Problem 1: Attributive Grammar and Syntax-Directed Translation [40 points]

In this problem you need to develop a grammar for regular expressions over the alphabet \{0,1\} and develop a L-attributed Syntax-Directed Definition for the translation of regular expressions to Non-Deterministic Finite Automata (NFA) using the notions of the Thompson’s construction described in class. Assume a generic “terminal” symbol production of the form \texttt{expr} \rightarrow \texttt{char} where \texttt{char} is a lexeme with a \texttt{val} attribute with either 0 or 1 value. Because your translation needs to create a graph, assume you have a set of auxiliary functions that allow you to create, reference the states of the NFA as well as add edges between them. In particular, assume you have a function \texttt{newState} that returns a reference to a newly created (and uniquely identified) state as well as a function \texttt{lookupState} that returns a reference to a state in the NFA. Also, use the function \texttt{addEdge} with two reference parameters and transition character that indicate a transition between states. You can also assign an attribute of ‘start’ or ‘final’ to a given state.

To help you structure your work, indicate the following:

a. [05 points] Describe the grammar for regular expressions
b. [10 points] Describe the attributes both synthesized and inherited used for your translation.
c. [20 points] Describe the translation using the semantic rules you have developed
d. [05 points] Show the translation for the Regular expression \texttt{RE=1.0*}

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). Comment on the need to save all the values at the end of the loop and how the SSA representation helps you in your evaluation of the code. Do not forget to include the \phi-functions.

\begin{verbatim}
b = 0;
d = 1;
a = ...;
i = ...;
L1: if(i > 0) {
  d = 0;
  b = b + 1;
  d = 1;
  i = i - 1;
  if(i < 0)
    a = ...;
    d = 0;
    goto Lbreak;
    goto L1;
}
Lbreak: x = a;
y = b;
\end{verbatim}
Problem 3: Symbol Table Organization [10 points]

For the PASCAL code below answer the following questions:

```pascal
01:  procedure main
02:      integer a, b, c;
03:  procedure f1(w,x);
04:      integer w, x;
05:  end;
06:  procedure f2(y,z);
07:      integer y, z;
08:  end;
09:  function f3(m,n):integer;
10:     integer m, n, x, y;
11:     x = a * (m+1);
12:     y = b * (n+1);
13:     f3 := x + y;
14:  end;
15:  function f4(k): integer;
16:      integer k;
17:      f4 := (k + 1);
18:  end;
19:  b = f3(c,z) + f4(c);
20:  end;
21: ...
22:  f1(a,b);
23:  end
```

a) [05 points] Draw the symbol tables for each of the procedures in this code (including main) and show their nesting relationship by linking them via a pointer reference in the structure (or record) used to implement them in memory. Include the entries or fields for the local variables, arguments and any other information you find relevant for the purposes of code generation, such as its type and location at run-time.

b) [05 points] For the statement in line 19 what are the specific instance of the variables used in this statement the compiler needs to locate? Explain how the compiler obtains the data corresponding to each of these variables table at compile time.

Problem 4: Intermediate Code Generation [10 points]

In C++ and other object-oriented programming languages, the access to data members, or fields of an object, is accomplished by the ‘.’ and ‘->’ operators. The choice of the use of these two operators depends on the l-value of the object expression. If it denotes a reference to an object the ‘->’ should be used, if a name of an object allocated automatically or statically, the ‘.’ operator must be used.

In this context, consider the code generation for expressions that access the data members or fields of objects. To do this you need to have access to the offset of the object where the field is located. This information should be stored in the symbol table that saves the class declaration and computes the size of each data field. Your code must also determine the address (base address) of the object that should be captured in the lhs of an expression.

As an example, the code below:

```cpp
a = b.f;
```

should lead to the generation of the following intermediate code where we have symbolically used the function offset(f) as the function that looks up the integer offset of the f field for the class b:

```cpp
t1 = &b;
t1 = t1 + offset(class(b),f);
t1 = *t1;
a  = t1;
```

In this context answer the following:

a) [05 points] Derived a SDT code generation scheme that handles expressions such as the ones described above as well as the ones where the l-value of an expression is a reference (and thus the ‘->’ operator must be used).

b) [05 points] Show your code generation scheme for the simple expression “c = a.f1 + b->f2” assuming the the fields f1 and f2 do exist in the corresponding class declarations.
Problem 5: Fault-Tolerance Constructs and Backpatching [30 points]

We have covered in class an SDT scheme to generated code using the back-patching technique for a while loop construct. In this exercise you will develop a similar scheme for a construct akin to the try-throw present in Java. Specifically, the construct has the syntax illustrated by the example below where in the presence of an error in the execution of the body, the computation is repeated up to a number $N$ of times. It is of course the responsibility of the programmers that the repetition of the computation does not have un-intended side-effects.

```
try $exp_1$
  { ...
    code ...
  }
throw $exp_2$;
```

and where the expressions $exp_1$ and $exp_2$ evaluate to integer values. A subtle point is that the value of the number of retries ($exp_1$) needs to be evaluated on entry of the construct and so changes to any of its operands in the code section of the construct have no effect on possible maximum number of times the code is repeated.

As the execution of the construct needs to examine for errors at run time, the compiler will check at the end of the construct if a special environment variable, named hw_error is non-zero. The variable also needs to be explicitly cleared upon entry of the construct. In case the variable assumes the zero value at the end of the first execution of the body, there is no re-execution and the computation proceeds.

A possible excerpt of a grammar that captures this construct is depicted below where the constructs for the expr non-terminal symbols are analogous to the ones used to define arithmetic expressions as described in class.

```
(1) stat → try $exp_1$ statlist throw $exp_2$
(2) statlist → stat ';' statlist
(3) statlist → stat
(4) stat → ...
```

In this exercise you are asked to develop a SDT scheme possibly using the backpatching technique to generate three-address code that implements the semantics of this construct. Should it be needed 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.