: Teste automatizado da função-solução ao problema 2, bem sucedido

Problema 3

Enunciado

Escrever um programa em ***Flex*** reconhecedor de cadeias sobre o alfabeto {**0**, **1**, **2**}, tais que o número de dígitos **2** seja divisível por 5 (Obs: zero é divisível por 5).

Solução proposta

Em resumo, as soluções são cadeias que ou não contenham nenhum dígito **2**, ou sejam repetições de subcadeias as quais, individualmente, contenham 5 vezes o dígito **2**.

Como visto na Codificação 4, o primeiro caso é previsto pela subexpressão **[01]+** , enquanto o segundo é resolvido por **(?:[01]*2[01]*2[01]*2[01]*2[01]*2[01]*)+**.

```
1  const lexer3 = moo.compile({
2      valido: {
3          match: /(?:[01]+|(?:[01]*2[01]*2[01]*2[01]*2[01]*2[01]*)+)(?![^])/,
4          lineBreaks: true
5      },
6      invalido: { match: /[^]+/, lineBreaks: true }
7  })
```

Codificação 4: Sintaxe Moo para resolver o problema 3

Similarmente aos problemas anteriores, foi gerada uma função-solução **predicate3**, que aceita uma cadeia de entrada e retorna o valor lógico **true** se, e somente se, a tal cadeia for solução do problema 3.

Testes realizados

Similarmente aos problemas anteriores, para verificação de **predicate3**, geraram-se as 100000 cadeias mais curtas contendo os caracteres **0**, **1**, **a** e **\n** e aplicou-se cada cadeia a **predicate3**, verificando se os resultados retornados seriam corretos. A Figura 6 ilustra alguns testes manuais e a Figura 7 invoca a rotina de teste automatizada (**test3**), que retorna uma lista vazia (**[]**), indicando que não houve erro em nenhuma das 100000 cadeias testadas (a rotina de teste foi especificada para retornar uma lista contendo as cadeias em que a função proposta errasse).

```
> predicate3('')
< true
> predicate3('00')
< true
> predicate3('002')
< false
> predicate3('0022')
< false
> predicate3('220022')
< false
> predicate3('2202022')
< true
> predicate3('22020221')
< true
> predicate3('2202022122002200120')
< true
> predicate3('220202212200220010')
< false
>
```

Figura 6: Alguns testes manuais da função-solução ao problema 3

```
> test3()
< ▶ []
>
```

Figura 7: Teste automatizado da função-solução ao problema 3, bem sucedido

Problema 4

Enunciado

Escrever um programa em ***Flex*** reconhecedor de cadeias sobre o alfabeto **{0, 1}**, com no mínimo cinco caracteres, tais que qualquer bloco de cinco caracteres consecutivos contenha no mínimo três dígitos **1**.

Solução proposta

Em estratégia similar à usada no problema 2, optou-se por escrever uma expressão regular que capturasse os **tokens inválidos**. Eles estão entre três classificações:

- Aqueles cujos caracteres não são exclusivamente **0** ou **1**, identificados por **(?:[01]*[^01][01]*)+**
- Aqueles que têm 4 caracteres ou menos, identificados por **[01]{1,4}**
- Aqueles que, apesar de serem perfeitos em relação aos dois pontos anteriores, possuem subsequências de 5 caracteres consecutivos com **menos** do que 3 dígitos **1**. Como há quantidade finita dessas subsequências, basta listá-las. Assim, são identificados por **[01]*(?:00000|00001|00010|00100|01000|10000|11000|10100|10010|10001|01100|01010|01001|00110|00101|00011)[01]***

Portanto se sugere a Codificação 5 como solução:

```
1 const lexer4 = moo.compile({
2     invalido: {
3         match: /(?:(?:[01]*[^01][01]*)+|[01]{1,4}|[01]*(?:00000|00001|00010|00100|01000|10000|
  11000|10100|10010|10001|01100|01010|01001|00110|00101|00011)[01]*)(?![^])/,
4         lineBreaks: true
5     },
6     valido: /.+/
7 })
```

Codificação 5: Sintaxe Moo para resolver o problema 4

Similarmente aos problemas anteriores, foi gerada uma função-solução **predicate4**, que aceita uma cadeia de entrada e retorna o valor lógico **true** se, e somente se, a tal cadeia for solução do problema 4.

Testes realizados

Similarmente aos problemas anteriores, para verificação de **predicate4**, geraram-se as 100000 cadeias mais curtas contendo os caracteres **0**, **1**, **a** e **\n** e aplicou-se cada cadeia a **predicate4**, verificando se os resultados retornados seriam corretos. A Figura 6 ilustra alguns testes manuais e a Figura 7 invoca a rotina de teste automatizada (**test4**), que retorna uma lista vazia (**[]**), indicando que não houve erro em nenhuma das 100000 cadeias testadas (a rotina de teste foi especificada para retornar uma lista contendo as cadeias em que a função proposta errasse).

```
> predicate4('')
< false
> predicate4('a')
< false
> predicate4('\n')
< false
> predicate4('100')
< false
> predicate4('1000')
< false
> predicate4('10000')
< false
> predicate4('1000011')
< false
> predicate4('10000111111111')
< false
> predicate4('10100111111111')
< false
> predicate4('10110111111111')
< true
> predicate4('101101110001111')
< false
> predicate4('10110111001111')
< true
> predicate4('101101110011110000')
< false
> predicate4('101101110011110001')
< true
> predicate4('1111')
< false
>
```

Figura 8: Alguns testes manuais da função-solução ao problema 4

```
> test4()
< ▶ []
>
```

Figura 9: Teste automatizado da função-solução ao problema 4, bem sucedido

Problema 5

Enunciado

Escrever um programa em ***Flex*** para fazer análise léxica de uma mini-linguagem que contenha os seguintes átomos: identificadores (ID), constantes inteiras (CTINT), constantes reais (CTREAL), operadores aditivos (OPAD), operadores multiplicativos (OPMULT), abre e fecha-parentesis (ABPAR e FPAR), abre e fecha-chaves (ABCHAV e FCHAV), sinal de atribuição (ATRIB), vírgula e ponto-e-vírgula (VIRG e PVIRG) e ainda as palavras reservadas **program**, **var**, **int** e **real**.

- Cada átomo tem seu tipo, e seu atributo depende desse tipo.

- Abre e fecha-parentesis, abre e fecha-chaves, sinal de atribuição, vírgula e ponto-e-vírgula são átomos que não devem ter atributos; seus tipos são ABPAR, FPAR, ABCHAV, FCHAV, ATRIB, VIRG e PVIRG, respectivamente.

- As palavras reservadas **program**, **var**, **int** e **real** também não devem ter atributos; seus tipos são, respectivamente, PROGRAM, VAR, INT e REAL.

- Identificadores têm como atributo a cadeia de seus caracteres; seu tipo é ID e sua sintaxe é

  letra (letra | dígito)*

- Constantes inteiras têm como atributo o seu valor inteiro; seu tipo é CTINT.

- Constantes reais têm como atributo o seu valor real; seu tipo é CTREAL e sua sintaxe é

  (dígito)+ . (dígito)*

- Tabela dos tipos e atributos dos operadores:

Átomo	Tipo	Atributo	Átomo	Tipo	Atributo
+	OPAD	MAIS	*	OPMULT	VEZES
-	OPAD	MENOS	/	OPMULT	DIV

- Tabela dos tipos dos átomos sem atributos:

Átomo	Tipo	Átomo	Tipo
program	PROGRAM	**{**	ABCHAV
var	VAR	**}**	FCHAV
int	INT	**=**	ATRIB
real	REAL	**,**	VIRG
(	ABPAR	**;**	PVIRG
)	FPAR		

- A função **main** do programa em **Flex** deve produzir uma saída autoexplicativa. Como sugestão, ela poderá mostrar os caracteres dos átomos do programa analisado e mais os tipos e os atributos desses átomos, tal como nos programas 1.8 e 1.9 dos slides de aulas.

Solução proposta

Como nada foi dito em relação ao tratamento de espaços em branco e **\n** (coletivamente, **whitespace**) nem sobre caracteres inválidos, adotaram-se os **tokens WHITESPACE** para os primeiros e **INVALIDO** para os segundos.

Logo a solução do problema é simplesmente a Codificação 6:

```
1    const lexer5 = moo.compile({
2      ID: {
3        match: /[a-zA-Z](?:[a-zA-Z]|[0-9])*/,
4        type: moo.keywords({
```

```
5             PROGRAM: 'program',
6             VAR: 'var',
7             INT: 'int',
8             REAL: 'real'
9           })
10      },
11      CTREAL: {
12          match: /[0-9]+\.[0-9]*/,
13          value: text => (parseFloat(text) as unknown) as string
14      },
15      CTINT: {
16          match: /[0-9]+/,
17          value: text => (parseInt(text, 10) as unknown) as string
18      },
19      add_sub: {
20          match: /-|\+/,
21          type: _ => 'OPAD',
22          value: text => (text === '+' ? 'MAIS' : 'MENOS')
23      },
24      mult_div: {
25          match: /\*|\//,
26          type: _ => 'OPMULT',
27          value: text => (text === '*' ? 'VEZES' : 'DIV')
28      },
29      ABPAR: {
30          match: /\(/,
31          value: empty_string
32      },
33      FPAR: {
34          match: /\)/,
35          value: empty_string
36      },
37      ABCHAV: {
38          match: /\{/,
39          value: empty_string
40      },
41      FCHAV: {
42          match: /\}/,
43          value: empty_string
44      },
45      ATRIB: {
46          match: /=/,
47          value: empty_string
48      },
49      VIRG: {
50          match: /,/,
51          value: empty_string
52      },
53      PVIRG: {
54          match: /;/,
55          value: empty_string
56      },
```

```
57      WHITESPACE: {
58          match: /\s+/,
59          lineBreaks: true,
60          value: empty_string
61      },
62      INVALIDO: /./
63  })
```

Codificação 6: Sintaxe Moo para resolver o problema 5

Entretanto, para traduzir as chaves contidas nos objetos **token** da biblioteca **Moo** para as chaves desejadas no enunciado do problema 5 (quais sejam, Tipo e Atributo), foi necessário código adicional de tradução (Codificação 7). Notar linhas 10-12, que empreendem a tradução

```
1   function lex5(inputstring: string) {
2       const tokens = []
3       lexer5.reset(inputstring)
4       let token
5       while ((token = lexer5.next()) !== undefined) {
6           if (token.type === 'WHITESPACE') {
7               continue
8           }
9           tokens.push({
10              Texto: token.text,
11              Tipo: token.type,
12              Atributo: token.value
13          })
14      }
15      return tokens
16  }
```

Codificação 7: Tradução dos tokens do formato Moo para o formato pedido no enunciado do problema 5

Para uniformizar a nomenclatura de todos os problemas resolvidos, criou-se a função-solução **test5**, que não passa de uma renomeação de **lex5** (apresentada na Codificação 7). Ambas aceitam uma cadeia de entrada e realizam a análise da tal cadeia, retornando uma lista com os **tokens** identificados (na qual os **WHITESPACE** já foram devidamente ignorados).

Testes realizados

Como não se percebeu nenhuma lógica de construção de cadeias que permitisse automatizar o teste (como a lógica “das cadeias mais curtas para as mais longas” aplicada aos problemas anteriores), restou a alternativa de construir uma sentença longa e observar se é corretamente analisada pela solução proposta, como na Figura 10:

Figura 10: Teste manual da função-solução ao problema 5

Experimentação do código

Para testar o código usado, basta abrir o arquivo **index.html** fornecido junto com este laboratório em um navegador e seguir as instruções da página.

Referências

[1] Mokarzel, F. C. **1º Laboratório de CES-41/2019**. Roteiro do laboratório 1 da disciplina CES-41 do Instituto Tecnológico de Aeronáutica, 2019.

[2] Radvan, T. **Moo**. Analisador léxico escrito em *javascript*. Repositório em: <https://github.com/no-context/moo>. Acesso em 04/03/2019.