Lindstrom scanning and link inversion

Dick Bruin and Walter A. Kosters
Leiden University, Department of Computer Science
P.O. Box 9512, 2300 RA Leiden, The Netherlands
Email: kosters@wi.leidenuniv.nl

In this short note we present a derivation (with implicit correctness proof) of Lindstrom scanning of binary trees, starting from simple specifications of tree traversals. In a similar way the link inversion algorithm can and will be derived. As general references we mention [1] and [2].

Binary trees are defined by

```
Tree ::= "nil" | "t(" Tree ",n(" Name "," Mark ")," Tree ")"   ,
```

where `Mark` is an integer and `Name` represents the name of the node. A root-left-root-right-root traversal of such a tree is generated by

```
Visit( nil ) = [ ] ,

Visit( t(L,n(r,0),R) ) = [r,0] + Visit( L ) + [r,1]
                               + Visit( R ) + [r,2]   ,
```

where —for the moment— we assume that initially all nodes contain `0`. Here the symbol `+` denotes concatenation of lists.
For trees `S` and `T`, and lists `v`, we define

```
Lindstrom( S,T,v ) = v + Visit( S ) + Visit( T )   .
```

Notice that

```
Lindstrom( nil,nil,v ) = v   ,

Lindstrom( nil,T,v ) = Lindstrom( T,nil,v )   .
```

Now we compute

```
Lindstrom( t(L,n(r,0),R),T,v )

    = v + [r,0] + Visit( L ) + [r,1]
                + Visit( R ) + [r,2] + Visit( T )

    = Lindstrom( L,t(R,n(r,1),T),v+[r,0] )   ,
```

if we define

```
Visit( t(L,n(r,1),R) ) = [r,1] + Visit( L ) + [r,2] + Visit( R )   .
```

Notice that initially `Visit` was only defined for a tree with root containing `0`. Proceeding as above we get, for `x` in `{0,1,2}`,

```
Lindstrom( t(L,n(r,x),R),T,v )
    = Lindstrom( L,t(R,n(r,x+1),T),v+[r,x] )   ,
```

where we defined

```
Visit( t(L,n(r,2),R) ) = [r,2] + Visit( L ) + Visit( R )   ,

Visit( t(L,n(r,3),R) ) = Visit( L ) + Visit( R )   .
```

Finally we have

```
Lindstrom( t(L,n(r,3),R),T,v ) = v + Visit( L )
                                   + Visit( R ) + Visit( T )   .
```

In order to clearify this "halting condition", and also for showing similarity to the usual Lindstrom scanning, we state

Theorem

Suppose that a tree `S` initially has only zeroes in its `Mark` fields. Let `T` be an arbitrary tree and `v` an arbitrary list. Then the computation of `Lindstrom( S,T,v )` reaches `Lindstrom( T,Three(S),v+Visit(S) )`, where `Three` is defined by

```
Three( nil ) = nil   ,

Three( t(L,n(r,0),R) ) = t(Three(L),n(r,3),Three(R))   .
```

Proof

The proof of the theorem is by induction on `S`, the case `S = nil` being trivial. So we let `S = t(L,n(r,0),R)`, and assuming the truth of the theorem for `L` and `R` we get

```
Lindstrom( t(L,n(r,0),R),T,v )

    = Lindstrom( L,t(R,n(r,1),T),v+[r,0] )

    = Lindstrom( t(R,n(r,1),T),Three( L ),v+[r,0]+Visit( L ) )

    = Lindstrom( R,t(T,n(r,2),Three( L )),v+[r,0]+Visit( L )+[r,1] )

    = Lindstrom( t(T,n(r,2),Three( L )),Three( R ),
                 v+[r,0]+Visit( L )+[r,1]+Visit( R ) )

    = Lindstrom( T,t(Three( L ),n(r,3),Three( R )),
                 v+[r,0]+Visit( L )+[r,1]+Visit( R )+[r,2] )

    = Lindstrom( T,Three( S ),v+Visit( S ) )   .
```

As a consequence we have

Corollary

Suppose that a tree `S` initially has only zeroes in its `Mark` fields. Let `T*` be either `nil` or `t(nil,n(special,3),nil)`. Then the computation of `Lindstrom( S,T*,[ ] )` reaches `Lindstrom( T*,Three( S ),Visit( S ) )` and in this case the "halting condition" may be replaced with

```
Lindstrom( t(L,n(r,3),R),T,v ) = v  .
```

Notice that `Lindstrom` does not destroy the original tree structure; it only changes all zeroes into threes (this follows from the Theorem). It is also possible to drop all marking, introducing an explicit "halting condition" by means of `T*`. This leads to the following more familiar self-explaining program:

```
if ( Root <> NIL ) then
   New(Star);
   Present, Previous := Root, Star;
   while ( Present <> Star ) do
      if ( Present = NIL ) then
         Present, Previous := Previous, Present fi;
      Present, Present->Left, Present->Right, Previous :=
         Present->Left, Present->Right, Previous, Present;
   od;
fi;
```

In a similar way one can produce the link inversion algorithm. The only difference is that, instead of the original definition of `Visit( t(L,n(r,1),R) )`, we start from

```
Visit( t(L,n(r,1),R) ) = [r,1] + Visit( R ) + [r,2] + Visit( L )  .
```

In order to get the usual link inversion algorithm some computations are necessary, for instance

```
LinkInversion( nil,t(L,n(r,1),R),v )
    = LinkInversion( t(nil,n(r,1),R),L,v )  ,
```

giving a link inversion analogue of one of the equations above, for this choice of the second argument. It also appears that we now get

```
LinkInversion( t(L,n(r,x),R),T,v )
    = LinkInversion( A,t(B,n(r,x+1),C),v+[r,x] )  ,
```

where `(A,B,C)` is either `(L,T,R)`, `(R,L,T)` or `(L,R,T)`, corresponding with either `x = 0`, `x = 1` or `x = 2`.
So in this case the `Mark` fields are necessary. However, it appears that one bit per node is sufficient (using one global variable).

References

[1] D. Gries, The science of programming, Springer-Verlag, New York, 1981.

[2] T.A. Standish, Data structure techniques, Addison-Wesley, Reading, 1980.

Leiden, November 1987.