August 21, 2020

# CS360 QUIZ 5

Group members: Shivanshi Nagar, Stephen Hansen, Dennis George

Start Time: 1:30 PM
End Time: 3:00 PM

**Question 1**

**Trace execution of Prolog gcd(33,24,X). Indicate each step of usage of resolution and unification.**

```
gcd(u, 0, u).
gcd(u, v, w) ← not zero(v), gcd(v, u mod v, w).
```

Goal: <- gcd(33,24,X)

Resolution fails with the first clause (24 does not match 0). Use the second clause and unify:
gcd(33,24,X) <- not zero(24), gcd(24, 33 mod 24, X), gcd(33,24,X)
If zero(24) is false, not zero(24) is true
Simplify 33 mod 24 to 9, cancel gcd(33,24,X) from both sides giving:
<- gcd(24,9,X)
Resolution fails with the first clause (9 does not match 0). Use the second clause and unify:
gcd(24,9,X) <- not zero(9), gcd(9, 24 mod 9, X), gcd(24,9,X)
If zero(9) is false, not zero(9) is true
Simplify 24 mod 9 to 6, cancel gcd(24,9,X) from both sides giving:
<- gcd(9, 6, X)
Resolution fails with the first clause (6 does not match 0). Use the second clause and unify:
gcd(9,6,X) <- not zero(6), gcd(6, 9 mod 6, X), gcd(9,6,X)
If zero(6) is false, not zero(6) is true
Simplify 9 mod 6 to 3, cancel gcd(9,6,X) from both sides giving:
<- gcd(6,3,X)
Resolution fails with the first clause (3 does not match 0). Use the second clause and unify:
gcd(6,3,X) <- not zero(3), gcd(3, 6 mod 3, X), gcd(6,3,X)
If zero(3) is false, not zero(3) is true
Simplify 6 mod 3 to 0, cancel gcd(6,3,X) from both sides giving:
<- gcd(3,0,X)
This matches the first rule (resolves, i.e. resolution does not fail, as 0 = 0), so setting X to 3 gives the empty statement.
gcd of 33 and 24 is 3.

**Question 2**

**Take expressions $p \to q$ and $p \to r$ as hypotheses and prove the formula $p \to qr$ by deduction.**

1. $p \to q$ [ Hypothesis ]
2. $p \to r$ [ Hypothesis ]
3. $p \to q \equiv \overline{p} + q$ [ Tautology for line 1 (inference rule (a)) ]
4. $\overline{p} + q$ [ By inference rule (d) for lines 1, 3 ]
5. $p \to r \equiv \overline{p} + r$ [ Tautology for line 2 (inference rule (a)) ]
6. $\overline{p} + r$ [ By inference rule (d) for lines 2, 5 ]
7. $(\overline{p} + q)(\overline{p} + r)$ [ By inference rule (c) for lines 4, 6 ]
8. $(\overline{p} + q)(\overline{p} + r) \equiv \overline{p} + qr$ [ Tautology for line 7 (inference rule (a)) ]
9. $\overline{p} + qr$ [ By inference rule (d) for lines 7, 8 ]
10. $\overline{p} + qr \equiv p \to qr$ [ Tautology for line 9 (inference rule (a)) ]
11. $p \to qr$ [ By inference rule (d) for lines 9, 10 ]

Hence by deduction given hypotheses $p \to q$ and $p \to r$ we prove the conclusion $p \to qr$.

**Question 3.**
**Explain the principal components of the semantics of grammars of PLP Figures 4.5, 4.6. What are action routines? Explain their usage in the process of constructing attribute parse trees.**

The attribute grammars in Figures 4.5 and 4.6 hold neither numeric values nor target code fragments as the attributes. Instead, the attributes of these grammars point to nodes of a syntax tree. *make_leaf* returns a pointer to a newly allocated syntax tree node containing the value of a constant. *make_un_op* and *make_bin_op* return pointers to newly allocated syntax tree nodes containing a unary or binary operator, respectively, and pointers to the supplied operands. Figure 4.5 is a grammar which constructs a syntax tree for basic calculator expressions using a bottom-up S-attributed grammar. Figure 4.6 is a grammar which constructs a syntax tree for basic calculator expressions using a top-down L-attributed grammar.

An action routine is a semantic function that the programmer instructs the compiler to execute at a particular point in the parse. When the parser finds a pointer to an action routine at the top of the stack, the parser will simply call the function passing it the appropriate attributes as arguments.

In the example grammar for Figure 4.5, constants are defined by an action routine to make a leaf node for constant values. This routine is passed up the parse as a function pointer. Negation statements have their own action routine which takes a given function pointer and run a negation on it (passing up the function as a pointer) via *make_un_op*. Similarly addition, subtraction, multiplication, and division each have associated action routines to *make_bin_op* which build the associated operation symbol into the tree and contain pointers to derived functions. The end result is that the function construction via action routines matches the construction of the parse tree and the tree may be built while the grammar is parsed.

Similarly for Figure 4.6 the same action rules are used, the only difference from Figure 4.5 being the manner in which the grammar is parsed. Rather than working bottom-up (starting with building constant leaves and working pointers up), it starts at the highest level and continuously updates function pointers located higher in the tree to point to other functions. The end result is that both grammars use action routines to construct the syntax tree during the parse.

**Question 4**

**Design a context-free grammar for polynomials in $x$. Add semantic functions to produce an attribute grammar that will store the outcome of multiplication by $x$ (as a string) in a synthesized attribute of the root of the parse tree.**

**Hint: Modify the attribute grammar of PLP Exercise 4.17 (Quiz 5 review problem).**

```
P -> T more_Ts
        more_Ts.st = T.m                P.m = more_Ts.m
T-> num T_tail
        T_tail.c = num.v                T.m = T_tail.m
T_tail -> x exp
        exp.c = T_tail.c                T_tail.m = exp.m
T_tail -> epsilon
        T_tail.m = T_tail.c + “x”
exp -> ** num
        exp.m = float_to_string(exp.c) + “x **” + int_to_string(num.v + 1)
exp -> epsilon
        exp.m = float_to_string(exp.c) + “x ** 2”
more_Ts1 -> + T more_Ts2
        more_Ts2.st = more_Ts1.st + “+” + T.m                more_Ts1.m = more_Ts2.m
more_Ts -> epsilon
        more_Ts.m = more_Ts.st
```

This grammar follows the attribute grammar of PLP Exercise 4.17 with some small changes. The grammar in 4.17 accumulated the derivative of the polynomial as a string. This grammar makes alterations to instead accumulate the polynomial multiplied by x as a string in the root of the parse tree. The changes, in particular, involved the following. Constants are adjusted to have an additional “x” term (see T_tail -> epsilon) in the string representation (rather than encode to the empty string, which was the derivative). Given a constant multiplied by x to some power, the constant and x remain but the power is increased by 1 for the additional multiplied x. If there is an x term with no associated power (exp -> epsilon), the string encoding is the constant times x squared to raise x by 1 power.