Q1

Bounded minimalization

Show that $\boldsymbol{\mu j}$ $\boldsymbol{((g(j, \underline{n})| \ 0 \leq j \leq J)}$ *:= the minimum value of* $\boldsymbol{j}$ *for which the predicate* $\boldsymbol{g(j, \underline{n})}$ *is true, is a PRF.*

Solution:

If g (j, $\underline{n}$) is a primitive recursive (which is true because it is a predicate function) then

f(j, $\underline{n}$) = μ j ((g(j, n) | 0 ≤ j ≤ J) which f(j, $\underline{n}$) = 0 else f(j, $\underline{n}$) = j is also a primitive recursive function.

The given function is a PRF because it is a bounded minimalization function and it is created with the composition of PRFs.

Q2.

4.6.1. (a) Give a derivation of the string $aaabbbccc$ in the grammar of Example 4.6.2.

(b) Prove carefully that the grammar in Example 4.6.2 generates the language $L = \{a^n b^n c^n : n \geq 1\}$.

Example 4.6.2: The following grammar G generates the language $\{a^n b^n c^n : n \geq 1\}$. $G = (V, \Sigma, R, S)$, where

$$
\begin{aligned}
V &= \{S, a, b, c, A, B, C, T_a, T_b, T_c\}, \\
\Sigma &= \{a, b, c\}, \text{and} \\
R &= \{S \to ABCS, \\
&\quad S \to T_c, \\
&\quad CA \to AC, \\
&\quad BA \to AB, \\
&\quad CB \to BC, \\
&\quad CT_c \to T_c c, \\
&\quad CT_c \to T_b c, \\
&\quad BT_b \to T_b b, \\
&\quad BT_b \to T_a b, \\
&\quad AT_a \to T_a a, \\
&\quad T_a \to e\}.
\end{aligned}
$$

The first three rules generate a string of the form $(ABC)^n T_c$. Then the next three rules allow the A's, B's, and C's in the string to "sort out" themselves correctly, so that the string becomes $A^n B^n C^n T_c$. Finally, the remaining rules allow the T_c to "migrate" to the left, transforming all C's to c's, and then becoming T_b. In turn, T_b migrates to the left, transforming all B's into b's and becoming T_a, and finally T_a transforms all A's into a's and then is erased.

It is rather obvious that any string of the form $a^n b^n c^n$ can be produced this way. Of course, many more strings that contain nonterminals can be produced; however, it is not hard to see that the only way to erase all nonterminals is to follow the procedure outlined above. Thus, the only strings in $\{a, b, c\}$ that can be generated by G are those in $\{a^n b^n c^n : n \geq 1\}$.$\Diamond$

Solution:

a) We can generate the wanted string by applying the rules starting from S.

S

$\rightarrow$ ABCS

$\rightarrow$ ABCABCS

$\rightarrow$ ABCABCABCS

$\rightarrow$ ABCABCABCT_c

$\rightarrow$ ABCABACBCT_c

$\rightarrow$ ABCABABCCT_c

$\rightarrow$ ABACBABCCT_c

$\rightarrow$ ABABCABCCT_c

$\rightarrow$ ABABACBCCT_c

$\rightarrow$ ABABABCCCT_c

$\rightarrow$ ABAABBCCCT_c

$\rightarrow$ AABABBCCCT_c

$\rightarrow$ AAABBBCCCT_c

$\rightarrow$ AAABBBCCT_c c

$\rightarrow$ AAABBBCT_c cc

$\rightarrow$ AAABBBT_b ccc

$\rightarrow$ AAABBT_b bccc

$\rightarrow$ AAABT_b bbccc

$\rightarrow$ AAAT_a bbbccc

$\rightarrow$ AAT_a abbbccc

$\rightarrow$ AT_a aabbbccc

$\rightarrow$ T_a aaabbbccc

$\rightarrow$ aaabbbccc

b) We need to show to cases to prove that the given grammar generates the language L= { a^n b^n c^n, n >= 1} First, we need to show that if w Є L then we are able to generate w starting from S with only given rules. Second case is if the w does not belong to L then there is no way that we can generate w starting from S with only given rules.

(i) Given w in the form of a^n b^n c^n (w Є L):

- Apply the rule S → ABCS n times.
- Sort the ABCs by with the rules CA → AC, BA → AB, and CB → BC. These rules should be applied n*(n-1)/2 times for each to get $A^nB^nC^nT_c$ form. Since there is every binary permutation of A, B, and C it is certain that we can create the wanted form.
- Push the T_c and change the type of T along the way from left to right and convert the non-terminals to the matching terminals.
- After the last T_a nonterminal, we created the w which belongs to L.

(ii) w ∈ Σ* if w does not belong to L, it can be on two different forms. Either it can have different number of as, bs, or cs, or it can have the wrong order.

- First let's examine the case of inequality between the number of letters.
- Without loss of generality, suppose there is a string 'w' with a different number of 'a's (and 'A's) and 'b's (and 'B's).
- Define a function f(w) that calculates the difference between the counts of 'a's (and 'A's) and 'b's (and 'B's) in 'w'.
- Observe that in the given context-free grammar G, every production rule (of the form A → a) has the property that f(A) = f(a). In other words, the difference between 'a's and 'b's is preserved across production rules.
- If there is a derivation of a string from u to v in the grammar (denoted u⇒* v), it implies that f(u) = f(v), since each production rule maintains the difference.
- Now, consider the starting symbol S. Since the language L(G) consists of strings with equal numbers of 'a's and 'b's, s(S) = 0.
- Thus, for any string 'w' that belongs to L(G), we must have s(w) = 0, since it is derived from S and all production rules maintain the difference.
- The contrapositive of this statement is: if s(w) is not equal to 0, then 'w' does not belong to L(G)

- Now, let's examine the case of wrong order of letters.
- The goal is to show that if a string 'w' contains an ordering problem, it does not belong to the language L(G). Without loss of generality, assume 'w' has a 'b' preceding an 'a'.
- If the starting symbol S derives a string 'uTv' (where T is one of the non-terminal symbols {Ta, Tb, Tc}, and both 'u' and 'v' are strings of non-terminal and terminal symbols), then 'u' consists of only non-terminal symbols and 'v' consists of terminal symbols.
- Now, consider the point in the derivation where the misplaced 'a' is produced. The string at this point must have the form 'uTaav' for some 'u' (only non-terminal symbols) and 'v' (terminal symbols).
- However, there are no production rules in the grammar that can transform 'Ta' into a string containing 'Tb'. Therefore, we cannot derive any string that includes 'Tb' from 'uTaav'.
- If there are no 'B's left in 'u', all the 'b's are to the right of the misplaced 'a', which contradicts our assumption that 'w' has a 'b' preceding an 'a'.
- If there are 'B's remaining in 'u', we cannot apply any production rules to convert them into terminal symbols. As a result, we cannot derive a string containing only terminal symbols from 'uTaav', contradicting our assumption that 'w' consists of terminal symbols only.

- Consequently, the assumptions that S⇒* w, w belongs to Σ* (the set of all strings of terminal symbols), and 'w' contains no ordering problem are collectively inconsistent. This means that any string with an ordering problem cannot be a part of the language L(G).

Thus, we proved that the given grammar generates the given language L.

4.6.2. Find grammars that generate the following languages:
(a) $\{ww : w \in \{a, b\}^*\}$
(b) $\{a^{2^n} : n \geq 0\}$
(c) $\{a^{n^2} : n \geq 0\}$

Solution:

a)

G = (V, T, R, S)

T = {a,b}

b)

G = (V, T, R, S)

V= {a, S, L, M, $}

T = {a}

V-T = {S, L, M, $}

R = { S -> La$, L -> LM, L -> e, Ma ->aaM, M$ -> $, $ -> e }

An example generation can be formed like this:

w = a^8 where n = 3

S

-> La$

-> LMa$

->LMMa$

->LMMMa$

->MMMa$

->MMaaM$

->MaaMaM$

->MaaaaMM$

->aaMaaaMM$

->aaaaMaaMM$

->aaaaaaMaMM$

->aaaaaaaaMMM$

->aaaaaaaaMM$

->aaaaaaaaM$

->aaaaaaaa$

->aaaaaaaa

c)

G = (V, T, R, S)

V = {a, S, L, A, Y, R}

T = {a}

R = {S -> LAYR, ZA -> aAZ, Za -> aZ, ZR -> AAYR, aY -> Ya, AY -> YA, LY -> LZ, YR -> X, aX -> Xa, AX -> Xa

LX -> e}

An example generation can be formed like this:

w = a^4 where n = 2

S ->

-> LAYR

-> LYAR

-> LZAR

-> LaAZR

-> LaAAAYR

-> LaAAAX

-> LaAAXa

-> LaAXaa

-> LaXaaa

-> LXaaaa

-> aaaa

4.6.3. Show that any grammar can be converted into an equivalent grammar with rules of the form $uAv \to uwv$, with $A \in V - \Sigma$, and $u, v, w \in V^*$.

For each terminal a ∈ Σ add the rule A → a along with the new nonterminal A. Now we need to convert the other rules to the wanted format.

To achieve for all Ai Є V and w Є V* we need to remove each grammar rule of the form A1 A2... An → w, where n ≥ 2, then add these new rules:

A1 A2 ... An→ R1 A1 A2... An (where A1 goes to R1 A1)

Also, to achieve that Ai → e we need to add:

• Ri Ai Ai+1... An→ Ri+1 Ai+1 for each i ≤ n

Lastly we need to add:

• Rn+1 → w

The resulting grammar will generate the same language L. We showed that it is possible to have the form of uAv → uwv for u, v, w Є V* and AE V-Σ for each rule of the grammar.

4.7.1. Let $f : \mathbf{N} \mapsto \mathbf{N}$ be a primitive recursive function, and define $F : \mathbf{N} \mapsto \mathbf{N}$ by

$$F(n) = f(f(f(\dots f(n)\dots))),$$

where there are n function compositions. Show that F is primitive recursive.

Solution:

f.f.f.f……f(n)

F(n) = f • f • f • … • f(n)

This means that F(n) is the composition of finite number of f functions. f is a primitive recursive function, and the composition of primitive recursive functions are also primitive recursive functions. Hence F(n) is a primitive recursive function.

4.7.2. Show that the following functions are primitive recursive:

(a) factorial$(n) = n!$.

(b) gcd(m, n), the greatest common divisor of m and n.

(c) prime(n), the predicate that is 1 if n is a prime number.

(d) p(n), the nth prime number, where $\mathsf{p}(0) = 2$, $\mathsf{p}(1) = 3$, and so on.

(e) The function log defined in the text.

Solution:

a) factorial(n) = n * factorial(n-1), is PR
factorial(0) = succ • $zero_1$(0) = 1
factorial(n+1) = h(n, factorial(n)) = (n+1) * factorial(n) where:
h(n, k) = mult(n,k)

b) gcd(m, n) can be written as a partial function.
gcd(m, n) = { if rem(m, n) = 0 : n
else : gcd (n, rem(m, n)

}

rem(n, 0) = $zero_1$(n) = 0

rem(n, m+1) = rem(n,m) + 1 if rem(n,m) < n-1

= 0 if rem(n,m) = n-1

rem(n, m+1) = mult ((succ(rem(n,m)) , {rem(n,m) < pred(n)}) = h(n,m, rem(n,m))

h(n,m,k) = (mult • ((succ • id3,3) ,({ id3,3 < (pred • id3,1) }) (n,m,k) = mult ((k+1),{k < n-1})

This shows that rem(n, m) is PR. We will use this function inside the composite function gcd(m, n).

gcd(m, 0) = $zero_1$(n) = 0

gcd(m, n+1) = n ; if rem(m,n+1) = 0

= gcd(n+1, rem(m,n+1)) ; if rem(n+1, m) != 0

gcd(m, n+1) = plus (mult(n , {rem(n+1, m) == 0}), mult(gcd(n+1, rem(m,n+1)), , {rem(n+1, m) == 0}))

Since gcd is a composition of primitive recursive functions and it can be recursively defined it is a primitive recursive function.

c) prime(n) = {x > 1 & (∀t) ≤ x [t = 1 ∨ t = x ∨ ∼ (t|x)].

Since y|x is a primitive recursive function prime(n) is also a PR because it is the bounded minimalization of the composite function of divisibility and predicate functions. We can show that divisibility function is a primitive recursive function like this:

y|x ⇔ (∃t) ≤ x (y · t = x)

d) p(0) = 0, p(1) = 2, p(3) = 5 etc.

We can define pn recursively:
p(0) = 0
p(n+1) = min (t ≤ p(n)! + 1) [Prime(t) & t > pn]
We used the bounded minimalization and primitive recursion to show that p(n) is a pr function.

e) We can use the bounded minimalization to show that log is primitive recursive function.
log(m, n) = $\log_m$(n)
Assume that m, n > 0
log(m, 1) = 0
log(m, n+1) = h(m, n, log(m, n))
We can use the bounded minimalization to show that log is primitive recursive function.
log(m, n+1) = min (x < n) [m^x <= n]