Problem 1: Attributive Grammar and Syntax-Directed Translation [30 points]

In class we described a SDT translation scheme for declarations in a PASCAL-like language where the identifiers precede the type declaration itself. In this exercise you are asked to develop an attributive grammar and syntax-directed definition that performs type checking for integer and real values.

This type checking means that your syntax-directed definition should first identify the value of expressions based on the declared types and then should check (hence the name type-checking) that on binary operations such as addition or multiplications (to make matters simple here) both operands are of the same type. To accomplish this you have a partial grammar below for which you need to:

a) [10 points] Define the attributes for each non-terminal symbol and the corresponding semantics rules.

b) [10 points] Determine the order in which the attributes need to be evaluated.

c) [10 points] Show an example of the attribute values in the code example: “s = a * b + 1.0” where “s” and “a” are declared as integer and “b” is declared as real. Include in the tree the declarations for these two variables.

Note that in the case of a type error, or mismatch the semantic action must be able to propagate an undefined or erroneous value to other expressions or identifiers to be later used by the compiler error reporting functions.

Solution:

a) As we saw in class the grammar segment shown above is already in a form that allows for synthesized attributes to be defined and propagated up a parse tree in a single pass. As such we define for all of the non-terminal symbols with the exception of Assign, Stmt and StmtList a synthesized attribute type. This attribute assumes the values, undetermined, integer, and real.

The implementation obviously makes use of a table, often referred to as a symbol table, where the information of the various variable declarations is stored and retrieved.

Given this introduction the grammar and corresponding semantic rules are described below:
Decl → id List
{ saveTypeInTable(id.string, List.type); }
List0 → ‘,’ id List1
{ saveTypeInTable(id.string, List1.type); List0.type = List1.type; }
List → ‘.’ Type
{ List.type = Type.type; }
Type → integer
{ Type.type = integer; }
Type → real
{ Type.type = real; }
Assign → id = Expr
{ if(lookUpTypeInTable(id.string) != Expr.type) then report compile time error; }
Expr0 → Expr1 + Expr2
{ if(Expr1.type != Expr2.type) then Expr0.type = undetermined;
else Expr0.type = Expr1.type; }
Expr0 → Expr1 * Expr2
{ if(Expr1.type != Expr2.type) then Expr0.type = undetermined;
else Expr0.type = Expr1.type; }
Expr → id
{ Expr.type = lookUpTypeInTable(id.string); }
Expr → Const
{ Expr.type = Const.type; }
Const → intconst
{ Const.type = integer; }
Const → realconst
{ Const.type = real; }

b) Given that the attributes are all synthesized they can be evaluated in a single bottom-up, left-to-right pass over the parse tree. This order is compatible with the LR parsing method studied in class and hence can be implemented as part of the parsing execution step.

c) For the sequence of input tokens derived from the sequence of characters “s = a * b + 1.0” we would obtain the parse tree depicted below for which we have annotated the corresponding nodes type attribute values and order of evaluation.
Problem 2: Static-Single Assignment Representation [10 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.

```plaintext
x = a * y + x;
if(x < 10) then
    y = 0;
else
    y = 1;
x = y * x;
```

Solution:

This particular example is fairly simple as there are no loops. As such an SSA representation for the code above is as shown below:

```plaintext
x_1 = a_0 * y_0 + x_0;
if(x_1 < 10) then
    y_1 = 0;
else
    y_2 = 1;
y_3 = \phi(y_1, y_2)
x_2 = y_3 * x_1;
```

As can be observed by inspection each use has a single definition point that reaches it and each value is defined only once. Notice also that the use of $y_0$ is the last use of that variables value before the if statement. As noted this representation makes very explicitly the last use and thus its use in optimizations such as register allocation.
Problem 3: Intermediate Code Generation [30 points]

For the assignment instruction below perform the following:

\[ x = (a + (b \times 2)) + 1 \]

a) [10 points] Augment the SDT scheme with a rule corresponding to the production \( E \rightarrow \text{const} \) and using a “value” attribute for the constant with its numeric value.

b) [05 points] Generate three-address instructions using the SDT scheme described in class (skipping over the rules for parenthesis to simplify) and without any minimization of temporaries.

c) [10 points] Redo the code generation but reusing temporaries with the method described in class.

d) [05 points] Argue that the solution found in c) is optimal.

Solution:

a) The rule for the additional production is very similar to the rule for the identifier case seen in class. In this case, however, you need to generate some code to save the constant value in a temporary and set the place attribute for the expression \( E \) to have the name of that temporary. The rule is thus as follows:

\[
E \rightarrow \text{const} \quad || \quad \text{tmp = newtemp(); E.place = tmp; E.code = \{ 'tmp = const.value' \}}
\]

b) Below is the annotated (simplified) parse tree where we show the code that is added to the output for each node resulting the in the final three-address code sequence as follows:

\[
t1 = 1; \\
t2 = 2; \\
t3 = b \times t2; \\
t4 = a + t3; \\
t5 = t4 + t1; \\
x = t5;
\]

c) Using the \text{newtemp} allocation function described in class that attempt to reuse a temporary as soon as its value is used we would get the code sequence shown below.

\[
t1 = 1; \\
t2 = 2; \\
t2 = b \times t2; \quad // \text{t2 is used and thus immediately reused on LHS} \\
t2 = a + t2; \quad // \text{idem} \\
t2 = t2 + t1; \quad // \text{idem} \\
x = t2;
\]

d) The solution above uses 2 temporaries while being good is not really optimal. We could replace many instances of \( t1 \) and \( t2 \) by the corresponding constant and thus eliminate several instructions and also reduce the number of temporary to a single temporary register as shown below.

\[
t1 = 1; \quad // \text{eliminated} - \text{forward propagated} \\
t2 = 2; \quad // \text{eliminated} - \text{forward propagated} \\
t1 = b \times 2; \\
t1 = a + t1; \\
t1 = t1 + 1; \\
x = t1;
\]
Problem 4: Back-patching of Loop Constructs [30 points]

In class we saw the actions for a Syntax-Directed Translation scheme to generated code using the back-patching technique for a while loop construct. In this exercise you will develop a similar scheme for the do-while construct using the production below and also taking into account continue and break statements. Argue that your solution works for the case of nested loops and break and continue statements at different nesting levels.

(1) \( S \rightarrow \text{do} \ M \text{ while } E; \)
(2) \( S \rightarrow \text{continue}; \)
(3) \( S \rightarrow \text{break}; \)
(4) \( L \rightarrow S \ ; L \)
(4) \( L \rightarrow S \)

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 without necessarily having to show an example.

Solution: We seen in class a possible approach to this SDT scheme is to have additional synthesized attributes for the statements, respectively a \textit{nextlist} and a \textit{breaklist}. In the \textit{nextlist} are the addresses of unresolved \texttt{goto} instructions that correspond to \texttt{continue} statements whereas in the \textit{breaklist} are the addresses of unresolved \texttt{goto} instructions that correspond to \texttt{break} statements. While the \textit{nextlist} need to be patched with the addresses of the first instruction of the current nesting level, \textit{i.e.} the first instructions that evaluates the control predicate of the loop, the \textit{breaklist} needs to be patched with the first address following the current S construct (that corresponds to a while loop). This cannot be immediately recognized at this level in the back-patching and thus the address of the \texttt{goto} in the \textit{breaklist} is passed “up” as part of the synthesized attribute \textit{nextlist} of S.

\[
\begin{align*}
(1) & \quad S \rightarrow \text{do } M_1 \text{ while } M_2 \ E; \quad \{ \text{backpatch}(L._{\text{nextlist}}, M_2._{\text{quad}}); \text{backpatch}(E._{\text{truelist}}, M_1._{\text{quad}}); \}
& \quad \text{S.nextlist = merge}(E._{\text{falseclist}}, L._{\text{breaklist}}); \text{S.breaklist = nil; } \\
(2) & \quad S \rightarrow \text{continue}; \quad \{ \text{S.nextlist = nextAddr(); emit('goto __'); S.breaklist = nil;} \\
(3) & \quad S \rightarrow \text{break}; \quad \{ \text{S.breaklist = nextAddr(); emit('goto __'); S.nextlist = nil; } \\
(4) & \quad L_1 \rightarrow S \ ; M_2 L_2 \quad \{ \text{backpatch}(S._{\text{nextlist}}, M_2._{\text{quad}}); \}
& \quad L_1._{\text{nextlist}} = L_2._{\text{nextlist}}; \}
& \quad L_1._{\text{breaklist}} = \text{merge}(S._{\text{breaklist}}, L_2._{\text{breaklist}}); \}
(5) & \quad L \rightarrow S \quad \{ \text{L.breaklist = S._{\text{breaklist}}; L._{\text{nextlist}} = S._{\text{nextlist}}; } \\
(6) & \quad M_1 \rightarrow \epsilon \quad \{ \text{M}_1._{\text{quad}} = \text{nextAddr;} \} \\
(7) & \quad M_2 \rightarrow \epsilon \quad \{ \text{M}_2._{\text{quad}} = \text{nextAddr;} \}
\end{align*}
\]

Regarding the first production, the first back-patching command fills in the places where the control in L is transferred to the next iteration, that is, to the evaluation of the conditional E that is given by the \( M_2._{\text{quad}} \) value. The second back-patching command links the places where the evaluation of E is false to \( M_1._{\text{quad}} \) that is to the top of the loop. Next we merge the places where are \texttt{goto} instructions with the \( E._{\text{falseclist}} \) as both these have addresses where the \texttt{gotos} will transfer control to the first instruction following the loop. The \texttt{continue} generates a single entry in a \textit{nextlist} whereas the \texttt{break} generates a single entry in a \textit{breaklist} of the corresponding S symbol. Regarding the sequencing of statement in production (4) we have to link the addresses from \texttt{continue} instructions in S with the first instruction in L_2. The addresses that correspond to break instructions in either and S or L_2 need to be merged whereas the L_1._{\text{nextlist}} is simply the locations that need to be filled in with the addresses after L_2 which is only known at the next level up. Note that nested loop will have the break instruction just to the nest level up (see the role of \( L._{\text{breaklist}} \) in (5) and \( S._{\text{nextlist}} \) in (1) and then the S._{\text{nextlist}} in (4) where it is patched to \( M_2._{\text{quad}} \).