CSCI 565 - Compiler Design
Spring 2016
Midterm Exam

March 11, 2016 at 1:00 PM in class (KAP 147)
Duration: 2h 30 min.

Please label all pages you turn in with your name and student number.

Name: ________________________________  Number: ________

Grade:
Problem 1 [30 points]:
Problem 2 [20 points]:
Problem 3 [20 points]:
Problem 4 [30 points]:

Total:

INSTRUCTIONS:

1. This is a open-book exam and you may bring notes either typed or handwritten for your own personal use during the exam.

2. Please identify all the pages where you have answers that you wish to be graded. Also make sure to label each of the additional pages with the problem you are answering.

3. Use black or blue ink. No pencil answers allowed.

4. Staple or bind additional answer sheets together with this document to avoid being misplaced or worse, lost. Make sure this cover page is stapled at the front.

5. Please avoid laconic answers so that we can understand you understood the concepts being asked.
### Problem 1: Context-Free-Grammars and Parsing Algorithms [30 points]

Consider the context-free grammar (CFG) below for regular expression over the binary digit alphabet $\Sigma = \{0, 1\}$ with the concatenation operator $\cdot$, alternation operator $|$ and Kleene closure operator $\ast$. The start symbol is $RegExp$ and the set of terminal and non-terminal symbols are respectively $NT = \{ RegExp \}$ and $T = \{ (, )', .', \ast, '0', '1' \}$ and the five productions are as shown below.

1. $RegExp \rightarrow (' RegExp ')'$
2. $RegExp \rightarrow RegExp \ast$
3. $RegExp \rightarrow RegExp . RegExp$
4. $RegExp \rightarrow '0' \mid '1'$
5. $RegExp \rightarrow '0'$
6. $RegExp \rightarrow '1'$

**Questions:**

a) [05 points] Is this grammar suitable to be parsed using a predictive LL(0) parsing algorithm? Why?

b) [15 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] Typically the LR(0) table construction algorithm leads to tables that are very 'dense'. Construct the SLR parsing table by indicating the FOLLOW sets used to trim the entries on the table. Will this solve the conflicts? If not, comment on the nature of the conflict and how you think you could avoid 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 $(0) \rightarrow RegExp \; S$ and compute the set of LR(0) items as depicted below for a total of 12 states.

Based on this DFA of states as sets of LR(0) items, we can construct the LR(0) parsing table below.
c) In this case one can attempt to resolve the conflicts in the LR(0) parsing table using the FOLLOW set of RegExp. Unfortunately, FOLLOW(RegExp) = { '*', '.', '|', ')' } and as such there will still be a ‘reduce’ operation for these columns.

The problem is that this grammar is inherently ambiguous. A way to solve the ambiguity that arises by having consecutive '.' and '|' operators is to include an additional precedence-enforcement non-terminal symbol and the corresponding productions (as we’ve seen in class for the arithmetic grammar). A ‘factor’ and a ‘term’ non-terminals would give higher precedence to the concatenation '.' operator over alternation '|' operator.
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 \ '. ' \ 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 two attributive grammars based on this CFG G to compute the decimal-number value of the attribute val of S. For example, the translation of the string ‘101.001’ would result in a value of S.val of 5.125. Note that you are not supposed to modify the grammar.

Questions:

a) [10 points] Design an L-attributed syntax-directed definition for this grammar. As a suggestion, use an inherited attribute L.side that reflects which side of the decimal point a bit is on and use that information to adjust the subsequent rules.

b) [10 points] Design an S-attributed syntax-directed definition for this grammar.

Solution:

a) In this solution we have 2 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>
</tbody>
</table>

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

\[
\begin{align*}
S & \rightarrow L_1 \ ' ' \ L_2 \\
& \quad | L_1.\text{side} = R; \ L_1.\text{depth} = 0; \\
& \quad | L_2.\text{side} = L; \ L_2.\text{depth} = 0; \\
& \quad | S.\text{val} = L_1.\text{val} + L_2.\text{val} * 2^{L_2.\text{depth}} \\
S & \rightarrow L \\
& \quad | L.\text{side} = R; \ L.\text{depth} = 0; \\
& \quad | S.\text{val} = L.\text{val} \\
L & \rightarrow B \\
& \quad | L.\text{val} = B.\text{val} * 2^{L.\text{depth}} \\
& \quad | L.\text{length} = 1; \\
L_0 & \rightarrow L_1 \ B \\
& \quad | \text{if (}L_0.\text{side} \ == L) \text{ then} \\
& \quad | L_1.\text{depth} = L_0.\text{depth} + 1; \\
& \quad | L_0.\text{val} = L_1.\text{val} + B.\text{val} * 2^{L_0.\text{depth}}
\end{align*}
\]
|| else
|| Lₐ.depth = L₀.depth - 1;
|| L₀.val = L₁.val + B.val * 2^(L₀.length)
|| L₁.side = L₀.side
|| L₀.length = L₁.length + 1;

B → ‘0’ || B.val = 0
B → ‘1’ || B.val = 1

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”.

b) 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>
</tbody>
</table>

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

S → L₁ ‘.’ L₂    || S.val = L₁.val + L₂.val * 2^(L₂.length)
S → L            || S.val = L₁.val
L → B            || L.val = B.val
|| L.length = 1;
L₀ → L₁ B        || L₀.length = L₁.length + 1;
|| L₀.length = 2 * L₁.val + B.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

\[
S \rightarrow L_1 \cdot L_2 \quad \| \quad S.val = L_1.val + L_2.val \times 2^{-L_2.plnz}
\]

\[
S \rightarrow L \quad \| \quad S.val = L_1.val
\]

\[
L \rightarrow B \quad \| \quad L.val = B.val
\]

\[
\quad \quad \| \quad \text{if (B.val == 1) then}
\]

\[
\quad \quad \quad \quad \| \quad L.plnz = 1
\]

\[
\quad \quad \quad \quad \| \quad \text{else}
\]

\[
\quad \quad \quad \quad \| \quad L.plnz = 0;
\]

\[
\quad \quad \quad \| \quad L.length = 1;
\]
\[
L_0 \rightarrow L_1 B \quad \begin{align*}
&\quad L_0.\text{length} = L_1.\text{length} + 1; \\
&\quad \text{if (B.val == 1) then} \\
&\quad \quad L_0.\text{plnz} = L_0.\text{length} \\
&\quad \quad \text{else} \\
&\quad \quad L_0.\text{plnz} = L_1.\text{plnz}; \\
&\quad L_0.\text{length} = 2 \times L_1.\text{val} + B.\text{val}; \\
\end{align*}
\]

\[
B \rightarrow '0' \quad \begin{align*}
&\quad B.\text{val} = 0 \\
\end{align*}
\]

\[
B \rightarrow '1' \quad \begin{align*}
&\quad B.\text{val} = 1 \\
\end{align*}
\]

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 \text{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.

```plaintext
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) {
        goto Lbreak;
    }
    if (b == 0){
        goto Lbreak
    } else {
        a = a + d;
    }
    goto Lloop;
}
Lbreak: x = a;
y = b;
```

Solution:
**Problem 4: Back-patching of Loop Constructs [30 points]**

In class we have covered SDT schemes to generate code using the back-patching technique for a `while` and other loop constructs. In this exercise you are going to focus on the control-flow constructs such as the `continue`, `break` and `return` statement in imperative language such as C. In the case of the `continue` statement, the control is returned to the following iteration of the current looping construct. The `break` statement returns control to the first statement following the looping construct (i.e., ends the current loop construct and possibly returns to the immediately enclosing looping construct in the case of a nesting scenario). Lastly, the `return` statement returns control to the epilogue of the enclosing procedure, in effect terminating the execution of the currently executing procedure. It does bypasses all nesting looping constructs.

In this exercises you are asked to develop a syntax-directed definition scheme using the grammar productions listed below where for simplicity we have omitted the productions for the `Exp` non-terminal symbols in this grammar and have omitted some details about assignment statements. Use exclusively synthesized attributed and back-patching to generate code that correctly handles `break`, `continue` and `return` statements in terms of control flow. Do not focus on the structured control-flow of conditional if statement as this was covered in class.

(1) $\text{Stat} \rightarrow \text{while} \ Exp \ '{\{ } \text{List} \ '{\} }$

(2) $\text{Stat} \rightarrow \text{if} \ Exp \ '{\{ } \text{List} \ ')' \text{ else } '{\{ } \text{List} \ ')'$

(3) $\text{Stat} \rightarrow \text{return} \ Exp$

(4) $\text{Stat} \rightarrow \text{continue}$

(5) $\text{Stat} \rightarrow \text{break}$

(6) $\text{Stat} \rightarrow \text{id} \ '{=} \ Exp$

(7) $\text{List} \rightarrow \text{Stat} \ '{;} \ \text{List}$

(8) $\text{List} \rightarrow \text{Stat} \ '{;}$

Do not forget to show the augmented production with the marker non-terminal symbols, $M$ and possibly $N$ along with the corresponding rules for the additional symbols and productions. Argue for the correctness of your solution for the sample code below. Note that you are not being asked to draw the entire parse tree for this example, but rather to explain how the attributes of the return statement are carried over the parse tree and use for the purpose of backpatching.

```c
while exp {
    a = exp;
    if exp {
        continue;
    } else {
        break;
    }
    if exp {
        return exp;
    }
    b = exp;
}
```

**Solution:**

This solution is very similar to the one described in class. The really only difference is the use of another synthesized attribute, the `returnlist` to be back-patched only at the top-most production in the procedure. Only at this stage, and knowing with the exit “landing pad” or the “epilogue” for the procedure is, can the compiler know where to jump to in case of a `return` statement. As such, this `returnlist` attribute is propagated up the parse tree possibly being augmented with other `return` statements.
(1) Stat → while M₁ Exp ‘{’ M₂ List ‘}’ { 
    backpatch(List.nextlist, M₁.quad);
    backpatch(Exp.truelist, M₂.quad);
    Stat.nextlist = merge(Exp.falseList, List.breaklist);
    Stat.breaklist = nil;
    Stat.returnlist = List.returnlist;
}

(2) Stat → if Exp ‘{’ M₁ List₁ N ‘}’ else ‘{’ M₂ List₂ ‘}’ { 
    backpatch(Exp.truelist, M₁.quad);
    backpatch(Exp.truelist, M₂.quad);
    Stat.nextlist = merge(List₁.nextlist, List₂.nextlist, N.nextlist);
    Stat.breaklist = merge(List₁.breaklist, List₂.breaklist);
    Stat.returnlist = merge(List₁.returnlist, List₂.returnlist);
}

(3) Stat → return Exp { 
    Stat.nextlist = nil;
    Stat.breaklist = nil;
    Stat.returnlist = newList(nextAddr()); emit('goto __');
}

(4) Stat → continue; { 
    S.nextlist = newList(nextAddr()); emit('goto __');
    S.breaklist = nil;
    S.returnlist = nil;
}

(5) Stat → break; { 
    S.nextlist = nil;
    S.breaklist = newList(nextAddr()); emit('goto __');
    S.returnlist = nil;
}

(6) Stat → id ‘=’ Exp { 
    Stat.nextlist = null;
    Stat.breaklist = null;
    Stat.returnlist = null;
}

(7) List₁ → Stat ; M₂ List₂ { 
    backpatch(Stat.nextlist, M₂.quad);
    List₁.nextlist = List₂.nextlist;
    List₁.breaklist = merge(Stat.breaklist, List₂.breaklist);
    List₁.returnlist = merge(Stat.returnlist, List₂.returnlist);
}

(8) List → Stat { 
    List.nextlist = Stat.nextlist;
    List.breaklist = Stat.breaklist;
    List.returnlist = Stat.returnlist
}

(9) M₁ → ε { 
    M₁.quad = nextAddr;
}

(10) M₂ M₂ → ε { 
    M₂.quad = nextAddr;
}
The returnlist attribute is always propagated up the parse tree being merged along the way with other return statements’ locations of “to-be-filled”goto instructions. As such, and when reaching the top node of a procedures code, the corresponding rule can then backpatch all these locations with the appropriate value. This works well as there is a single exit point – the epilogue of a code (in principle) where all the return instructions should jump to before the epilogue code executed all the instructions regarding cleaning up the called procedure execution environment.