Assignment 2 — 2014

Theodore S. Norvell

6892 Due Oct 23 2014

Q0 [10]

(a) [5] Design the signature pre and post-conditions for a procedure that computes the set of all n letter words that can be made from the items of the set where n is a natural number.

A Solution

proc *words*($n : \mathbb{N}, S : \text{Set}\langle T\rangle$) : Set $\langle \text{Seq}\langle T\rangle\rangle$
    precondition true
    postcondition $result = \{s \in \text{Seq}\langle T\rangle \mid s.\text{length} = n \wedge s\{0, ..n\} \subseteq S\}$

(b) [5] Design the procedure.

Some solutions: There are a couple of ways to do this.

One is Divide and Conquer. For this I will need a procedure that produces the set of all sequences that extend a given prefix to length n. Recall that t is a prefix of s is there is a u such that $s = t\hat{}u$. I.e. I need a procedure

proc *wordsWithPrefix*($n : \mathbb{N}, S : \text{Set}\langle T\rangle, t : \text{Seq}\langle T\rangle$) : Set $\langle \text{Seq}\langle T\rangle\rangle$
    precondition $t.\text{length} \le n \wedge t\{0, ..t.\text{length}\} \subseteq S$
    postcondition $result = \{s \in \text{Seq}\langle T\rangle \mid s.\text{length} = n + t.\text{length} \wedge s\{0, ..n\} \subseteq S \wedge t$ is a prefix of $s\}$
    if $n = t.\text{length}$ then
        return $\{t\}$
    else
        var $R := \emptyset$
        for $x \in S$ do $R := R \cup wordsWithPrefix(n, S, t\hat{}[x])$ end for
        return R
    end if
end *wordsWithPrefix*

Then the original problem is solved by $wordsWithPrefix(n, S, [\,])$. I.e. we can implement the specification from part (a) with

proc *words*($n : \mathbb{N}, S : \text{Set}\langle T\rangle$) : Set $\langle \text{Seq}\langle T\rangle\rangle$
    precondition true

postcondition $result = \{s \in \text{Seq}\langle T\rangle \mid s.\text{length} = n \wedge s\{0,..n\} \subseteq S\}$
return $wordsWithPrefix(n, S, [\,])$
end *words*

One property of this solution is that the union is always of disjoint sets, so we can represent S very efficiently, e.g., by a doubly linked list.

A second approach is bottom up. Find all words of length $n-1$ and then extend them to make longer words. Here the unions might not be of disjoint sets:

```
proc words( n : ℕ, S : Set⟨T⟩ ) : Set⟨Seq⟨T⟩⟩
    precondition true
    postcondition result = {s ∈ Seq⟨T⟩ | s.length = n ∧ s{0,..n} ⊆ S}
    if n = 0 then
        return {[]}
    else
        var P := words(n − 1, S)
        var R := ∅
        for t ∈ P
            for x ∈ S
                for i ∈ {0,..n − 1} R := R ∪ {t[0,..i]ˆ[x]ˆt[i,..n − 1]} end for
            end for
        end for
        return R
    end if
end words
```

We could also do this nonrecursively

```
proc words( n : ℕ, S : Set⟨T⟩ ) : Set⟨Seq⟨T⟩⟩
    precondition true
    postcondition result = {s ∈ Seq⟨T⟩ | s.length = n ∧ s{0,..n} ⊆ S}
    var P := {[]}
    var k := 0
    // inv P contains all words of length k with elements from S and 0 ≤ k ≤ n
    while k < n do
        var R := ∅
        for t ∈ P
            for x ∈ S
                for i ∈ {0,..k} R := R ∪ {t[0,..i]ˆ[x]ˆt[i,..n − 1]} end for
            end for
        end for
        P := R
        k := k + 1
    end while
    return P
end words
```

Q1 [10] Given a sequence of one or more arrays that we wish to find the product of, say $ABCD$, there are several ways the sequence could be parenthesized. In the example we have

$$
\begin{gathered}
(((AB)C)D) \\
((AB)(CD)) \\
((A(BC))D) \\
(A((BC)D)) \\
(A(B(CD)))
\end{gathered}
$$

(a)[5] Design a procedure that, given a sequence of n characters, prints a list that contains of all parenthesizations of that sequence. Include pre- and postconditions, even if they are not very formal.

A solution:
As an informal specification

proc *parens*(s : Seq ⟨Char⟩) : Set ⟨Seq ⟨Char⟩⟩
  precond s.length > 0
  postcond *result* = the set of all parenthesizations of s

Given a sequence like "ABCDE", we can consider all the places the final multiplication could go, after the A, after the B, etc. For each of these, we can consider all the ways of parenthesizing the items on either side of the final multiplication.

```
proc parens( s : Seq ⟨Char⟩ ) : Set ⟨Seq ⟨Char⟩⟩
   precond s.length > 0
   postcond result = the set of all parenthesizations of s
   if s.length = 1 then
       return {s}
   else
       // Try each way of splitting the sequence
       var R := ∅
       for k ∈ {1, ..s.length} do
           var P := parens( s[0, ..k] )
           var Q := parens( s[k, ..s.length] )
           for p ∈ P, q ∈ Q do
               R := R ∪ ['('] ^p^q^ [')']
           end for
       end for
       return R
   end if
end parens
```

Another solution: This solution finds all the ways to combine a sequence of partial solutions.

As an informal specification

proc *parens*(s : Seq ⟨Seq ⟨Char⟩⟩) : Set ⟨Seq ⟨Char⟩⟩
  precond s.length > 0 and each item of s is a valid parenthesizations.
  postcond *result* = the set of all ways of combining the items of s to make
    a valid parenthesization

For example if we input a sequence

[“A”, “(BC)”, “((DE)F)”]

there are 2 ways to combine these, so the output is

{“(A((BC)((DE)F))”, “((A(BC))((DE)F))”}

If s has length 1, there is only one solution. When s is longer, we can combine any two adjacent items we can then compute all ways of completing the task with a recursive call. There are s.length $- 1$ adjacent pairs; we need to try each.

proc *parens*(s : Seq ⟨Seq ⟨Char⟩⟩) : Set ⟨Seq ⟨Char⟩⟩
  precond s.length > 0 and each item of s is a valid parenthesizations.
  postcond *result* = the set of all ways of combining the items of s to make
    a valid parenthesization
  if s.length $= 1$ then return $\{s(0)\}$
  else
    var $R := \emptyset$
    for $i \in \{0, ..i - 1\}$
      // Try combining items i and $i + 1$
      val $t := s[0, ..i]$
      val $u :=$ [‘(’ˆ$s(i)$ˆ$s(i + 1)$ˆ‘)’]
      val $v := s[i + 1, ..s.\text{length}]$
      // Note that tˆuˆv is shorter by 1 than s.
      $R := R \cup$ *parens*(tˆuˆv)
    end for
    return R
  end if
end *parens*

This solution is considerably less efficient than the first, as it will find some solutions more than once.

(b)[5] Suppose that besides a sequence of n characters (representing the names of matrices), we are also given a list D of $n+1$ dimensions. The dimensions of matrix i are $d(i)$ rows by $d(i+1)$ columns. Each parenthesization is then associated with a cost which is the sum of the costs of the multiplications. The cost of multiplying a p by q matrix with a q by r matrix is $p \times q \times r$.

For example suppose we want to find the product $ABCD$ where

Matrix	Rows	Columns
A	3	4
B	4	5
C	5	2
D	2	4

The costs of the 5 parenthesizations is

Parethesization	cost
(((AB)C)D)	114
((A(BC))D)	160
((A(BC))D)	88
(A((BC)D))	120
(A(B(CD)))	168

,

so the minimum cost is 88.

Design an algorithm to compute the cost of the least-cost parenthesization. Do not worry too much about efficiency of your algorithm.

Some Solutions: I'll keep the same structure as before, but this time instead of unioning to get a set, I'll compute the minimum

proc $minCostMM(\ d : \mathrm{Seq}\,\langle\mathbb{N}\rangle\ ) : \mathbb{N}$
  precond d.length > 1
  postcond $result =$ the minimum cost of all parenthesizations
  if d.length $= 2$ then
    return 0
  else
    // Try each way of splitting the sequence
    var $r := \emptyset$
    for $j \in \{1, ..d.\mathrm{length} - 1\}$ do
      var $p := minCostMM(\ d[0, .., j]\ )$
      var $q := minCostMM(\ d[j, .., d.\mathrm{length} - 1]\ )$
      $r := r \min\, (p + q + (d(0) \times d(j) \times d(d.\mathrm{length} - 1)))$
    end for
    return r
  end if
end $minCostMM$

Another way to do it is to leave d alone, but to give indices for the first and last dimensions that are involved. This is marginally more efficient as there is no data structure manipulation needed.

proc $minCostMM(\ i, k : \mathbb{N}, d : \mathrm{Seq}\,\langle\mathbb{N}\rangle\ ) : \mathbb{N}$
  precond $d.\mathrm{length} > 1 \wedge 0 \le i < k < d.\mathrm{length}$
  postcond $result =$ the minimum cost of computing the $d(i)$ by $d(k)$ matrix

from the $k-i$ matrices represented by dimensions $d(i), .., d(k)$
if $k - i = 1$ then
return 0
else
// Try each way of splitting the sequence
var $r := \emptyset$
for $j \in \{i+1, .., k-1\}$ do
var $p := minCostMM(i, j, d)$
var $q := minCostMM(j, k, d)$
$r := r \min (p + q + (d(i) \times d(j) \times d(k)))$
end for
return r
end if
end *minCostMM*

Now we have

proc *minCostMM*($d : \text{Seq}\langle\mathbb{N}\rangle$) : $\mathbb{N}$
precond d.length > 1
postcond $result =$ the minimum cost of all parenthesizations
return $minCostMM(0, d.\text{length} - 1, d)$
end *minCostMM*

Another solution. The final solution is based on the final solution presented for part (a). This time there are 2 input lists. The first is a list c of the costs to produce each of the parenthesizations represented by the s parameter in the last solution for part a. The second is a list d which holds the dimensions of the parenthesizations represented by s. The s parameter isn't actually needed, so we leave it out. For example to compute the minimum cost of computing $ABCD$, where A is 3 by 4, B is 4 by 5, C is 5 by 2, and D is 2 by 4, we would call

$$minCostMM([0, 0, 0, 0], [3, 4, 5, 2, 4])$$

The 0s here represent the cost of computing 4 individual inputs, which we assume is 0 in each case, since they are single matrices and not products.

proc *minCostMM*($c : \text{Seq}\langle\mathbb{N}\rangle, d : \text{Seq}\langle\mathbb{N}\rangle$) : $\mathbb{N}$
precond c.length $> 0 \wedge d$.length $= c$.length $+ 1$
postcond $result =$ the minimum cost of all parenthesizations assuming that c represents the cost of a sequence of input parenthesizations and d represents the dimensions
if c.length $= 1$ then
return $c(0)$
else
var $m := \infty$
for $i \in \{0, ..i - 1\}$
// Try combining items i and $i+1$ of c
val $t := c[0, ..i]$
val $u := [d(i) \times d(i+1) \times d(i+2) + c(i) + c(i+1)]$
val $v := c[i+1, ..c.\text{length}]$

// Note that $t\hat{}u\hat{}v$ is shorter by 1 than c.
// In recursing, we cut out the dimension shared by the matrices
// whose costs are represented by $c(i)$ and $c(i+1)$; i.e., we cut out $d(i+1)$.
$m := m \min \mathit{minCostMM}(t\hat{}u\hat{}v, d[0, ..i+1]\hat{}d[i+2, ..d.\text{length}])$
end for
return m
end if
end *minCostMM*

Q2 [5] An ordered tree is a directed tree such that each node is either a leaf or a branch. Leaves have no children. Branches have a sequence of 0 or more children. For this question, nodes are labelled with nonempty, finite strings consisting of lower-case letters.

Design a context-free grammar that describes the language of depth-first traversals of such finite ordered trees. Three examples from the language are

```
fred
georgina()
henry(ingrid(john),kate,marty())
```

In these examples `fred`, `john`, and `kate` label leaf nodes; `georgina` and `marty` label branch nodes with no children; `ingrid` labels a branch node with one child; and `henry` labels a branch node with three children.

Be sure to describe the alphabet, the nonterminal set, the starting nonterminal, and the production set of the grammar.

A Solution:

- Alphabet: $A = \{\text{'a'}, \text{'b'}, .., \text{'z'}, \text{'('}, \text{')'}, \text{','}\}$
- Nonterminals: $N = \{\text{tree}, \text{nonemptyList}, \text{list}, \text{letter}, \text{word}\}$
- Start nonterminal: tree.
- Productions

$$\begin{array}{rcl}
\text{tree} & \rightarrow & \text{word} \\
\text{tree} & \rightarrow & \text{word(list)} \\
\text{list} & \rightarrow & \epsilon \\
\text{list} & \rightarrow & \text{nonemptyList} \\
\text{nonemptyList} & \rightarrow & \text{tree} \\
\text{nonemptyList} & \rightarrow & \text{nonemptyList}, \text{nonemptyList} \\
\text{word} & \rightarrow & \text{letter} \\
\text{word} & \rightarrow & \text{word word} \\
\text{letter} & \rightarrow & \text{a} \\
\text{letter} & \rightarrow & \text{b} \\
 & & ... \\
\text{letter} & \rightarrow & \text{z}
\end{array}$$

Bonus [5] Design a procedure that inspects a string and determines whether it is in the language described in Q2.

Solution. With a few changes to the grammar, the technique of recursive descent can be used, as outlined in slide set 10.5.

For the sake of variety, I'll present a solution that relies on a completely different principle. The kind of parser presented below is called a shift-reduce parser. For all but the simplest of grammars, these are usually not coded by hand, but rather derived from grammars using tools. Yacc and bison are two well known tools for turning grammars into shift-reduce parsers.

Suppose the input is in t. The parser uses variables

- s — the remaining input followed by a sentinel symbol \$.
- α — a stack of alphabet and nonterminal symbols. The top of the stack will be to the right, i.e., the bottom is at index 0.

We initialize the variables with

$$s := t\$ \qquad \alpha := \epsilon$$

This establishes an invariant

$$\alpha s \overset{*}{\Longrightarrow} t\$ \tag{0}$$

There are two kinds of steps taken by the parser. Note that each preserves invariant (0).

- A shift step removes the first item from the input string and pushes it onto the stack.

$$\text{shift} = (\alpha := \alpha\hat{}[s(0)]\ s := s[1, ..s.\text{length}])$$

- A reduce step pops a sequence of symbols that equals the right-hand side of a production and then pushes the left-hand side of the same production.

$$\text{reduce}(n \to \beta) = (\alpha := \alpha[0, ..\alpha.\text{length} - \beta.\text{length}]\hat{}[n])$$

The algorithm is to take shift and reduce steps according to the following table until no further step is possible. If the table says two actions are applicable, the first one is taken

Top of stack is	**Next input is in**	**Action**
any letter x	any	reduce(letter $\to x$)
letter	any	reduce(word $\to$ letter)
word word	any	reduce(word $\to$ word word)
word	{',', ')', \$}	reduce(tree $\to$ word)
word(list)	any	reduce(tree $\to$ word(list))
'('	{')'}	reduce(list $\to \epsilon$)
nonemptyList	{')'}	reduce(list $\to$ nonemptyList)
tree	{',', ')'}	reduce(nonemptyList $\to$ tree)
nonemptyList, nonemptyList	any	reduce(nonemptyList $\to$ nonemptyList, nonemptyList)
any	any but \$	shift

Upon stopping, if $\alpha =$ tree and $s =$ \$, then, by the invariant (0), tree\$ $\overset{*}{\Longrightarrow} t\$$ and so tree $\overset{*}{\Longrightarrow} t$.

The table is designed so that the following invariant is also maintained

$$\text{if tree} \stackrel{*}{\Longrightarrow} t \text{ then tree\$} \stackrel{*}{\Longrightarrow} \alpha s \tag{1}$$

It is also designed so that. when no rule is applicable, either $\alpha = \text{tree} \wedge s = \$$ or it is not true that tree\$ $\stackrel{*}{\Longrightarrow} \alpha s$. So, provided that we have successfully maintained invariant (1), upon stopping, unless $\alpha = \text{tree} \wedge s = \$$ is true, tree\$ $\stackrel{*}{\Longrightarrow} \alpha s$ is untrue and so by (1), tree $\stackrel{*}{\Longrightarrow} t$ will also be untrue

The complete algorithm is

$s := t\$$
$\alpha := \epsilon$
while there is a rule in the table that applies
    apply the first rule that applies
end while
$f := \alpha = \text{tree} \wedge s = \$$