Patrick Austin  
CS 326

Homework # 4

10/13/2015

1. Introducing an explicit terminator has the advantage of clarity and readability- it is explicitly stated that the scope of the matching if statement is now over. When a programmer is reading with complicated nested if statements in the absence of nice blocking out of code, the clearer picture of the control flow provided by the explicit terminator might be appreciated.

2.

exp -> num | id | op exp exp

op -> + | \*

This grammar is not ambiguous. Any string that can be created in this grammar has one unique parse tree.

3. This code snippet does not produce short-circuit behavior because it uses applicative-order evaluation. Arguments a and b are evaluated before being ANDed, so short circuiting will not occur.

One could implement short circuit behavior using normal order evaluation, since in that case the an argument would only be evaluated if needed. For example, AND would only need to evaluate its second operand if the first operand was true. If the first operand was false, AND could immediately return false without evaluating the second operand, and so it would have short circuit behavior.

4.

a.

/\* Algorithm: use OR of a flag that starts out true and the condition c to assure s inside the while executes at least once, as it would in a do statement. Set the flag to false after first run, so that the while loop continuing is decided solely by c. \*/

bool flag = true;

while ( c || flag )

{

flag = false;

s;

}

b.

/\* Algorithm: don’t execute s in the do part of the do/while loop unless condition c is met. This will prevent code from executing on the first run through the loop if c is not satisfied, as would happen in a normal while loop. \*/

do

{

if ( c )

s;

}

while ( c );

c.

/\* Algorithm: don’t execute s or s2 in the do part of the do/while loop unless condition c is met. This will prevent code from executing on the first run through the loop if c is not met. If c is met, s will execute along with s2. s2 will ‘step’ closer to breaking condition c and exiting the loop. \*/

do

{

if ( c )

{

s;

s2;

}

}

while( c );

5.

; Given list L, which is assumed to be a list of numbers, returns the sum of the numbers in the

; list using tail recursion. tailsum wrapper function takes list L only, in order to hide subtotal

; parameter from the user.

( define tailsum ( lambda ( L )

; sumhelper helper function does the actual heavy lifting

( letrec ( ( sumhelper ( lambda ( L subtotal )

; if empty list is reached...

( if ( null? L )

; return the subtotal so far, which will be the final total

subtotal

; otherwise, repeat sumhelper operation using cdr of L and

; the subtotal plus the car of L. No computation follows this

; recursive call, so tail recursion is satisfied.

( sumhelper ( cdr L ) ( + subtotal ( car L ) ) ) ) ) ) )

; when calling tailsum with L, call sumhelper with L and starting subtotal of 0

( sumhelper L 0 ) ) )

6.

sumhelper ( L subtotal )

start:

if null L

return subtotal

if not null L

subtotal := subtotal + car L

L := cdr L

goto start