Bryan Kline

CS326

Homework 4

LaTex (TeXstudio)

10/18/2016

1. The if statement in Pascal has the syntax:

```
if boolean_expression then statement else statement
```

In Ada, the syntax for the if statement is:

```
if boolean_expression then statement else statement end if
```

What are the advantages of introducing an explicit terminator, such as the 'end if' in Ada?

The advantages of an explicit terminator are that with a terminator it's very clear what the syntax and semantics will be so that someone reading it will understand it and the person writing it will write it in the way they intend. If there is no terminator then it's possible that the end of last statement or expression and the beginning of the next are ambiguous. This may simply cause a syntactic error, in which case it would be easy to find and correct, or it may cause a semantic or logical error which could be hard to find.

2. Write a grammar that describes arithmetic expressions in prefix notation, where possible operators are + and *, and possible operands are numbers or identifiers. You do not need to specify the internal structure of numbers and identifiers – assume that they are returned by the scanner as terminal symbols nr and id. Also assume that each operator takes exactly two operands. Is your grammar ambiguous? Why?

$Expr \rightarrow operator\ (Expr|operand)\ (Expr|operand)$
$operator \rightarrow +|*$
$operand \rightarrow nr|id$

The grammar is not ambiguous because there are only two operators in the grammar that both only take two operands and because prefix notation is not ambiguous. Each operator takes two operands and so whatever follows an operator will be two operands, even if they are expressions themselves, they will be evaluated and operated on in the correct order.

3. Suppose that we try to write a short-circuit version of and (with two operands) in C as:

```
int sc_and (int a, int b)
{
    return a ?  b :  0;
}
```

Explain why this does not produce a short-circuit behavior. Would it work if normal-order evaluation were used? Why?

This does not produce a short-circuit AND because it's a function and so while the behavior is much like short-circuiting, the arguments are passed by value and so copies are made of them and so in a sense they are evaluated and have to have a place made for them on the stack. The advantage of short-circuiting would be that, in the case where operands are large, expensive structures for example, if the latter operand weren't evaluated then there would a savings in terms of time and memory, but here they are still passed into the function and so take up computational resources. If it were normal-order it still wouldn't be short-circuiting and woulnd't even seem to be, as the above function does.

4. In the C programming language:

a) Show how to simulate a do statement (shown below) with a while statement.

```
do
    s;
while(c);
```

To simulate a `do while` with a `while` there only needs to be the block preceding the loop:

```
s;
while(c)
    s;
```

b) Show how to simulate a while statement (shown below) with a do statement.

```
while(c)
    s;
```

To simulate a `while` with a `do while` there only needs to be a conditional statement in the loop:

```
do
    if(c)
        s;
while(c);
```

c) Show how to simulate a for statement (shown below) with a do statement.

```
for(s1;c;s2)
    s;
```

To simulate a `for` with a `do while` the counter needs to be initialize outside the loop and then a conditional inside the loop and the counter also needs to be incremented in the loop.

```
s1;
do
    if(c)
        s;
        s2;
while(c);
```

5. Using the Scheme programming language, write a tail-recursive function that returns the sum of all elements in a list of numbers. You will probably want to also define a "helper" function, as shown in Section 6.6.1 of the textbook.

NOTE: No helper function was needed to implement the tail-recursive function for calculating the sum of a list. The following is tail-recursive and sums the elements of a list.

```
(define (sum L)
        (cond
        ((null? (cdr L)) (car L))
        (else (+ (car L) (sum (cdr L))))))
```

6. (Extra Credit) Show (in low-level pseudo-code, as illustrated in the textbook) what would be the target code generated for the tail-recursive function from problem 5.

NOTE: I'm not sure exactly what the syntax for the target code should be or exactly what low-level pseudo-code from the textbook this question is referring to, but I will use the target code from the lecture slides on tail-recursion.

```
sum(L)
    r1 = (car L)
    r2 = (cdr L)
    start:
        if(r2 == null)
            return r1
        else
            r1 = r1 + (car r2)
            r2 = (cdr r2)
            goto start
```