Problem 1: Context-Free-Grammars and Parsing Algorithms [40 points]

Consider the CFG with non-terminal symbols N={S, E, A}, with start symbol S, terminal symbols T={ id, ';' , '=' } and the productions P listed below.

(1) S → E $
(2) E → E ';$ A
(3) E → A '
(4) A → id
(5) A → A '=' A

Questions:

a) [10 points] As specified, can this grammar be parsed using a predictive LL algorithm? Why or why not?
b) [20 points] Compute the DFA that recognizes the LR(0) sets of items for this grammar and construct the corresponding LR(0) parsing table. Comment on the nature of the conflicts, if any.
c) [10 points] How different would the SLR table look like? If there were conflicts in the original table will this table construction algorithm resolve them?

Solution:

a) This grammar is clearly not LL(1) as it is left-associative as shown by productions 2, 3 and 4 of the grammar.

b) We start by augmenting the grammar with an initial production (1) S → E $ and compute the set of LR(0) items as depicted below for a total of 9 states.

![DFA Diagram]
Based on this DFA of states as sets of LR(0) items, we can construct the LR(0) parsing table below.

<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift s5</td>
<td>goto s2 goto s1</td>
</tr>
<tr>
<td>s1</td>
<td>shift s8 Accept</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>shift s4 shift s3</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>reduce r3 reduce r3 reduce r3 reduce r3 reduce r3</td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>shift s5 goto s6</td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>reduce r4 reduce r4 reduce r4 reduce r4 reduce r4</td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td>reduce r5 reduce s4 reduce r5 reduce r5 reduce r5</td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td>shift s5 goto s7</td>
<td></td>
</tr>
<tr>
<td>s8</td>
<td>reduce r2 reduce r2 reduce r2 reduce r2 reduce r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this case one can attempt to resolve the conflicts in the LR(0) parsing table using the FOLLOW set of A and E. Unfortunately, FOLLOW(A) = { ‘$’, ‘;’, ‘=’ } and FOLLOW(E) = { $, ‘;’ } and as such there will still be a ‘shift/reduce’ conflict for state s6 on input ‘=’ as shown below.

<table>
<thead>
<tr>
<th>STATE</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift s5</td>
<td>goto s2 goto s1</td>
</tr>
<tr>
<td>s1</td>
<td>shift s8 Accept</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>shift s4 shift s3</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>reduce r3 reduce r3 reduce r3 reduce r3 reduce r3</td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>shift s5 goto s6</td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>reduce r4 reduce r4 reduce r4 reduce r4 reduce r4</td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td>reduce r5 reduce s4 reduce r5 reduce r5 reduce r5</td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td>shift s5 goto s7</td>
<td></td>
</tr>
<tr>
<td>s8</td>
<td>shift s4 reduce r2 reduce r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 2: Syntax-Directed Translation [20 points]

Consider the CFG G whose productions P are listed below with start symbol S, non-terminal symbols NT = {S, L, B} and terminal symbols T = {'0', '1', '.'}. This grammar captures the set of strings over the binary alphabet \( \Sigma = \{0, 1\} \) with an integer and a fractional part.

\[
\begin{align*}
(1) & \quad S \rightarrow L \cdot \cdot L \\
(2) & \quad S \rightarrow L \\
(3) & \quad L \rightarrow L B \\
(4) & \quad L \rightarrow B \\
(5) & \quad B \rightarrow '0' \\
(6) & \quad B \rightarrow '1'
\end{align*}
\]

In this problem you are asked to develop an attributive grammars based on G to compute the decimal-number value of the attribute `val` of S and check if the value is greater than a specific input threshold decimal value. For instance, if the input is “101.001” and the input threshold is 5, then the “output” to be captured as an attribute of S, should be true.

**Note that you are not supposed to modify the grammar.**

**Solution:**

In this solution we have 3 inherited and 2 synthesized attributes, namely \{side, depth\} and \{val, length\}. The table below summarizes the type and associated non-terminal symbols for each attribute.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Non-Terminal Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>side</td>
<td>enumerated: {L, R}</td>
<td>L</td>
</tr>
<tr>
<td>val</td>
<td>real</td>
<td>S, L, B</td>
</tr>
<tr>
<td>length</td>
<td>integer</td>
<td>L</td>
</tr>
<tr>
<td>depth</td>
<td>integer</td>
<td>L</td>
</tr>
<tr>
<td>threshold</td>
<td>real</td>
<td>S</td>
</tr>
<tr>
<td>check</td>
<td>bool</td>
<td>S</td>
</tr>
</tbody>
</table>

The following are the productions and the corresponding rules for the L-attributed grammar.

\[
\begin{align*}
S \rightarrow L_1 \cdot \cdot L_2 & \quad || \quad L_{1, \text{side}} = R; \quad L_{1, \text{depth}} = 0; \\
& \quad || \quad L_{2, \text{side}} = L; \quad L_{2, \text{depth}} = 0; \\
& \quad || \quad S.\text{val} = L_{1, \text{val}} + L_{2, \text{val}} \cdot 2^(-L_{2, \text{depth}}) \\
& \quad || \quad \text{if (S.val} > S.\text{threshold)} \quad S.\text{check} = \text{true}; \quad \text{else} \quad S.\text{check} = \text{false} \\
S \rightarrow L & \quad || \quad L.\text{side} = R; \quad L_{1, \text{depth}} = 0; \\
& \quad || \quad S.\text{val} = L_{1, \text{val}} \\
L \rightarrow B & \quad || \quad L.\text{val} = B.\text{val} \cdot 2^{(L_{-\text{depth}})} \\
& \quad || \quad L.\text{length} = 1;
\end{align*}
\]
Using this L-attributed grammar we depicted below the set of values for the various attributes for the parse tree for the input string “101.001”.

A possible S-attributed grammar requires that all the attributes be synthesized. On the right side of the parse three, that is for the integer part of the binary number representation, a possible approach is to propagate the value of the bits seen so far along the tree and multiply them by 2 at each level and then adding the value for the bit at the current level. Cumulatively, at the top you will get the correct amount. The fractional part relies on the same approach but you need to propagate up the length of this fractional section and then multiply the entire cumulative value by a negative power of two. The S-attributed grammar below depicts this “solution”.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Non-Terminal Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>real</td>
<td>S, L, B</td>
</tr>
<tr>
<td>length</td>
<td>integer</td>
<td>L</td>
</tr>
<tr>
<td>threshold</td>
<td>real</td>
<td>S</td>
</tr>
<tr>
<td>check</td>
<td>boolean</td>
<td>S</td>
</tr>
</tbody>
</table>

The following are the productions and the corresponding rules for the S-attributed grammar.

\[
S \rightarrow L_1 \cdot \cdot L_2 \quad \text{||} \quad S.\text{val} = L_1.\text{val} + L_2.\text{val} \cdot 2^{(L_0.\text{length})}
\]

\[
S \rightarrow L \quad \text{||} \quad S.\text{val} = L_1.\text{val}
\]
As you might have noticed, this solution has a flaw exemplified by the input string “101.100”. The length of the fractional part is computed as 3 and as a result the value computed using the S-attributed “solution” described above is 5.125 rather than the correct value of 5.5. A correct solution needs to capture the “effective” length of the fractional part or the position of the last non-zero bit.

As such we keep two counters, the length of the string (as before) and the position-of-the-last-non-zero bit ("plnz") that is set to be the value of length when at the corresponding position of the bit, the bit value is 1. The revised solution below captures this and as you can check, it produces the correct result for the input string “101.100” as effectively all 0 trailing bits are ignored.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Non-Terminal Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>real</td>
<td>S, L, B</td>
</tr>
<tr>
<td>length</td>
<td>integer</td>
<td>L</td>
</tr>
<tr>
<td>plnz</td>
<td>integer</td>
<td>L</td>
</tr>
</tbody>
</table>

The following are the productions and the corresponding rules for this revised S-attributed grammar:

```
S → L₁ ‘.’ L₂  || S.val = L₁.val + L₂.val * 2^(-L₂.plnz)
S → L          || S.val = L₁.val
L → B          || L.val = B.val
                || if (B.val == 1) then
                ||  L.plnz = 1
                || else
                ||  L.plnz = 0;
                || L.length = 1;
L₀ → L₁ B      || L₀.length = L₁.length + 1;
                || if (B.val == 1) then
                ||  L₀.plnz = L₀.length
                || else
                ||  L₀.plnz = L₁.plnz;
                || L₀.length = 2 * L₁.val + B.val;
B → ‘0’        || B.val = 0
B → ‘1’        || B.val = 1
```

Below we depict two parse trees for two inputs, respectively “101.001” and “100.100” showing that the S-attributed grammar correctly computed the desired value for the val attribute of the goal symbol of the grammar.
Problem 3. SSA Representation [20 points]

For the sequence of instructions shown below depict an SSA-form representation (as there could be more than one). Do not forget to include the \( \phi \)-functions. Also discuss what sort of optimizations could be exploited in this specific case, meaning, using the information about which definitions reach which uses and derive specific variable values.

\begin{verbatim}
  b = 0;
  d = 0;
  a = 1;
  i = ...;
Lloop: if(i > 0){
    if(a < 0){
      b = i;
    } else {
      b = 0;
    }
    i = i - 1;
    if(i < 0) {
      i = 0;
      goto Lbreak;
    } else {
      a = a + d;
    }
    goto Lloop;
}
Lbreak: x = a;
        y = b;
\end{verbatim}

Solution:

\begin{verbatim}
  b_i = 0;
  d_i = 0;
  a_i = 1;
  i_i = ...;
Lloop: i_i = \phi(i, i_i)
  a_i = \phi(a, a_i)
  b_i = \phi(b, b_i)
if(i_i > 0){
  if(a_i < 0){
    b_{i_i} = i_i;
  } else {
    b_{i_i} = 0;
  }
  b_i = \phi(b_i, b_{i_i})
  i_i = i_i - 1;
  if(i_i < 0) {
    i_i = 0;
    goto Lbreak;
  } else {
    a_i = a_i + d_i;
  }
  goto Lloop;
}
Lbreak: x = a_i;
        y = b_i;
\end{verbatim}

In this specific case, the assignment “\(a = a + d\)" always used the very first definition of the variable “\(a\)" as the zero value. As such this assignment is in fact void, as the variable \(a\)’s value remains unchanged. As a consequence, the variable “\(a\)" always retains the same value for all the code’s execution, that is 1. Lastly, the assignment at the end of the code to the x variable thus assumes the 1 value as well.
Problem 4: Intermediate Code Generation [20 points]

In class we covered intermediate code generation for a variety of high-level programming constructs such as boolean expressions, assignments and loop constructs using three-address instructions. In this problem you are asked to use the SDT schemes presented in class and show the generated code for the snippet of code below using the short-circuit scheme for the code generation of predicates. For this code, suggest an improvement that will result in fewer instructions being executed.

State the assumption you make and use symbolic labels for targets of jump as you see fit and draw a memory layout of the code.

```c
i = 0;
while ((a < b) and (i < 10)) {
  if (c[i] < 0) {
    break;
  }
  i = i + 1;
}
```

Solution:
The layout of this code is fairly simple as suggested in one of the lectures about control-flow. At the top we will have an if statement along with a loop label followed by the body of the while-loop. At the bottom of the loop there is an unconditional jump to the test section of the loop’s header. In terms of code generation, we assume there is a symbolic label denoting where the control flows when the loop terminates. We can name this label as $S$.next as an inherited symbolic attribute. The three-address intermediate code below is a direct application of the SDT schemes described in class assuming $L3 = S$.next.

```c
i = 0
t1 = 10
L1: if a < b goto L2
goto L3
L2: if i < t1 goto L4
goto L3
L4: t2 = c[i]
t3 = 0
if t2 < t3 goto L3
t4 = i + 1
i = t4
goto L1
L3:
```

The key observation in making this code better in terms of the number of executed instructions is that the portion of the while loop predicate $(a < b)$ is invariant with respect to the loop’s execution. Its value does not change throughout the execution and thus needs to be evaluated only once. The code can be rewritten as shown below.

```c
i = 0
t1 = 10
if a >= b goto L3
L2: if i < t1 goto L4
goto L3
L4: t2 = c[i]
t3 = 0
if t2 < t3 goto L3
t4 = i + 1
i = t4
goto L2
L3:
```