

Voorbeeldvragen Algoritmen & Datastructuren I

Pattern Matching

Titularis: Prof. Dr. Wolfgang De Meuter

Assistenten: Nathalie Oostvogels, Simon Van de Water

15 december 2015

Vraag 1

Sommige varianten van Scheme (zoals DrScheme) laten toe dat er andere “soorten” haakjes gebruikt worden om de leesbaarheid van een programma te bevorderen. Een Scheme-expressie begint dan door een haakje “(”, vierkant haakje “[”, of accolade “{” te openen, en eindigt door met het overeenkomstige teken te sluiten: “(. . .)”, “[. . .]”, of “{ . . . }”. Een geneste Scheme-expressie is een Scheme-expressie die begint en eindigt in een omliggende Scheme-expressie. Ongeldige Scheme-expressies zijn expressies waarvan de “haakjes” niet op deze manier gebalanceerd zijn.

Zoals je weet merk je het gebruik van ongedefinieerde variabelen in een Scheme-programma pas wanneer je het programma uitvoert. Een andere veel voorkomende fout zijn slecht gebalanceerde haakjes. Om reeds bij het schrijven van het programma deze fouten op te merken willen we onze Scheme-programma's doorlopen om te kijken waar ongedefinieerde variabelen gebruikt worden en deze rood onderlijnen, zoals bij spellingscorrectie voor tekstverwerking. Bovendien willen we tegelijk ook slecht gebalanceerde haakjes opsporen.

Een Scheme-programma is een lange string waarin geneste Scheme-expressies staan tussen haakjes. Schrijf een procedure die de hoeveelste positie in de tekst dat een woord (dat een ongedefinieerde variabele kan zijn) staat **en** in welke expressie van het programma het woord voorkomt teruggeeft.

Dit wil zeggen dat de procedure die je gaat schrijven twee argumenten neemt: een string die het Scheme-programma voorstelt en een string die het te zoeken woord voorstelt. Je procedure moet 3 getallen teruggeven:

- het eerste getal geeft de positie in de tekst van het woord aan,
- het tweede getal geeft de positie aan van het begin van de Scheme-expressie in de programmatekst,
- en het derde getal tenslotte geeft het nesting-niveau aan van de expressie waarin de fout staat.

Voorbeelden

```
(match-identifier-in-expression
  "(define (hello-world) [display 'Helloworld] (newline))" "display")
>> {23 22 1}
```

... Want “display” staat op positie 23, de expressie waarin deze identifier te vinden is op positie 22, en deze expressie is 1 niveau diep genest (“(define . . .)” is de enige omliggende expressie).

```
(match-identifier-in-expression
  "(define (hello-world) (begin [display 'Helloworld] (newline)))" "newline")
>> {52 51 2}
```

... Want “newline” staat op positie 52, de expressie waarin deze identifier te vinden is op positie 51, en deze expressie is 2 niveau's diep genest.

```
(match-identifier-in-expression
  "(define (hello-world) [display 'Helloworld] (newline))" "display")
>> #f
```

```
(match-identifier-in-expression
  "(define hello-world) [display 'Helloworld] (newline))" "display")
>> ERROR invalid Scheme expression at: 19.
```

```
(match-identifier-in-expression  
  "(define (hello-world) [display 'Helloworld] (newline))" "display")  
>> ERROR invalid Scheme expression at: 20.
```

Schrijf deze procedure zo efficiënt mogelijk. Je kan je hiervoor baseren op de code en de bestaande ADT's uit de cursus. Geef goed aan welke code je gebruikt of aanpast. Hoe snel is je algoritme? Leg uit.

```

(library
  (brute-force)
  (export match)
  (import (rnrs base (6))))

(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (loop (+ i-t 1) 0))))))

```

```

(library
  (kmp)
  (export match)
  (import (rnrs base (6))
          (rnrs io simple)))

(define (compute-failure-function p)
  (define n-p (string-length p))
  (define sigma-table (make-vector n-p 0))
  (let loop
    ((i-p 2)
     (k 0))
    (cond
      ((>= i-p n-p)
       (vector-set! sigma-table (- n-p 1) k))
      ((eq? (string-ref p k)
            (string-ref p (- i-p 1)))
       (vector-set! sigma-table i-p (+ k 1))
       (loop (+ i-p 1) (+ k 1)))
      ((> k 0)
       (loop i-p (vector-ref sigma-table k)))
      (else ; k=0
       (vector-set! sigma-table i-p 0)
       (loop (+ i-p 1) k))))
  (vector-set! sigma-table 0 -1)
  (lambda (q)
    (vector-ref sigma-table q)))

(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define sigma (compute-failure-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (loop (+ i-t (- i-p (sigma i-p))) (if (> i-p 0)
                                              (sigma i-p)
                                              0))))))

```

```

(library
  (quicksearch)
  (export match)
  (import (rnrs base (6))))

(define (compute-shift-function p)
  (define n-p (string-length p))
  (define min-ascii (char->integer (string-ref p 0)))
  (define max-ascii min-ascii)

  (define (create-table index)
    (if (< index n-p)
        (begin
          (set! min-ascii (min min-ascii (char->integer (string-ref p index))))
          (set! max-ascii (max max-ascii (char->integer (string-ref p index))))
          (create-table (+ index 1)))
        (make-vector (- max-ascii min-ascii -1) (+ n-p 1))))

  (define (fill-table index)
    (if (< index n-p)
        (let* ((ascii (char->integer (string-ref p index)))
               (vector-set! shift-table (- ascii min-ascii) (- n-p index)))
          (fill-table (+ index 1))))

  (define shift-table (create-table 0))
  (fill-table 0)
  (lambda (c)
    (let ((ascii (char->integer c)))
      (if (>= max-ascii ascii min-ascii)
          (vector-ref shift-table (- ascii min-ascii)
                      (+ n-p 1))))))

(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define shift (compute-shift-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (let ((c-t (string-ref t (mod (+ i-t n-p) n-t))))
         (loop (+ i-t (shift c-t)) 0))))))

```

Vraag 2

Schrijf een procedure die gegeven een string, de eerste positie in de string teruggeeft waar de *langste herhaalde prefix* uit die string zich bevindt, d.w.z. de *eerste* positie in de string waar de langst mogelijke prefix begint die ergens in dezelfde string herhaald wordt.

Enkele voorbeelden:

- voor de string “abcdxyzabcdxyz” is dit 7;
- Voor de string “pabcdxyzabcdxyz” is er geen herhaalde prefix.

Zorg ervoor dat de performantie van je oplossing niet slechter is dan $O(n)$ voor een string van lengte n .

Je kan vertrekken van de geziene procedures uit hoofdstuk 2 die je terugvindt op de ommezijde. Geef desgevallend duidelijk aan van welke procedure je vertrekt.

Vraag 3

Implementeer een procedure `match-pattern-in-word` die een patroon in een string zoekt en een paar teruggeeft dat bestaat uit twee getallen: de positie van het patroon in de string en alsook het rangnummer van het woord waarin dat patroon voorkomt. Wanneer een patroon over meerdere woorden loopt geef je het rangnummer van het eerste woord. Doe dit zo efficiënt mogelijk.

Een voorbeeld:

```
> (match-pattern-in-word "Zoek het patroon in het woord" "troon")  
(11 . 3)
```

Ter herinnering geven we je de patroonherkenningsalgoritmen uit de cursus waarop je je implementatie kan baseren.