PRG1 (11): 正規表現とその周辺

脇田建

2016.11.7

正規表現

- 授業で学ぶもの
- r ::=
  - ε
  - c
  - r1 r2
  - r1 | r2
  - r*
- Scalaで使える正規表現
- r ::= 左のもの
  - [abcdef]
  - [a-z] [0-9]
  - [^ \t]
  - r? r+ r{3} r{3,} r{3, 5}
  - \d \s \w
  - (r) (?:r)
  - \1

Scalaにおける正規表現の使用例

- lx10/src/reexameple.scala

正規表現の処理方法

- 大学では DFA に変換する方法を学ぶが。。。
  - 正規表現 => NFA => ε閉包 => Kleene 閉包 => DFA
    - DFAの計算量は入力文字列長nについて線形
- 巷に溢れる正規表現ライブラリは、NFAを用いたものが多い。
  - 計算量は入力文字列長さについて指数オーダー
  - lx10/src/rebenchmark.{scala, py}

$a?^k a^k$

- 例: 正規表現 (a?a?a?a?a?aaaaa) vs 文字列(aaaaa)

- Scala – 26: 1.51秒、27: 3.21秒、28: 6.19秒、29: 13.35秒

- Python – 26: 2.11秒、27: 9.18秒、28: 18.78秒

仮想機械を用いた正規表現エンジン

- 正規表現処理のための専用の仮想機械を用意し、与えられた正規表現を仮想機械の命令列に変換した上で仮想機械を実行することで、さまざまな正規表現処理が実行できる。
- 正規表現仮想機械が提供する４つの命令
  - char c: 文字 c を受理。次の文字を見にいく
  - jump x: x 番目の命令に移動する
  - split x, y: 並列実行。x番目の命令から始まる命令列の実行系とy番目の命令から始まる命令列の実行系を並行動作させる（並列実行も可能だが普通はやらない。（論理的）並行 != (実時間的)並列）
  - match: 受理する

正規表現の命令列への変換規則 (T)

- T[r1 r2] => T[r1]; T[r2]

- T[r1 | r2] =>
  split L1, L2
  L1: { T[r1]; jump L3 }
  L2: T[r2]
  L3:

- T[r?] =>
  split L1, L2
  L1: T[r]
  L2:

- T[r*] =>
  L1: split L2, L3
  L2: { T[r]; jump L1 }
  L3:

- T[r+] =>
  L1: { T[r]; split L1, L2 }
  L2:

命令列の変換例: T[a+b+]

- T[a+b+]; match =>
- T[a+]; T[b+]; match =>
  - L1: T[a]
    split L1, L2
    L2: T[b+]
    match =>
- L1: char a
  split L1, L2
  L2: T[b+]
  match =>
- L1: char a
  split L1, L2
  L2: char b
  split L2, L3
  L3: match
- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- 0番命令から開始：aを読み込む

Thread	pc: 命令	sp (文字列参照)
T1	0: char a	aab

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b

- split命令なので2番命令から始まるT2を生成。T2は出待ち。

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- T1はaを読み込む

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T2	2: char b	a**a**b

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T1	1: split 0, 2	a**a**b
T2	2: char b	a**a**b

- split命令なので2番命令
  から始まるT3を生成。
  T3は出待ち。

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- T1はaを読み込めず死亡。
  T2が動き始める

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T1	1: split 0, 2	a**a**b
T1	0: char a	aa**b**
T2	2: char b	a**a**b
T3	2: char b	aa**b**

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- T2はbを読み込めず死亡。
  T3が動き始める

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T1	1: split 0, 2	a**a**b
T1	0: char a	aa**b**
T2	2: char b	a**a**b
T3	2: char b	aa**b**

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- T3はbを読み込む

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T1	1: split 0, 2	a**a**b
T1	0: char a	aa**b**
T2	2: char b	a**a**b
T3	2: char b	aa**b**

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- 4番命令から始まるT4を生成。T4は出待ち。

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T1	1: split 0, 2	a**a**b
T1	0: char a	aa**b**
T2	2: char b	a**a**b
T3	2: char b	aa**b**
T3	3: split 2, 4	aab_

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- T3はbを読み込めず死亡。
  T4が動き始める

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T1	1: split 0, 2	a**a**b
T1	0: char a	aa**b**
T2	2: char b	a**a**b
T3	2: char b	aa**b**
T3	3: split 2, 4	aab_
T3	2: char b	aab_
T4	4: match	aab_

仮想機械の動作: 入力文字列 aab

- 0: char a
  1: split 0, 2
  2: char b
  3: split 2, 4
  5: match

- match命令
  なので受理！

スレッド	pc: 命令	sp (文字列参照)
T1	0: char a	**a**ab
T1	1: split 0, 2	a**a**b
T1	0: char a	a**a**b
T1	1: split 0, 2	a**a**b
T1	0: char a	aa**b**
T2	2: char b	a**a**b
T3	2: char b	aa**b**
T3	3: split 2, 4	aab_
T3	2: char b	aab_
T4	4: match	aab_

Scalaによる実装例

- プロジェクト lx11
  - コンパイラ：正規表現 → 仮想命令列
  - 再帰的な仮想機械: RecursiveBacktrackingVM
  - 逐次的な仮想機械: IterativeBacktrackingVM
  - Thompsonによる効率的な仮想機械:KenThompsonVM

extends (case classes)

- trait を拡張した case class

  - trait RegularExpression
    case object Empty extends RegularExpression
    case class Character(c: Char) extends Instruction
    case class Jump(x: Int) extends Instruction
    case class Split(x: Int, y: Int) extends Instruction
    case object Match extends Instruction

- RegularExpression – 親trait

- Emptyオブジェクト、Characterクラス、Jumpクラス、Splitクラス、Matchオブジェクト – trait の子オブジェクトや子クラス

- 「Emptyオブジェクトの親traitはRegularExpressionです」「JumpクラスはRegularExpression の子クラスです」「MatchオブジェクトはRegularExpressionの子オブジェクトです」

extends object

- src
  - object Empty extends RegularExpression
  - object RecursiveBacktrackingVM extends VM
  - object IterativeBacktrackingVM extends VM
  - object KenThompsonVM extends VM
- test
  - object TooSlow extends Exception

extends class

- trait RegularExpression {
  // 型だけが宣言されたメソッド：trait を extends した object/class が責任をもって定義しなくてはいけない
  def _compile(label: Int): (Int, LProgram)

  // trait に定義された val/var/def はこの trait を extends した object/class に継承される。
  def compile: Program = (_compile(0)._2 ++ List(Match)).toArray
}

- object Empty extends RegularExpression { … }
  class C(c) extends RegularExpression { … }
  class Concatenate(r1, r2) extends RegularExpression { … }
  class Alternate(r1, r2) extends RegularExpression { … }
  class Star(r) extends RegularExpression { … }

override def f(...) { ... }

- 親のメソッドの定義を子が再定義するときに用いる。多くの場合は、親のメソッドをより詳細化したい場合に再定義する。

  - class AnyRef {
    def toString(): String { … }
    }

  - trait RegularExpression {
    / / Scala は、明示的に extends しない trait / object / class は AnyRef を extends すると見做す
    def _compile(label: Int): (Int, LProgram)

    def compile: Program = (_compile(0)._2 ++ List(Match)).toArray
    }

  - case class C(c: Char) extends RegularExpression {
    override def toString: String = c.toString
    def _compile(label0: Int): (Int, LProgram) = { … }
    }

大域脱出：try … catch { case … } / throw e

- try {「なにかやりたい処理」} catch { case … => 例外処理 }
  - try:「なにかやりたい処理」を実行しつつ、例外に備える
  - catch: try ブロックの実行中の例外をパターンマッチで掴まえる。
- **object** TooSlow **extends Exception**
  - 例外としては Exception という特殊なクラスを extends したオブジェクトを利用できる。
- 例
  - try {
    for (k <- 15 to 30) {
    val t_start = System.nanoTime()
    benchmark(k)
    val t = (System.nanoTime() - t_start) / 1000000000.0
    if (t > time_limit) throw **TooSlow**
    // throw文はTooSlow 例外を発生する。次の瞬間、これまでの実行は中止され、該当する catch 構文が実行される。
    println(f"$k%6d: $t%5.2fs")
    }
    } catch { case **TooSlow** => println(“時間かかりすぎ！") }

- Russ Cox, “Regular Expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, …),” (Jan 2007).
  http://bit.ly/1Fo3RaY

- Russ Cox, “Regular expression matching: the virtual machine approach,” (Dec. 2009).
  http://bit.ly/2f0kRSO