Aula 003 - Análise Léxica

Expressões Regulares

Prof. Rogério Aparecido Gonçalves
rogerioag@utfpr.edu.br

Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão - Paraná - Brasil

Ciência da Computação

BCC36B - Compiladores

Agenda

1. Processo de Varredura
2. Linguagens Regulares
3. Próximas Aulas

Processo de Varredura

Introdução

- Vimos que as formas de reconhecimento da marcas: **identificação de padrões**
  - **expressões regulares**
  - autômatos finitos (máquinas de estados finitos)

Linguagens Regulares

Linguagens Regulares

- Uma marca é um conjunto de *strings*
- Um outro nome para conjunto de strings é *linguagem*
- Geralmente os conjuntos de *strings* que caracterizam as marcas de linguagens de programação são linguagens regulares
- Em linguagens formais, uma linguagem regular é qualquer conjunto de strings que pode ser expresso usando uma **expressão regular**
- Logo, o fato dos tipos dos marcadores serem linguagens regulares dá uma notação conveniente para especificarmos como classificar os marcadores!

Expressões Regulares

- Assim como uma expressão aritmética denota um número (por exemplo, 2+3*4 denota o número 14), uma expressão regular denota uma linguagem regular
  - Por exemplo, a0+ denota a linguagem:
    - `{"a0", "a00", "a000", ... }`
- Vamos explorar expressões regulares usando uma função `findAll` que existe na maioria das linguagens de programação, que recebe uma expressão regular e uma string e retorna todas as ocorrências daquela expressão regular na string
  - Ex: `findAll("a0+", "a0 fooa000bar a005") => ["a0", "a000", "a00"]`

UTFPR

Caracteres e classes

- Caracteres e classes de caracteres são o tipo mais simples de expressão regular
- Denotam conjuntos de cadeias de um único caractere
- A expressão `a` denota o conjunto `{"a"}`, a expressão "x" o conjunto `{"x"}`
- A expressão `.` é especial e denota o conjunto alfabeto (conjunto de todos os caracteres)
- Uma classe `[abx]` denota o conjunto `{"a", "b", "x"}`
- Uma classe `[ab-fx]` denota {"a", "b", "c", "d", "e", "f", "x"}
- Uma classe `[^ab-fx]` denota o conjunto complemento da classe `[ab-fx]` em relação ao alfabeto

Concatenação e justaposição

- A concatenação ou justaposição de expressões regulares denota um conjunto com cadeias de vários caracteres, onde cada caractere da cadeia vem de uma das expressões concatenadas
- `[a-z][0-9]` denota o conjunto {"a0", "a1", ..., "a9", "b0", ..., "b9", ..., "z9"}
- `while` denota o conjunto { "while" }
- `[wW][hH][iI][lL][eE]` denota o conjunto { "while", "While", "wHile", "WHile", ... }
- `...` denota o conjunto de todas as cadeias de três caracteres (incluindo espaços!)

Repetição

- O operador + denota a repetição de um caractere ou classe de caracteres
  - `[a-z]+` denota o conjunto `{"a", "aa", "aaa", ..., "b", "bb", ..., "aba", ... }`, ou seja, cadeias formadas de caracteres entre a e z
  - `[a-z][0-9]+` denota o conjunto {“a0”, “a123”, “d25”, ... }‘, ou seja, cadeias formadas por um caractere de *a* à *z* seguidas por um ou mais dígitos
- O operador * é uma repetição que permite zero caracteres ao invés de ao menos 1
  - `[a-z][0-9]*` denota o conjunto acima, mais o conjunto `{"a", "b", ... "z"}`
  - `\"[^\"]*\"` denota o conjunto de cadeias de quaisquer caracteres entre aspas duplas, exceto as próprias aspas duplas, e inclui a cadeia `\"\"`

União e opcional

- Uma barra | em uma expressão regular denota a união dos conjuntos das expressões à esquerda e à direita da barra
  - `[a-zA-Z_][a-zA-Z0-9_]*|[0-9]+` é a união do conjunto denotado por `[a-zA-Z_][a-zA-Z0-9_]*` com o conjunto denotado por `[0-9]*`
- O operador ? denota o conjunto da expressão que ele modifica, mais a cadeia vazia
  - `[0-9]+([.][0-9]+)?` denota o conjunto de todos as sequências de dígitos, mais o conjunto das sequências de dígitos seguidas por um ponto e outra sequência de dígitos

UTFPR

Precedência

- A precedência dos operadores em uma expressão regular, da menor para a maior, é:
  1. |
  2. concatenação
  3. +, * e ?
- Naturalmente, podemos usar parênteses para mudar a precedência quando conveniente
- Na prática, é possível escrever uma especificação léxica sem precisar |, ( ) e ?, usando múltiplas regras para a mesma classe de token, e a especificação pode ficar mais legível assim

Especificação Léxica

- A *especificação léxica* de uma linguagem é uma sequência de regras, onde cada regra é composta de uma expressão regular e um tipo de *token*
- Uma regra diz que se os próximos caracteres presentes na entrada pertencerem ao conjunto denotado pela sua expressão regular, então o próximo *token* da entrada pertence ao seu tipo
- Para a linguagem de comandos simples, onde os *tokens* são numerais inteiros, identificadores, `+`, `-`, `(`, `)`, `=`, `;`, `print`, uma possível especificação léxica é dada no slide seguinte

Lista de comandos simples

ER	Token
`[0-9]+`	NUM
`[a-zA-Z]+`	ID
`\+`	"+"
`\-`	"-"
`\(`	"("
`\)`	")"
`=`	"="
`;`	";"
`print`	PRINT

Um fragmento da linguagem de programação Java

ER	Token	ER	Token
`&&`	E_LOGICO	`else`	ELSE
`||`	OU_LOGICO	`[a-zA-Z]`	ID
`\+`	'+'	`[a-zA-Z_][a-zA-Z0-9_]+`	ID
`\+\+`	INC	`[0-9]+`	NUM
`/`	'/'	`[0-9]+\.[0-9]+`	NUM
`\.`	'.'	`[0-9]+\.`	NUM
`while`	WHILE	`\.[0-9]+`	NUM
`if`	IF	`\"\"`	STRING
`for`	FOR	`\"[^\"\n]+\"`	STRING

Ambiguidade na especificação

- Uma especificação mais complexa como a das maiorias das linguagens de programação imperativas, é naturalmente ambígua
  - Uma entrada `123.4` pode ser um token NUM (`123.4`), dois tokens NUM (`123` e `.4`), um token NUM seguido de um `.` seguido de outro NUM (`123`, `.`, `4`), ou variações disso (`1`, `23`, `.4`)
  - Uma entrada `fora` pode ser um token ID (`fora`), ou um token FOR e um ID (`for`, `a`)
  - `while` pode ser tanto um ID quanto um token WHILE
- Precisamos de regras para remoção da ambiguidade

Removendo este tipo de ambiguidade

- Caso mais de uma regra consiga classificar os próximos caracteres da entrada, dá-se preferência aquela que consegue classificar o maior número de caracteres
  - Ou seja, `123.4` é um único token NUM, e `fora` é um token ID
- Se ainda assim existem várias regras que classificam o mesmo número de caracteres, dá-se preferência à que vem primeiro
  - Logo, `while` seria classificado como WHILE (palavra reservada) => Usa-se precedência

Exercícios - Considere os seguintes dados:

```
41o9'N      8o38'W      11-02-2007 9:10:34  Porto
38o42'N     9o11'W      11-02-2007 9:10:42  Lisboa
51o30'25"N  0o07'39"W   11-02-2007 9:10:43  Londres
22o54'30"S  43o11'47"W  11-02-2007 9:10:43  Rio
55o45'8"N   37o37'56"E  11-02-2007 9:10:45  Moscovo
```

- Crie expressões regulares para extrair os seguintes dados presentes na tabela a seguir:

Campo	**Exemplo**
Latitude	41o9'N
Longitude	8o38'W
Data	11-02-2007
Hora	9:10:34
Cidade	Porto

Exercícios - Extração de links

- Abra a página de algum portal e realize a extração do **conteúdo** de uma *tag* `HTML` informada pelo usuário.

Leitura Recomendada

Capítulo 2: Varredura, Seção 2.2. Expressões Regulares

- LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220.

Capítulos 2 e 3

- JARGAS, Aurélio Marinho. Expressões regulares: uma abordagem divertida . 4. ed. rev. e ampl. São Paulo: Novatec, 2012. 223 p. ISBN 9788575222126.

Expressões Regulares em C

- TREVISAN, Thobias Salazar. Usando Expressões Regulares na Linguagem C. 06/07/2004. Disponível em `http://www.thobias.org/doc/er_c.html`

Próximas Aulas

Próximas Aulas: Análise Léxica

- Autômatos
- Ferramentas para a Análise Léxica.