Sample Problems – Final Examination

Item Sets and DFA Construction: LR(0), LR(1), and LALR(1)

1. As the initial step in the design of an SLR parser, construct the DFA that corresponds to the LR(0) item sets for the grammar. Make sure that you clearly identify the items in each set.

   \[
   \begin{align*}
   S & \rightarrow A \ B \\
   A & \rightarrow S \ y \ | \ x \\
   B & \rightarrow y \ S 
   \end{align*}
   \]

   Is the grammar SLR? Why or why not? Reason out your answer without constructing the entire parsing table!

2. Using the grammar in Problem 1, compute the LR(1) item sets. Is the grammar LR(1)? Why or why not? Reason out your answer without constructing the entire parsing table!

3. Using your results from Problem 2, identify the LR(1) item sets that are merged to arrive at the LALR(1) item sets. Is the grammar LALR(1)? Why or why not? Reason out your answer without constructing the entire parsing table!

4. As the initial step in the design of an SLR parser, construct the DFA that corresponds to the LR(0) item sets for the grammar. Make sure that you clearly identify the items in each set.

   \[
   \begin{align*}
   S & \rightarrow d \ D \ d \\
   S & \rightarrow e \ D \ e \\
   S & \rightarrow d \ E \ e \\
   S & \rightarrow e \ E \ d \\
   D & \rightarrow f \\
   E & \rightarrow f 
   \end{align*}
   \]

   Is the grammar SLR? Why or why not? Reason out your answer without constructing the entire parsing table!

5. Using the grammar in Problem 4, compute the LR(1) item sets. Is the grammar LR(1)? Why or why not? Reason out your answer without constructing the entire parsing table!

6. Using your results from Problem 5, identify the LR(1) item sets that are merged to arrive at the LALR(1) item sets. Is the grammar LALR(1)? Why or why not? Reason out your answer without constructing the entire parsing table!
Attribute Grammars

Expressions in the Lisp functional programming language are formed using nested parentheses and operators which apply to exactly two operands. Below are some example Lisp expressions with their infix and postfix equivalents:

\[
\begin{align*}
(= x (+ a (* b 5))) & \quad x = (a+(b*5)) & b 5 * a + x = \\
(= y (- (* d (+ a 3)) (/ 8 d))) & \quad y = (((a+3)*d)-(8/d)) & a 3 + d * 8 d / - y =
\end{align*}
\]

For Lisp expressions, assume that multiplication and division have equal precedence, which is higher than addition and subtraction (which also have equal precedence).

7. Consider the grammar for infix expressions given below.

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T * F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow ( E ) \\
F & \rightarrow id \\
F & \rightarrow int
\end{align*}
\]

Write an attribute grammar the converts expressions to their Lisp equivalents. For \( a + b * c \), the Lisp equivalent is \( ( + a ( * b c ) ) \), where there is one operator and two operands within each (...). Show that your attribute grammar works for the expression \( (a + b) * d - (c / d) \). Clearly name your attributes and identify each attribute as either inherited or synthesized.

8. Consider the grammar for Lisp expressions:

\[
\begin{align*}
E & \rightarrow ( = id L ) \\
L & \rightarrow ( A L1 L1 ) \\
L & \rightarrow ( M L1 L1 ) \\
L1 & \rightarrow int | id | L \\
A & \rightarrow + | - \\
M & \rightarrow * | / 
\end{align*}
\]

a. Write an attribute grammar that converts Lisp expressions into their infix equivalents. That is, while parsing a Lisp expression, the grammar should generate the infix equivalent shown above. Demonstrate that your attribute grammar works by evaluating the grammar for the Lisp expression: \( ( = y (- ( * d ( + a 3 ) ) ( / 8 d ) ) ) \). (Hint: Notice that for the infix expression, I've included a complete set of parentheses to indicate the grouping of the operands. This should make the translation easier than only having to put in parentheses some of the time.)

b. Write an attribute grammar that converts Lisp expressions into their infix equivalents. That is, while parsing a Lisp expression, the grammar should generate the infix equivalent shown above. Demonstrate that your attribute grammar works by evaluating the grammar for the Lisp expression: \( ( = y (- ( * d ( + a 3 ) ) ( / 8 d ) ) ) \). (Hint: Notice that for the infix expression, I've included a complete set of parentheses to indicate the grouping of the operands. This should make the translation easier than only having to put in parentheses some of the time.)
9. Optimize the flow graph which is shown below. Indicate which kinds of optimization you are utilizing at each step of the optimization process. Lastly, clearly show the final optimized version of your flow graph and include a synopsis of the number of operations that were saved (e.g., 6 mults, 3 array accesses, 3 assignments, etc.) in each basic block and for the entire flow graph.
Sample Problems – Final Examination

10. Make sure that you can generate intermediate code for the following two attribute grammars from slides 25 and slides 26 of Chapter 8. In the attribute grammars, the variable `newstat` keeps track of statement numbers and is initialized to 100. Each call to the `emit` function also prints out the current statement number (from `newstat`) and then increments the variable `newstat` by one (1). Just try a few examples.

### Revised Attribute Grammar with Emit

```
S → id := E { p := lookup(id.name);
    if p ≠ nil then
        emit(p := 'E.place')
    else error }

E → E₁ + E₂ { E.place := newtemp;
    emit(E.place := 'E₁.place + E₂.place) }

E → E₁ · E₂ { E.place := newtemp;
    emit(E.place := 'E₁.place * E₂.place) }

E → ¬ E₁ { E.place := newtemp;
    emit(E.place := '¬E₁.place) }

E → ( E₁ ) { E.place := E₁.place }

E → id { p := lookup(id.name);
    if p ≠ nil then
        E.place := p
    else error }
```

### Attribute Grammar for Boolean Expressions

```
E → E₁ or E₂ { E.place := newtemp;
    emit(E.place := 'E₁.place + E₂.place) }

E → E₁ and E₂ { E.place := newtemp;
    emit(E.place := 'E₁.place * E₂.place) }

E → ¬ E₁ { E.place := newtemp;
    emit(E.place := '¬E₁.place) }

E → ( E₁ ) { E.place := E₁.place }

E → id₁, id₂ { E.place := newtemp;
    emit(id₁.id place relop.id₂.place 'goto' nextstat + 3);
    emit(E.place := '0');
    emit('goto' nextstat + 2);
    emit(E.place := '1') }

E → true { E.place := newtemp;
    emit(E.place := '1') }

E → false { E.place := newtemp;
    emit(E.place := '0') }
```