The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, SPRING SEMESTER 2009–2010

COMPILERS

ANSWERS

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

**Answer THREE questions**

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject-specific translation directories are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

Note: ANSWERS
Question 1

(a) The following is a valid MiniTriangle program (the line numbers are for reference only). Introduce five compile-time errors into the code by making small changes (like adding, changing, or deleting a lexical symbol or two). There should be two different syntactic errors, two different type errors, and one other contextual error. For each error, explain why it is an error and what kind of an error it is. You can consider each error in isolation: it is not necessary to consider any interaction between the introduced errors. The syntax of MiniTriangle is given in Appendix A.

```plaintext
1  let
2    var k : Integer
3  in
4    begin
5      k := 10;
6      while k > 0 do
7        let
8          const k2 : Integer = k * k
9        in
10       begin
11          putint(k2);
12          k := k - 1
13       end
14    end
```

**Answer:** Many possibilities. For example:

- **Change var to varr** at line 2. This is a syntax error as the keyword `let` has to be followed by a declaration, and a declaration must start with either the keyword `var` or the keyword `const` according to the MiniTriangle concrete syntax.

- **Leave out = k * k** from line 8. This is a syntax error as a constant definition must include a defining expression according to the MiniTriangle concrete syntax.

- **Declare k to be of type Char.** This will result in a type error e.g. at line 5 where an integer literal is being assigned to k.

- **Leave out > 0 from line 6.** This means the condition of the while loop will no longer be an expression of type `Bool` which will cause a type error to be reported.

- **Replace k2 by e.g. z at line 11.** This will result in a contextual error as no variable named z has been defined.
(b) Draw the abstract syntax tree for the following MiniTriangle fragment:

```
let var p : Bool := k > 42
in
  if p then
    putint(((k+1)))
  else
    k := 0
```

The relevant grammar is given in Appendix A. Start from the production for `Command`. (5)

**Answer:** Abstract syntax tree. Variable spelling terminals have been annotated with their spellings. These annotations are typeset in typewriter font.
(c) Construct a BNF grammar for expressions and numeric literals according to the following:

- The only form of basic expressions are numeric literals and the literal `null` (the empty list).
- The operators are as follows, in decreasing order of precedence, along with their arity (binary or unary) and associativity:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arity</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>unary</td>
<td>N/A</td>
</tr>
<tr>
<td>+</td>
<td>binary</td>
<td>left</td>
</tr>
<tr>
<td>&lt;, =, &gt;</td>
<td>binary</td>
<td>non</td>
</tr>
<tr>
<td>:</td>
<td>binary</td>
<td>right</td>
</tr>
</tbody>
</table>

Unary operators do not have any associativity, hence N/A (not applicable). Note that the three operators `<, =, and >` have the same precedence. Non-associative means that e.g. `3 < 5 < 7` should not be allowed: explicit parentheses would have to be used to make the meaning clear.

- Parentheses are used for grouping in the standard way.
- The numeric literals are non-empty sequences of decimal digits (0–9), possibly including a single decimal point. A numeric literal may not start or end with a decimal point.
- The entire grammar should be unambiguous and constructed in such a way that the parse tree of an expression reflects operator precedence and associativity.

**Answer:** For example:

\[
\begin{align*}
Exp1 & \rightarrow \ Exp2 : Exp1 \mid Exp2 \\
Exp2 & \rightarrow \ Exp3 < Exp3 \mid Exp3 = Exp3 \mid Exp3 > Exp3 \mid Exp3 \\
Exp3 & \rightarrow \ Exp3 + Exp4 \mid Exp4 \\
Exp4 & \rightarrow \ - \ Prim \mid Prim \\
Prim & \rightarrow \ LitNum \mid null \mid ( \ Exp1 ) \\
LitNum & \rightarrow \ Digits \mid Digits . Digits \\
Digits & \rightarrow \ Digit Digits \mid Digit \\
Digit & \rightarrow \ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]
Question 2

(a) Consider the following HTML-like document markup language. The terminals of the language are \texttt{word} and the following element tags:

<table>
<thead>
<tr>
<th>Element</th>
<th>Start Tag</th>
<th>End Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 Heading</td>
<td>&lt;h1&gt;</td>
<td>&lt;/h1&gt;</td>
</tr>
<tr>
<td>Level 2 Heading</td>
<td>&lt;h2&gt;</td>
<td>&lt;/h2&gt;</td>
</tr>
<tr>
<td>Paragraph</td>
<td>&lt;p&gt;</td>
<td>&lt;/p&gt;</td>
</tr>
<tr>
<td>Bold</td>
<td>&lt;b&gt;</td>
<td>&lt;/b&gt;</td>
</tr>
</tbody>
</table>

The following BNF grammar specifies the syntax of valid documents. $D$ is the start symbol:

\[
\begin{align*}
D & \rightarrow H1 \ D1 \\
D1 & \rightarrow H1 \ D1 \mid H2 \ D1 \mid P \ D1 \mid \epsilon \\
H1 & \rightarrow <h1> \ Ws \ </h1> \\
H2 & \rightarrow <h2> \ Ws \ </h2> \\
P & \rightarrow <p> \ Ws \ </p> \\
Ws & \rightarrow W \ Ws \mid W \\
W & \rightarrow \texttt{word} \mid <b> \ Ws \ </b>
\end{align*}
\]

Construct recursive-descent parsing functions for the non-terminals $D$, $D1$, $H1$, and $W$. We are only interested in checking whether the sequence of input tokens conforms to the grammar, not in building any structured representation.

Work in a functional style in Haskell, where a parsing function takes a list of tokens (along with any other arguments needed) and, in the case of a successful parse, using an option type like Haskell’s \texttt{Maybe}, returns the list of remaining tokens, otherwise returns an indication of failure, like \texttt{Nothing}. The parsing functions for $H2$, $P$, and $Ws$ are assumed to be given with signatures (in Haskell syntax):

\[
\begin{align*}
parsesH2 :: [\text{Token}] & \rightarrow \text{Maybe} [\text{Token}] \\
parsesP :: [\text{Token}] & \rightarrow \text{Maybe} [\text{Token}] \\
parsesWs :: [\text{Token}] & \rightarrow \text{Maybe} [\text{Token}]
\end{align*}
\]

The token type is defined as follows (in Haskell syntax):

\[
\text{data Token} = \text{H1S | H1E} \quad -- \text{Heading 1 start and end} \\
| \text{H2S | H2E} \quad -- \text{Heading 2 start and end} \\
| \text{PS | PE} \quad -- \text{Paragraph start and end} \\
| \text{BS | BE} \quad -- \text{Bold start and end} \\
| \text{Word [Char]} \quad -- \text{Word}
\]

(10)
**Answer:** The grammar is not left-recursive. Thus no need to first transform the grammar to eliminate left recursion. The grammar is also trivially suitable for predictive parsing. Thus the grammar can be transliterated straightforwardly into e.g. Haskell as follows (all parsing functions are given for the sake of completeness):

```
parseD :: [Token] -> Maybe [Token]
parsed ts =
  case parseH1 ts of
    Nothing -> Nothing
    Just ts1 -> parsed1 ts1

parsed1 :: [Token] -> Maybe [Token]
parsed1 ts =
  case ts of
    H1S : _ -> case parseH1 ts of
      Just ts1 -> parsed1 ts1
      _ -> Nothing
    H2S : _ -> case parseH2 ts of
      Just ts1 -> parsed1 ts1
      _ -> Nothing
    PS : _ -> case parseP ts of
      Just ts1 -> parsed1 ts1
      _ -> Nothing
      _ -> Just ts          -- Success on epsilon

parseH1 :: [Token] -> Maybe [Token]
parseH1 ts =
  case ts of
    H1S : ts1 -> case parseWs ts1 of
      Just (H1E : ts2) -> Just ts2
      _ -> Nothing
    _          -> Nothing

parseH2 :: [Token] -> Maybe [Token]
parseH2 ts =
  case ts of
    H2S : ts1 -> case parseWs ts1 of
      Just (H2E : ts2) -> Just ts2
      _ -> Nothing
    _          -> Nothing

parseP :: [Token] -> Maybe [Token]
```
parseP ts =
  case ts of
    PS : ts1 -> case parseWs ts1 of
      Just (PE : ts2) -> Just ts2
      _ -> Nothing
    _ -> Nothing

parseWs :: [Token] -> Maybe [Token]
parseWs ts =
  case parseW ts of
    Just ts1 -> case parseWs ts1 of
      Just ts2 -> Just ts2
      _ -> Just ts1
    _ -> Nothing

parseW :: [Token] -> Maybe [Token]
paseW ts =
  case ts of
    Word _ : ts1 -> Just ts1
    BS : ts1 -> case parseWs ts1 of
      Just (BE : ts2) -> Just ts2
      _ -> Nothing
    _ -> Nothing
Consider the following context-free grammar (CFG):

\[
S \rightarrow aAb \mid cS \mid B \\
A \rightarrow eAB \mid Af \mid g \\
B \rightarrow d
\]

\( S, A \) and \( B \) are nonterminal symbols, \( S \) is the start symbol, and \( a, b, c, d, e, f, \) and \( g \) are terminal symbols.

Explain how a bottom-up (LR) parser would parse the string \( ccaeegfdddb \) according to this grammar by reducing it step by step to the start symbol. Also state what the handle is for each step.

**Answer:** A bottom-up parser traces out a right-most derivation in reverse. It would reduce the word \( ccaeegfdddb \) as follows:

\[
\begin{align*}
ccaeegfdddb & \Leftarrow \text{(reduce by } A \rightarrow g) \\
ccaeAfdddb & \Leftarrow \text{(reduce by } A \rightarrow Af) \\
ccaeAdddb & \Leftarrow \text{(reduce by } B \rightarrow d) \\
ccaeABddb & \Leftarrow \text{(reduce by } A \rightarrow eAB) \\
ccaeAdbb & \Leftarrow \text{(reduce by } B \rightarrow d) \\
ccaeABbb & \Leftarrow \text{(reduce by } A \rightarrow eAB) \\
ccuAb & \Leftarrow \text{(reduce by } S \rightarrow aAb) \\
ccS & \Leftarrow \text{(reduce by } S \rightarrow cS) \\
cS & \Leftarrow \text{(reduce by } S \rightarrow cS) \\
S & \Leftarrow \text{(reduce by } S \rightarrow cS)
\end{align*}
\]

The handle has been underlined in each step.
(c) The DFA below recognizes the viable prefixes for the above CFG.

(i) Give five distinct examples of viable prefixes, of which at least one should be at least six symbols long and at least one should contain at least two non-terminals.

(ii) If an LR(0) parser is in state I6, with cae on the parse stack and the remaining input egfdde, what action does the parser take and what is the resulting state, parse stack, and remaining input?

(iii) If an LR(0) parser is in state I8, with caAb on the parse stack and no remaining input, what action does the parser take and what is the resulting state, parse stack, and remaining input?

Answer:

- E.g. $\epsilon$, a, c, ccaee, caeAB
  (An LR(0) parser DFA recognises viable prefixes. So just use the given DFA to find prefixes satisfying the requirements.)
- Shift $e$, stack caee, state I6, and remaining input gfdde.
- Reduce by $S \rightarrow aAb$, stack cS, state I11, remaining input $\epsilon$. 

G52CMP-E1
Question 3

(a) Given the abstract syntax for MiniTriangle in Appendix A, suggest a suitable Haskell representation (Haskell datatype definitions) for MiniTriangle expressions, declarations, and type denoters; i.e., the non-terminals Expression, Declaration, TypeDenoter. Use the type Name to represent terminals of kind Name, and the type Integer to represent terminals of the kind IntegerLiteral. Explain your construction.

Answer: Each of the nonterminals Expression, Declaration and TypeDenoter is mapped to a Haskell datatype definition with one constructor for each form of the syntactic construct, i.e., one constructor for each production. These constructors are named by the labels given in the rightmost column. Each non-terminal and “variable spelling terminal” of the RHS of a production gets mapped to a field of the corresponding constructor of the type corresponding to the non-terminal or variable-spelling terminal. A sequence of non-terminals (the EBNF *-construct) is translated into a list of elements of the type of the non-terminal, and an optional non-terminal into a Maybe of the type in question. (Additionally, these fields may be named, but here we’ll go for the most basic option.)

If we apply this method to the given MiniTriangle abstract syntax grammar, we get:

```haskell
data Expression = ExpLitInt Integer
  | ExpVar Name
  | ExpApp Expression [Expression]

data Declaration = DeclConst Name TypeDenoter Expression
  | DeclVar Name TypeDenoter (Maybe Expression)

data TypeDenoter = TDBaseType Name
```

(b) The following is a fragment of a Happy parser specification for MiniTriangle, dealing with expressions and declarations. The semantic actions for constructing an abstract syntax tree as a result of successful parsing have been left out (indicated by a boxed number, like [3]). Complete the fragment by providing suitable semantic actions using the representation of MiniTriangle expressions and declarations you defined in (a) above. Assume that the semantic value of the non-terminals unary_operator and binary_operator is of type Name, representing the name of a unary or binary function corresponding to the operator in question. Assume further that the type of the semantic value of the terminal LITINT is Integer and the one of ID is Name.
expressions :: [Expression]
expressions
  : expression { 1 }
  | expression ',' expressions { 2 }

expression :: Expression
expression
  : primary_expression { 3 }
  | expression binary_operator primary_expression { 4 }

primary_expression :: Expression
primary_expression
  : LITINT { 5 }
  | ID { 6 }
  | unary_operator primary_expression { 7 }
  | '(' expression ')' { 8 }

declaration :: Declaration
declaration
  : CONST ID ':' type_denoter '=' expression { 9 }
  | VAR ID ':' type_denoter { 10 }
  | VAR ID ':' type_denoter ':=' expression { 11 }

Answer:

\[
\begin{align*}
1 &= [\$1] \\
2 &= \$1 : \$3 \\
3 &= \$1 \\
4 &= \text{ExpApp (ExpVar \$2) [\$1, \$3]} \\
5 &= \text{ExpLitInt \$1} \\
6 &= \text{ExpVar \$1} \\
7 &= \text{ExpApp (ExpVar \$1) [\$2]} \\
8 &= \$2 \\
9 &= \text{DeclConst \$2 \$4 \$6} \\
10 &= \text{DeclVar \$2 \$4 Nothing} \\
11 &= \text{DeclVar \$2 \$4 (Just \$6)}
\end{align*}
\]

Marking: 2 marks each for 4, 7, 10, 11, 1 mark each for the remaining seven fragments. (4 \times 2 + 7 = 15)
Question 4

(a) Formally state the progress and preservation theorems, explain what they mean in plain English, and why, when taken together, these theorems mean that “well-typed programs do not go wrong”. (7)

Answer:

- **THEOREM [PROGRESS]**: Suppose that \( t \) is a well-typed term (i.e., \( t : T \)), then either \( t \) is a value or else there is some

- **THEOREM [PRESERVATION]**: If \( t : T \) and \( t \; \rightarrow \; t' \) then \( t' : T \).

The progress theorem says that any well-typed term either is a value, i.e. the end result, or it will be possibly to evaluate the term one step further.

The preservation theorem says that well-typed terms evaluate to well-typed terms.

Thus, taken together, if we have a well typed term which isn’t a value, then it can be evaluated one step into another well-typed term. If this is still not a value, then it can be evaluated again, and so on, until we eventually, if the evaluation terminates, get a value.

Of course, the evaluation may fail to reach a normal form, but at least it will not get “stuck” in a normal form which is not a value. This is what it means for a program to not “go wrong”.

(b) Consider the following expression language:

\[
\begin{align*}
\text{expressions:} & \\
| & n & \text{natural numbers, } n \in \mathbb{N} \\
| & x & \text{variables, } x \in \text{Name} \\
| & (e, e) & \text{pair constructor} \\
| & \text{if } e \text{ then } e \text{ else } e & \text{conditional}
\end{align*}
\]

where Name is the set of variable names. The types are given by the following grammar:

\[
\begin{align*}
\text{types:} & \\
| & \text{Nat} & \text{natural numbers} \\
| & \text{Bool} & \text{Booleans} \\
| & t \times t & \text{pair (product) type}
\end{align*}
\]

The ternary relation \( \Gamma \vdash e : t \) says that expression \( e \) has type \( t \) in the typing context \( \Gamma \) and it is defined by the following typing rules:
\[
\Gamma \vdash n : \text{Nat} \quad \text{(T-NAT)}
\]
\[
\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \quad \text{(T-VAR)}
\]
\[
\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \quad \text{(T-PAIR)}
\]
\[
\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t} \quad \text{(T-COND)}
\]

A typing context, \( \Gamma \) in the rules above, is a comma-separated sequence of variable-name and type pairs, such as

\[x : \text{Nat}, \, y : \text{Bool}, \, z : \text{Nat}\]

or empty, denoted \( \emptyset \). Typing contexts are extended on the right, e.g. \( \Gamma, \, z : \text{Nat} \), the membership predicate is denoted by \( \in \), and lookup is from right to left, ensuring recent bindings hide earlier ones.

(i) Use the typing rules given above to formally prove that the expression 

\[\text{if } b \text{ then } 42 \text{ else } x\]

has type \( \text{Nat} \) in the typing context

\( \Gamma_1 = b : \text{Bool}, \, x : \text{Nat} \)

The proof should be given as a proof tree. (5)

**Answer:**

\[
\begin{array}{c}
b : \text{Bool} \in \Gamma_1 \\
\hline
\Gamma_1 \vdash b : \text{Bool} \quad \text{T-VAR} \quad \text{T-NAT} \quad \Gamma_1 \vdash x : \text{Nat} \quad \Gamma_1 \vdash \text{Nat} \in \Gamma_1 \quad \text{T-VAR}
\end{array}
\]

\[
\frac{\Gamma_1 \vdash b : \text{Bool} \quad \Gamma_1 \vdash \text{Nat} \in \Gamma_1 \quad \Gamma_1 \vdash x : \text{Nat}}{\Gamma_1 \vdash \text{if } b \text{ then } 42 \text{ else } x : \text{Nat}} \quad \text{T-COND}
\]

(ii) The expression language defined above is to be extended with function abstraction and application as follows:

\[
\begin{array}{c}
e \rightarrow \text{expressions:} \\
\vdots \quad \vdots \quad \vdots
\end{array}
\]

\[
\begin{array}{c}
| \quad \lambda x : t . e \quad \text{function abstraction} \\
| \quad e \; e \quad \text{function application}
\end{array}
\]

\[
\begin{array}{c}
t \rightarrow \text{types:} \\
\vdots \quad \vdots \quad \vdots
\end{array}
\]

\[
\begin{array}{c}
| \quad t \rightarrow t \quad \text{function (arrow) type}
\end{array}
\]

G52CMP-E1

*Turn Over*
For example, the function abstraction \( \lambda x \cdot \text{Nat}. x \) is the identity function on \( \text{Nat} \), the natural numbers, and \( (\lambda x \cdot \text{Nat}. x) \ 42 \) is the application of the identity function on natural numbers to 42.

Provide a typing rule for each of the new expression constructs, in the same style as the existing rules, reflecting the standard notions of function abstraction and function application. (6)

**Answer:**

\[
\begin{align*}
\Gamma, x : t_1 \vdash e : t_2 & \quad \text{(T-ABS)} \\
\Gamma \vdash \lambda x : t_1. e : t_1 \rightarrow t_2 \\
\Gamma \vdash e_1 : t_1 \rightarrow t_1 & \quad \Gamma \vdash e_2 : t_1 \\
\Gamma \vdash e_1 e_2 : t_1 & \quad \text{(T-APP)}
\end{align*}
\]

(iii) Use the extended set of typing rules to formally prove that the expression

\[
\text{fst } ((\lambda f : \text{Nat} \rightarrow \text{Nat}. (f \ 3, 11)) \ \text{dec})
\]

has type \( \text{Nat} \) in the typing context

\[
\Gamma_2 = \text{dec} : \text{Nat} \rightarrow \text{Nat}, \ \text{fst} : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}
\]

The proof should be given as a proof tree. (7)

**Answer:** Let \( g = \lambda f : \text{Nat} \rightarrow \text{Nat}. (f \ 3, 11) \). Let us first determine the type of the expression \( g \):

\[
\begin{align*}
\Gamma_3 \vdash f : \text{Nat} \rightarrow \text{Nat} & \quad \text{T-VAR} \\
\Gamma_3 \vdash 3 : \text{Nat} & \quad \text{T-VAR} \\
\Gamma_3 \vdash 3 : \text{Nat} & \quad \text{T-NAT} \\
\Gamma_3 = \Gamma_2, f : \text{Nat} \rightarrow \text{Nat} & \quad \text{T-APP} \\
\Gamma_3 \vdash (f \ 3, 11) : \text{Nat} \times \text{Nat} & \quad \text{T-PAIR} \\
\Gamma_2 \vdash g : (\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \times \text{Nat}) & \quad \text{T-ABS}
\end{align*}
\]

We then check the type of the application \( g \ \text{dec} \)

\[
\begin{align*}
\Gamma_2 \vdash g : (\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \times \text{Nat}) & \quad \text{T-VAR} \\
\Gamma_2 \vdash \text{dec} : \text{Nat} \rightarrow \text{Nat} & \quad \text{T-VAR} \\
\Gamma_2 \vdash g \ \text{dec} : \text{Nat} \times \text{Nat} & \quad \text{T-APP}
\end{align*}
\]

Finally, we turn to the complete expression:

\[
\begin{align*}
\Gamma_2 \vdash \text{fst} : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} & \quad \text{T-VAR} \\
\Gamma_2 \vdash \text{fst} : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} & \quad \text{T-VAR} \\
\Gamma_2 \vdash g \ \text{dec} : \text{Nat} \times \text{Nat} & \quad \text{T-APP} \\
\Gamma_2 \vdash \text{fst } (g \ \text{dec}) : \text{Nat}
\end{align*}
\]
Question 5

(a) Consider the following code skeleton (note: nested procedures):

```pascal
var a, b, c: Integer
proc P
    var x, y, z: Integer
    proc Q
        var u, v: Bool
        ...
        begin ... Q() ... end
    proc R
        var w: Bool
        ...
        begin ... Q() ... end
        begin ... R() ... end
    begin ... P() ... end
```

The variables `a`, `b`, and `c` are global, the variables `x`, `y`, and `z` are local to procedure `P`, and procedures `Q` and `R` are also local to `P`.

Assume a stack-based memory allocation scheme with dynamic and static links.

(i) Show the layout of the activation records on the stack after the main program has invoked the procedure `P`. Explain how global and local variables are accessed from `P`. (4)

(ii) Show the layout of the activation records on the stack once procedure `Q` has called itself recursively once after having been called from `R` which in turn was called from `P` which was called from the main program. Explain how global variables, `P`'s variables, and `Q`'s own local variables (and what instance of the latter) are accessed from `Q`. (6)

**Answer:**

(i) *Activation record layout (stack grows downwards):*

```
<table>
<thead>
<tr>
<th></th>
<th>SB</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>ST</td>
<td>LB</td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td>var</td>
<td></td>
<td></td>
<td>st, addr</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

G52CMP-E1

Turn Over
SB is Stack Base, LB is Local Base (or Frame Pointer), ST is Stack Top. The activation record of the currently active procedure/function is the one between LB and ST. The solid arrow from the current activation record represents the dynamic link, i.e. it refers to the activation record of the caller and is thus equal to the previous value of LB. The dashed arrow represents the static link. Global variables are accessed relative to SB. Variables local to P are accessed relative to LB.

(ii) Activation record layout (stack grows downwards):

SB is Stack Base, LB is Local Base (or Frame Pointer), ST is Stack Top. The solid arrows represent the dynamic link, the dashed ones is the static link. Global variables are accessed relative to SB. Variables local to Q are accessed relative to LB. Hence the second instance of these variables are being accessed (corresponding to the currently active procedure). P’s variables are accessed by following the static links until the activation record corresponding to the correct enclosing scope has been reached. In this case, only one step as P’s scope directly encloses Q’s scope. So P’s variables can be accessed relative to the static link in Q’s activation record.
(b) This question concerns register allocation.

(i) Explain what register allocation is and why it is useful. Illustrate by a small example (a code fragment before and after register allocation). (7)

(ii) Outline how the graph-colouring register allocation method works. You need to explain the notions of liveness and interference graph. Again, illustrate with an example or examples. (8)

Answer: See lecture notes.
Appendix A

This appendix contains the grammars for the MiniTriangle lexical, concrete, and abstract syntax. The following typographical conventions are used to distinguish between terminals and non-terminals:

- nonterminals are written like this
- terminals are written like this
- terminals with variable spelling and special symbols are written like this

MiniTriangle Lexical Syntax:

\[
\begin{align*}
\text{Program} & \rightarrow (\text{Token} \mid \text{Separator})^* \\
\text{Token} & \rightarrow \text{Keyword} \mid \text{Identifier} \mid \text{IntegerLiteral} \mid \text{Operator} \\
& \quad \mid ; \mid : \mid := \mid = \mid ( \mid ) \mid eol \\
\text{Keyword} & \rightarrow \text{begin} \mid \text{const} \mid \text{do} \mid \text{else} \mid \text{end} \mid \text{if} \mid \text{in} \\
& \quad \mid \text{let} \mid \text{then} \mid \text{var} \mid \text{while} \\
\text{Identifier} & \rightarrow \text{Letter} \mid \text{Identifier Letter} \mid \text{Identifier Digit} \\
& \quad \text{except} \ \text{Keyword} \\
\text{IntegerLiteral} & \rightarrow \text{Digit} \mid \text{IntegerLiteral Digit} \\
\text{Operator} & \rightarrow ^ \mid * \mid / \mid + \mid - \mid < \mid <= \mid == \mid != \mid >= \mid > \mid && \mid || \mid ! \\
\text{Letter} & \rightarrow A \mid B \mid \ldots Z \mid a \mid b \mid \ldots z \\
\text{Digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\text{Separator} & \rightarrow \text{Comment} \mid \text{space} \mid eol \\
\text{Comment} & \rightarrow // \ (\text{any character except } eol)^* \ eol
\end{align*}
\]
MiniTriangle Concrete Syntax:  Note: According to this version of the grammar, all binary operators are left associative and have the same precedence.

\[
\begin{align*}
\text{Program} & \rightarrow \text{Command} \\
\text{Commands} & \rightarrow \text{Command} \\
& | \text{Command} ; \text{Commands} \\
\text{Command} & \rightarrow \text{VarExpression} := \text{Expression} \\
& | \text{VarExpression} ( \text{Expressions} ) \\
& | \text{if} \ \text{Expression} \ \text{then} \ \text{Command} \ \text{else} \ \text{Command} \\
& | \text{while} \ \text{Expression} \ \text{do} \ \text{Command} \\
& | \text{let} \ \text{Declarations} \ \text{in} \ \text{Command} \\
& | \text{begin} \ \text{Commands} \ \text{end} \\
\text{Expressions} & \rightarrow \text{Expression} \\
& | \text{Expression}, \text{Expressions} \\
\text{Expression} & \rightarrow \text{PrimaryExpression} \\
& | \text{Expression} \ \text{BinaryOperator} \ \text{PrimaryExpression} \\
\text{PrimaryExpression} & \rightarrow \text{IntegerLiteral} \\
& | \text{VarExpression} \\
& | \text{UnaryOperator} \ \text{PrimaryExpression} \\
& | ( \ \text{Expression} \ ) \\
\text{VarExpression} & \rightarrow \text{Identifier} \\
\text{BinaryOperator} & \rightarrow ^{\ ^} | * | / | + | - | < | <= | == | != | >= | > | && | || \\
\text{UnaryOperator} & \rightarrow - | ! \\
\text{Declarations} & \rightarrow \text{Declaration} \\
& | \text{Declaration} ; \text{Declarations} \\
\text{Declaration} & \rightarrow \text{const} \ \text{Identifier} : \ \text{TypeDenoter} = \ \text{Expression} \\
& | \text{var} \ \text{Identifier} : \ \text{TypeDenoter} \\
& | \text{var} \ \underline{\text{Identifier}} : \ \text{TypeDenoter} := \ \text{Expression} \\
\text{TypeDenoter} & \rightarrow \text{Identifier}
\end{align*}
\]
MiniTriangle Abstract Syntax: Name includes both Identifier and Operator; i.e., Identifier ⊆ Name and Operator ⊆ Name

Program → Command

Command → Expression := Expression
           | Expression ( Expression* )
           | if Expression then Command
           | else Command
           | while Expression do Command
           | let Declaration* in Command
           | begin Command* end

Expression → IntegerLiteral
            | Name
            | Expression ( Expression* )

Declaration → const Name : TypeDenoter = Expression
             | var Name : TypeDenoter ( = Expression | ε )

TypeDenoter → Name

Program
CmdAssign
CmdCall
CmdIf
CmdWhile
CmdLet
CmdSeq
ExpLitInt
ExpVar
ExpApp
DeclConst
DeclVar
TDBaseType