Homework 4

Abhi Agarwal

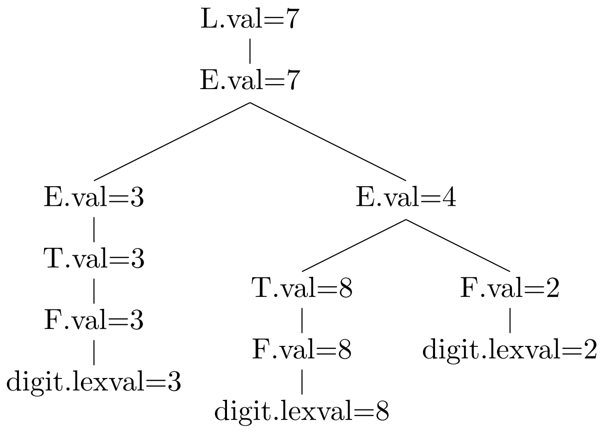
1 Syntax-Directed Definitions

1.1 Question 1.1

1.1.1 Extend the SDD to handle division ("/").

Production	Semantic Rule
L $\rightarrow E_1$ \$	L.val = E_1.val
E $\rightarrow E_1 + T_2$	E.val = E_1.val + T_2.val
E $\rightarrow T_1$	E.val = T_1.val
T $\rightarrow T_1 * F_2$	T.val = T1.val F2.val
T $\rightarrow T_1 \ / \ F_2$	T.val = T_1.val / F_2.val
T $\rightarrow F_1$	T.val = F_1.val
F $\rightarrow (E_1)$	F.val = E_1.val
F $\rightarrow$ digit	F.val = digit.lexval

1.1.2 Draw the annotated parse tree for 3 + 8/2 \$.



1.2 Question 1.2

S-attributed and/or L-attributed

1.2.1 For synthesized attribute s, and inherited attribute i:

These rules are circular - it is not possible to evaluate either A.s node or B.i without evaluating the other and so we will see if they are S-attributed below.

S-attributed: The rules are S-attributed are that an SDD is S-attributed if every attribute is synthesized, and in this example this is not the case. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself, but in the case of $A.s$ we are not seeing that. It's attribute values are referencing the ones of its parents, and so we rule out the fact that it can be S-attributed. **No** they are not S-attributed. The first rule defines an inherited attribute so they are automatically not S-attributed.

L-attributed: There is a cycle in the dependency graph here because we are referencing the first rule within the second rule, therefore it can't be L-attributed. Also the fact that A.s is to the right of C.s. **No** it is not L-attributed.

1.2.2 For synthesized attribute z and given an externally defined scheme IfZero:

S-attributed: There are no inherited attributes. An SDD is S-attributed if every attribute is synthesized. Therefore, **yes** this one is S-attributed.

L-attributed: Each attribute is synthesized as it is S-attributed. Nothing is inherited. We have to see if there are cycles in a dependency graph formed by these attributes in order to check if it is L-attributed. There seem to be no cycles in the dependency graph and therefore **yes** it is L-attributed.

1.2.3 For inherited attribute d and a synthesized attribute n:

S-attributed: **No** because this defines an inherited attribute F_1.d so the entire SDD cannot be S-attributed. Not everything is synthesized here.

L-attributed: **Yes** it has elements which are either synthesized and follow the rules for proper inherited attributes, and there does not contain any cycles within it.

2 Syntax Directed Translation in Hacs

2.1 Question 2.1

Output is T & ! (F | T)

This is because the bool.hx file we've passed into hacs is missing how to interpret the OR statement. The translation does not understand how to interpret | and therefore it has left it as it was and outputted it in the same way to the user. It has recognized that there's an overuse of the brackets and therefore it has removed a pair of them. Because this is a AND statement the code can't interpret how to proceed and therefore just caries on. It also has left the NOT operation because it doesn't know what the | and interprets it as something that can't be evaluated.

2.2 Question 2.2

Here we fix the issue and we introduce the | (OR) statement.

```
root@debian:/home/abhi/Desktop/compiler# ˙Bool.run –sort=B –action=Evaluate –term='((T)&!(F|T))'
F
```

Below is the code that I wrote for it. Sorry I couldn't copy paste it in.

```
// Syntax.
sort B
| ⟦ T ⟧@4 | ⟦ F ⟧@4                  // constants
| sugar ⟦ ( ⟨B#⟩ ) ⟧@4 → B#         // parenthesis
| ⟦ ⟨B@2⟩ | ⟨B@1⟩ ⟧@1                // disjunction
| ⟦ ⟨B@3⟩ & ⟨B@2⟩ ⟧@2                // conjunction
| ⟦ ! ⟨B@3⟩ ⟧@3                      // negation
;

// Semantic operations.
sort B;
| scheme And(B,B) ;
And(⟦T⟧, #2) → #2 ;
And(⟦F⟧, #2) → ⟦F⟧ ;

| scheme Or(B,B) ;
Or(⟦T⟧, #2) → ⟦T⟧ ;
Or(⟦F⟧, #2) → #2 ;

| scheme Not(B) ;
Not(⟦F⟧) → ⟦T⟧ ;
Not(⟦T⟧) → ⟦F⟧ ;

// Propagation.
attribute ↑b(B);
sort B;
↑b;        // associate b attribute to (current) E sort

// Evaluation scheme
sort B;
| scheme Evaluate(B) ;
Evaluate(B# ↑b(#b)) → #b ;

⟦ ⟨B#1 ↑b(#b1)⟩ & ⟨B#2 ↑b(#b2)⟩ ⟧ ↑b( And(#b1, #b2) ) ;
⟦ ⟨B#1 ↑b(#b1)⟩ | ⟨B#2 ↑b(#b2)⟩ ⟧ ↑b( Or(#b1, #b2) ) ;
⟦ ! ⟨B# ↑b(#b)⟩ ⟧ ↑b( Not(#b) ) ;
⟦ T ⟧ ↑b(⟦T⟧);
⟦ F ⟧ ↑b(⟦F⟧);
```