1995 Paper 3 Question 4

Compiler Construction

Construct the characteristic finite state machine for the following grammar.

$$\begin{array}{lcl} S & \rightarrow & A\ B\ \texttt{eof} \\ A & \rightarrow & A\ B \ \mid\ B\ a \\ B & \rightarrow & (A)\ \mid\ b \end{array}$$

[6 marks]

Explain what is meant by the FOLLOW set for a non-terminal symbol in a grammar, and derive the FOLLOW sets for A and B in the above grammar.

[4 marks]

Construct, with explanation, the SLR(1) *action* and *goto* matrices for the above grammar. [5 marks]

Illustrate how the SLR(1) parsing algorithm works for this grammar by showing the successive states of the parser stack and input stream while parsing

$$b\ a\ b\ (\ b\ a\ )\ \texttt{eof}$$

[5 marks]

1995 Paper 4 Question 4

Compiler Construction

Outline how minimum cost code can be compiled from a parse tree using rules that consist of tree template replacements, costs and corresponding code. Illustrate your answer by considering the translation of the expression

```
Add(Add(K1, K2), Add(K3, K4))
```

using the following rules:

```
#1  Ri  <-  Add(Ri, Rj)             cost: 2   code: ADDR  Ri,Rj

#2  Ri  <-  Add(Ri, Kc)             cost: 3   code: ADDI  Ri,c

#3  Ri  <-  Kc                      cost: 2   code: LOADI Ri,c

#4  Ri  <-  Add(Ri, Add(Rj, Kc))    cost: 4   code: ADDRI Ri,Rj,c
```

In your answer you should derive the finite state machine needed for the efficient matching of these four rules, and you should also give the cost and resulting translation of the given example expression. [15 marks]

Briefly discuss in what ways this algorithm may fail to generate optimum code when used in a compiler. [5 marks]

1995 Paper 5 Question 6

Compiler Construction

Give a BNF syntax for unsigned floating point constants that would allow constants such as

```
          1.2          3E4          0.23E-8
```

but not

```
          .123         20.          21.E4
```

[6 marks]

Construct a finite state machine that could be used in a lexical analyser to read in such floating point numbers. [7 marks]

Show how this finite state machine can be extended to evaluate the floating point number it reads. [7 marks]

1995 Paper 6 Question 6

Compiler Construction

Compare two possible mechanisms that can be used to allow efficient execution time access to free variables of functions in a programming language that permits functions to be defined within other functions. [12 marks]

Discuss whether more efficient implementations are possible if the language permits no recursive function calls, while still allowing functions to be defined within other functions. [8 marks]

1996 Paper 3 Question 4

Compiler Construction

Describe what is meant by a *phrase structured grammar* and a *context free grammar*. [3 marks]

Describe an algorithm to calculate the set $L_T(P)$ of all terminal symbols that can start a string derived from the non-terminal P using one or more productions of a given context free grammar. Illustrate your answer by calculating L_T sets for the following grammar:

```
S -> U V

V -> + U V | ϵ

U -> X W

W -> * X W | ϵ

X -> ( S ) | n
```

[6 marks]

Describe an algorithm to calculate the set `FOLLOW(P)` of terminal and non-terminal symbols for a given context free grammar, where

$$\texttt{FOLLOW(P) = \{ X | S } \overset{*}{\Rightarrow} \texttt{ ... P X ... \}}$$

i.e. all symbols that can follow `P` in a sentential form derived from the sentence symbol `S`. Illustrate your answer by calculating the `FOLLOW` sets for the grammar given above. [6 marks]

Outline possible ways in which the space used by the Action and Goto matrices of an SLR(1) parser can be reduced. [5 marks]

1996 Paper 4 Question 4

Compiler Construction

Describe a structure that could be used to represent the abstract syntax tree of the following λ-expression:

$$(\lambda a.(\lambda f.fa)(\lambda n.n+1))3$$

Outline the definition of an evaluator function `eval(expr, env)` that could be used to evaluate a given expression `expr` represented in this way in the context of an environment given by `env`. Pay particular attention to the treatment of bound variables and the mechanism you use for function calls. [7 marks]

Is it possible with your implementation to give it a λ-expression which would cause `eval` to recurse to an unlimited depth? If so, give such a λ-expression; if not, explain why. [3 marks]

Is it possible with your implementation to give it a λ-expression that causes an environment chain of unlimited length to be created during the evaluation? If so, give a λ-expression that would cause this; if not, explain why it is not possible. [3 marks]

In an erroneous implementation of `eval` the value of

$$(\lambda x.\mathit{body})$$

is an object that does not incorporate the contextual environment, and whose call evaluates *body* in an environment derived from the environment of the call. Explain why such an implementation is wrong, giving an example λ-expression that would yield different results when evaluated by this implementation and yours. [7 marks]

1996 Paper 5 Question 6

Compiler Construction

Outline the key features of the design of the part of a compiler that will translate the abstract syntax tree representation of a program into a stack-based intermediate code. Concentrate on those features used in the translation of the following fragment:

```
...
LET i = k
LET j = k
WHILE (i>0) AND (j<100) DO { i := i-1; j := j+2 }
...
```

In particular, concentrate on the mechanism you would choose to deal with

(*a*) the scopes of identifiers [6 marks]

(*b*) the compilation of boolean expressions involving the operators `NOT`, `AND` and `OR` [6 marks]

(*c*) the translation of the `WHILE` command [4 marks]

(*d*) the translation of the two assignments [4 marks]

1996 Paper 6 Question 6

Compiler Construction

Programming languages are usually compiled into the machine code of the target computer, but sometimes an interpretive system is used. Discuss the relative merits of these two approaches. [7 marks]

Outline the key features of the design of an interpretive code that would be suitable for an implementation of the C programming language and describe the overall structure of an interpreter for it. [7 marks]

What techniques could be used to improve (*a*) the space efficiency, and (*b*) the time efficiency of interpretive codes? [6 marks]

1997 Paper 3 Question 4

Compiler Construction

Consider the following grammar for expressions (`<E>`) and commands (`<C>`).

```
<E> ::= i | n | <E> - <E> | <E> ** <E> | ( <E> )

<C> ::= i := <E>
      | if <E> then <C> | if <E> then <C> else <C>
      | <C> repeatwhile <E> | <C> ; <C> | { <C> }
```

Show that there are syntactic ambiguities between (*a*) the minus (`-`) and exponentiation (`**`) operators, (*b*) the `if`-command and the `if-then-else`-command, and (*c*) the `if-then-else`-command and the `repeatwhile`-command. [4 marks]

Define, in a programming language notation of your choice, a recursive descent parser that will construct the abstract syntax tree for an input stream conforming to the above syntax for commands. You may assume the existence of a function `lex()` that will yield an integer representing the next lexical token from the input stream, and the functions `mk2(op,x)`, `mk3(op,x,y)` and `mk4(op,x,y,z)` that will construct abstract syntax tree nodes with a given operator and one, two or three operands. You should assume that exponentiation is right associative and more binding than subtraction which is left associative. The command following `then` should be the longest possible and the command before `repeatwhile` should be the shortest possible. [12 marks]

Briefly outline how you would modify your parser if the command to the left of `repeatwhile` was changed to be the longest (rather than the shortest). [4 marks]

1997 Paper 4 Question 4

Compiler Construction

Write brief notes on the possible target code sequences needed for the implementation of

(*a*) dynamically-sized local vectors

(*b*) labels and `GOTO` statements in a block-structured programming language

(*c*) access to free variables of functions using the static chain method

(*d*) an exception mechanism such as found in, for instance, ML or Modula-3

[5 marks each]

1997 Paper 5 Question 6

Compiler Construction

Describe an efficient tree pattern-matching algorithm that could be used to find a cheapest covering of an abstract syntax tree by pattern templates with given costs. Illustrate your algorithm using the following templates:

```
#1 R  <- k                     cost: 1

#2 R  <- f(R, k)               cost: 2

#3 R  <- f(R, R)               cost: 2

#4 R  <- f(R, f(R, k))         cost: 3

#5 R  <- f(f(R, k), R)         cost: 4
```

and the following tree:

```
f(f(k,k),f(k,k))
```

[20 marks]

1997 Paper 6 Question 6

Compiler Construction

Investigate whether the following grammar for regular expressions is SLR(1) by attempting to construct its Action and Goto matrices.

```
S -> R eof

R -> F | R + F

F -> P | F P

P -> x | ( R ) | P *
```

Find all the conflicts, if any, in the two matrices. [20 marks]

1998 Paper 3 Question 1

Compiler Construction

Give *short* answers for each of the following.

(*a*) Give a program which has a different result when interpreted using dynamic scoping from that using static scoping.

(*b*) Give a program (using `let` to declare variables) which runs with no run-time type errors under dynamic typing but is invalid in a static typing regime such as that of Java or C.

(*c*) Compared with `int v[8]`, why might the declaration `int v[`*e*`]` cause less efficient code to be compiled if *e* is not a constant expression, or even if *e* is a large constant?

(*d*) Give two context-free grammars which accept the same strings with one being ambiguous and one non-ambiguous.

(*e*) Give a data-structure type declaration in a language of your choice which might be useful to represent parse trees for the following grammar:

```
E  ->  E + T | E - T | T
T  ->  P ^ T | P
P  ->  ( E ) | n
```

Here `E` is the start symbol and `n` is a terminal symbol representing the result of lexing an integer constant.

[4 marks each]

1998 Paper 4 Question 4

Compiler Construction

Sketch parsers based on

(*a*) recursive descent, and [8 marks]

(*b*) a table-driven method of your choice (e.g. SLR(1)) [12 marks]

suitable for parsing the following grammar:

```
S  ->  E eof
E  ->  E + T | E - T | T
T  ->  P ^ T | P
P  ->  ( E ) | n
```

with `S` as the start symbol. The table-driven parser should include the associated algorithm which interprets the table. The parsers do not need to produce a parse tree, merely to report whether the input string is generated by the above grammar. You may assume there is a routine `lex()` which when called places the next symbol (`+`, `-`, `^`, `(`, `)`, `n`, `eof`) in variable `token`.

1998 Paper 5 Question 6

Compiler Construction

Explain how a parse-tree representation of a program may be converted into a stack-based intermediate language giving sketches of code to translate expressions, assignments and the `if-then-else` command; you should also explain how occurrences of a variable in an expression or assignment are translated.

The program may be assumed to conform to the following syntax:

```
E  ->  n | x | E + E | f(E,E)
D  ->  let f(x,x) = {Dseq; Cseq; E} | let x = E
C  ->  x := E; | if E then C else C
Cseq -> C | C Cseq
Dseq -> D | D Dseq
```

with start symbol `Dseq`. Here `n` corresponds to integer constants, `x` corresponds to identifiers used as variable names and `f` corresponds to identifiers used as function names (you may assume these are disjoint). The function declaration construct has the effect of defining a function which, when called, makes declarations, performs commands and then returns the result of its expression; note that therefore functions may be defined within functions, but the above restriction on identifiers means that they cannot be returned as results. [20 marks]

1998 Paper 6 Question 5

Compiler Construction

You have been given a new programming language with a C-like syntax, with integer variables and functions and with static binding of free variables. Your manager can parameterise certain aspects of the language, including the following three options:

- For "`int x = e;`" whether the variable `x` has the same l-value of `e` or whether a new l-value is created and initialised to the r-value of `e`. If `e` is only an r-value then a new l-value is created in both circumstances.

- For "`int f(int x) { ... }`" whether the variable `x` is passed by l-value ("by reference") or by r-value ("by value"). If the switch is set to "l-value" and the value passed is only an r-value then a new l-value is created, initialised and passed.

- For "`int f(int x) { ... y ... }`" (where the variable `y` is free to `f`) whether the value of `y` is calculated at the times of its uses (association by l-value) or at the time of the definition of `f` (association by r-value).

As a test of your programming skills your manager asks you to write a program which tells how the language has been parameterised. Do so by printing a 3-digit decimal number where the "hundreds" digit is one or two according to whether the first option is by l-value or r-value respectively, similarly with the "tens" digit for the second and with the "units" digit for the third option.

[10 marks]

Explain the structure of an object module which an assembler or compiler might produce to be processed by a linker. Your answer should include discussion of the various object module features needed to represent the compiled form of the C program:

```
int a[10] = { 2,3,5,7,11,13,17,19,23,29 };
extern int b[10];
extern int g(int);
int f(int y)
{   return g(y) + b[5] + a[6];
}
```

[10 marks]

1999 Paper 3 Question 4

Compiler Construction

A programming language has expressions e with the following syntax:

$$\begin{aligned} e ::= x \;\; & \mid n \mid e + e' \mid e(e') \mid (e) \\ & \mid \texttt{let}\ x = e\ \texttt{in}\ e' \\ & \mid \texttt{letsta}\ f(x) = e\ \texttt{in}\ e' \\ & \mid \texttt{letdyn}\ f(x) = e\ \texttt{in}\ e' \end{aligned}$$

where f and x range over identifiers and n ranges over numbers. The three *let* variants introduce simple variables (`let`) and (non-recursive) functions whose variables are statically (`letsta`) or dynamically (`letdyn`) bound.

Using e itself (or any related language whose relationship to e is explained) as abstract syntax define an evaluator *eval* which, when given an expression e and an environment ρ, yields the value of evaluating e in ρ. The evaluator can be written in a language of your choice or in mathematical pseudo-code. [12 marks]

Explain carefully in one sentence each:

(a) the forms of value which *eval* may return;

(b) the form(s) of value which constitute the environment;

(c) the use(s) of environment(s) in `letsta` and in a call to a function defined by `letsta`;

(d) the use(s) of environment(s) in `letdyn` and in a call to a function defined by `letdyn`.

[8 marks]

Hint: because both `letsta` and `letdyn` functions may be applied using the same function call syntax, you may find it helpful to use separate forms of value for the two forms of functions.

1999 Paper 4 Question 4

Compiler Construction

It is commonly suggested that Algol-60 call-by-name can be modelled by passing a function as a call-by-value parameter. Show how a program containing a definition

```
    int f(int x:name) { ... x ... x ... }
```

of `f` (where `x` occurs only in Rvalue context) and a call `f(`*e*`)` to `f` can be replaced by an equivalent definition and call using only call-by-value. [6 marks]

Most such explanations assume that the uses of `x` within `f` occur only in Rvalue context. However, Algol-60 also permits the equivalent of

```
    int g(int x:name) { if (p) { ... x := x+1; x := -x; ... }
                            return x;
                      }
```

and calls like `g(a[k()])` which, when `p` is `true`, would have the effect of calling `k()` five times and consequent access to five (possibly different) subscripts of array `a[]`. Develop your explanation for the first part of this question to cover also the case of a call-by-name parameter being used in both Lvalue and Rvalue contexts. [Hint: note that when `p` is `false` then the actual parameter to `g` need not be an Lvalue, so you may need two parameterless procedure arguments ("thunks").]

[8 marks]

Using the previous part or otherwise, give a translation of a definition and call `h(`*e*`)` using call-by-value-result (Ada `in out` mode) with no uses of the address-of (`&`) operator other than those involved in call-by-name. Your explanation is allowed to deviate from call-by-value-result by allowing side-effects in *e* to take place twice.

[6 marks]

1999 Paper 5 Question 6

Compiler Construction

Explain a possible implementation method for Java-style or ML-style exceptions and handlers. [8 marks]

Consider a simple arithmetic expression e of abstract syntax:

$$e ::= x \mid n \mid e + e' \mid e - e' \mid e * e' \mid e/e' \mid -e$$

where x ranges over a set of (global) variables, addressable by name, and n ranges over integer constants. Write a procedure in pseudo-code or a language of your choice which takes an expression e and prints (one-per-line) stack-machine instructions of the form

```
pushvar  x
pushnum  n
add         ; pop two items and push their sum
sub         ; pop two items and push their difference
mul         ; pop two items and push their product
div         ; pop two items and push their quotient
neg         ; replace top item with its negation
```

which, when executed, have the net effect of pushing just the value of e onto the stack. Each line of code emitted should contain a comment giving the number of items on the stack after its execution, thus the first `push` and the last instruction would both be commented with "1 item". [12 marks]

1999 Paper 6 Question 6

Compiler Construction

It is desired to obtain an unambiguous context-free grammar G' which generates the same strings as the following grammar G with start symbol `S`.

```
S -> E
E -> ( E ) | [ E ] | E * E | a | b | c
( E ) -> ( + E )
[ E ] -> [ - E ]
```

Define a suitable G' or explain why G already satisfies the criterion. [6 marks]

Write a context-free (Type 2) grammar which describes floating-point numbers of the form $[\pm]dd^*[.d^*][e[\pm]dd^*]$ where d stands for decimal digit and d^* stands for zero or more decimal digits. $[\cdots]$ means that the enclosed item is optionally present in the floating-point number. [7 marks]

Sketch a recursive descent parser for the following grammar H with start symbol `S`. You should assume the existence of a routine `lex()` which sets variable `token` to one of '`1`', '`2`', '`(`', '`)`', '`-`' or `eof`.

```
P -> 1 | 2 | (E)
E -> P | E - P
S -> E eof
```

[7 marks]

2000 Paper 3 Question 3

Compiler Construction

With reference to a strictly-typed block-structured programming language, write brief notes on the following topics:

(*a*) the allocation and recovery of records stored in a heap

(*b*) the implementation of variables of union type

(*c*) the allocation of arrays with non-manifest bounds

(*d*) the implementation of labels and GOTO commands

[5 marks each]

2000 Paper 4 Question 4

Compiler Construction

Consider the grammar

```
S -> E <eof>
E -> T + E
E -> T
T -> x
```

where `S` is the starting symbol, `<eof>` is a special token marking end of input and `x` is a terminal.

Explain and find the left, right and follow sets for all non-terminals in the grammar. [5 marks]

Suppose that an SLR parser for this grammar is required. One stage on the way to constructing the parsing tables is to create the *characteristic finite state machine* (sometimes known as the LR(0) states). Do this, explaining your working clearly. You do not need to complete the SLR parsing tables. [10 marks]

Now, assuming that the parsing tables have been constructed, show what values will be placed on a stack and comment about internal state while an SLR parser using this grammar processes the input text `x+x+x<eof>`. [5 marks]

Compiler Construction

Give a brief description of the main features of *either* Lex and Yacc *or* the corresponding Java tools JLex and Cup. [5 + 5 marks]

Illustrate their use by outlining how you would construct a parser for expressions composed of identifiers, integers, unary minus and binary operators $+$, $-$, $*$ and $/$. Your parser is expected to create a parse tree in a format of your choice representing the expression that is presented to it. If it helps, you may assume that expressions will be terminated by writing a semicolon after them. [10 marks]

2000 Paper 6 Question 6

Compiler Construction

Describe how a parse tree can be translated into a sequence of assembly language instructions based on a pattern matching graph derived from a set of tree rewriting rules where each rule has a cost and a corresponding fragment of code. Illustrate your answer using the following rules:

```
Ri = Kk                     LDI  Ri,Kk     Cost 2
Ri = add(Ri,Kk)             ADDI Ri,Kk     Cost 3
Ri = add(Ri,Rj)             ADD  Ri,Rj     Cost 3
Ri = add(Ri,add(Rj,Kk))     ADD  Ri,Rj,Kk  Cost 4
```

applied to the following parse tree:

`add(K1,add(add(K2,add(K3,K4)),add(K5,K6)))` [15 marks]

Discuss the advantages and disadvantages of this approach to code generation.

[5 marks]

2001 Paper 3 Question 3

Compiler Construction

A *regular grammar* is a grammar whose rules are in one of the two following forms (where A and B are non-terminal symbols and a is a terminal):

$$\begin{aligned} A &\to a \\ A &\to aB \end{aligned}$$

(*a*) Give a regular grammar which generates floating point numbers of exactly the following form:

$$(0|1)^{+}.(0|1)^{*}[e(0|1)^{+}]$$

where "()" indicates grouping, "[ ]" indicates optional item, "ρ^{+}" indicates one or more repetitions of ρ and "ρ^{*}" indicates zero or more repetitions of ρ. [8 marks]

(*b*) Give a non-regular grammar with fewer productions than your answer to (*a*) but which generates the same set of strings. [4 marks]

(*c*) Determine, with justification, for the following grammars

(*i*) whether S generates strings not generated by T; and

(*ii*) whether T generates strings not generated by S.

$$\left\{ \begin{array}{l} S \to aaS \\ S \to Scc \\ S \to d \end{array} \right\} \text{ and } \left\{ \begin{array}{l} T \to aTc \\ T \to d \end{array} \right\}$$

[4 marks]

(*d*) What is the significance for the compilation process of the idea of "strings which can be generated by regular grammars"? Your answer should explain where such a module recognising such strings would appear in a compiler and a possible external interface (functions, variables and/or objects) it might present to the rest of the compiler. [4 marks]

2001 Paper 4 Question 4

Compiler Construction

Consider the following grammar giving the concrete syntax of a language:

$$\begin{array}{lcl}
E & \to & id \\
C & \to & E = E; \\
C & \to & \{B\} \\
C & \to & C \textbf{ repeatwhile } E \\
C & \to & \textbf{if } E \textbf{ then } C \\
C & \to & \textbf{if } E \textbf{ then } C \textbf{ else } C \\
B & \to & B\ C \\
B & \to & C \\
S & \to & C\ eof
\end{array}$$

where C **repeatwhile** E has the same meaning as **do** C **while** E in C or Java.

(*a*) List the terminals and non-terminals of this grammar and explain the significance of S. [3 marks]

(*b*) Identify any ambiguities in the above grammar and rewrite it to remove them, ensuring that your new grammar generates exactly the same set of strings. [4 marks]

(*c*) Specify a suitable abstract syntax, for example by giving a type declaration in a programming language of your choice, which might be used to hold parse trees for this language. [3 marks]

(*d*) Give *either* a recursive descent parser *or* a characteristic finite state machine (e.g. for SLR(1)) with associated parser for your grammar. Your parser need not return a parse tree—it suffices for your parser either to accept or to reject the input string. [10 marks]

2001 Paper 5 Question 6

Compiler Construction

Consider a language J which has

- Java-like syntax
- nested definitions of procedures within other procedures
- local variables (with static binding)
- raising and handling of named, parameterless exceptions

Explain a possible run-time data structure which a compiler for J might use. [10 marks]

A naïve user of such a language suggests that the resultant compiled code will spend a significant fraction of execution time searching—both finding the store location corresponding to the use of a variable name and finding the exception handler corresponding to the raising of a given exception name.

Determine with justification whether this is so for your run-time data-structure proposed above. [4 marks]

Now instead suppose a simple interpreter for J is written, so that searches for variable (or exception) names search the appropriate environment for the variable value or exception handler code. To what extent are these searches bounded by (a) the number of variables or exceptions in the program or (b) the dynamic or static nesting of procedures? [6 marks]

Compiler Construction

(*a*) Describe one possible structure (e.g. ELF) of an object file. Illustrate your answer by considering the form of object file which might result from the following C program.

```
int a = 1, b = -1;
extern int g(int);
extern int c;
int f() { return g(a-b) + c; }
```

It is not necessary to consider the exact instruction sequence, just issues concerning its interaction with the object file format. [10 marks]

(*b*) Describe how a linker takes a sequence of such programs and produces an executable file. [4 marks]

(*c*) Compare and contrast *static* and *dynamic* linking in a system using your object file format. [6 marks]

2002 Paper 3 Question 1

Compiler Construction

(*a*) Give a diagram showing the phases of a typical compilation system for a language like C which produces a directly executable fully-linked binary file as output. For each phase, describe in a paragraph what it does (mentioning possible implementation techniques) and give a brief overview of the data-structures used for its input and output indicating whether they would normally reside in a file or in memory. (Do not specify details of any files used to automate the writing of any of the above phases.) [16 marks]

(*b*) Indicate how a typical Java implementation might differ and explain what is meant by *just-in-time compilation*. [4 marks]

2002 Paper 4 Question 2

Compiler Construction

The specification for a pocket-calculator-style programming language is as follows:

- Valid inputs consist either of an Expression followed by the $\boxed{\texttt{enter}}$ button or of an Expression followed by $\boxed{\texttt{store}}$ Identifier $\boxed{\texttt{enter}}$;

- Expressions consist of Numbers and Identifiers connected with the binary operators $\boxed{+}$, $\boxed{\times}$ and $\boxed{\uparrow}$ (in increasing binding power), with the unary operators $\boxed{-}$ and $\boxed{\texttt{abs}}$, and possibly grouped with parentheses. Unary operators bind more strongly than $\boxed{+}$ but weaker than $\boxed{\times}$ so that $-a + b$ means $(-a) + b$ but $-a \times b$ means $-(a \times b)$;

- Numbers consist of a sequence of at least one digit, possibly interspersed with exactly one decimal point, and possibly followed by an exponential marker "`e`" followed by a signed integer, e.g. `6.023e+22`. Identifiers are sequences of lower-case letters.

(*a*) Give a Context-Free Grammar (Type 2 in the Chomsky Hierarchy) for the set of valid input sequences using names beginning with an upper-case letter for non-terminals. It should be complete in that you should go as far as to define e.g.

```
Letter ::= a | b | ... | z
```

[10 marks]

(*b*) Indicate, giving brief reasoning, which non-terminals are appropriate to be processed using lexical analysis and which using syntax analysis proper. [5 marks]

(*c*) Give yacc or CUP input describing those elements deemed in part (*b*) to be suitable for syntax analysis. You need not give "semantic actions". [5 marks]

2002 Paper 5 Question 6

Compiler Construction

(*a*) Assuming a Java type is given to each variable, state a method by which an overloaded operator (such as `+`,`-` etc.) in a Java program can be determined to be an `int`, `real` or other operator. [3 marks]

(*b*) Explain, using pseudo-code as appropriate, how to convert a syntax-tree into stack code such as that used in the JVM. For the purposes of this question, you only need consider trees representing bodies of `void`-returning functions, and these bodies only as consisting of a list of statements of the form `int` x `=` e`;` or $x = e$; where x ranges over variables and e over expressions; expressions contain variables, integer constants, the binary operator `+` and static method invocations. [10 marks]

(*c*) Show how a sequence of simple stack code instructions, such as those used in your answer to part (*b*) above, can be translated into a sequence of instructions for a register-oriented architecture of your choice, for example ARM or Pentium. [7 marks]

2002 Paper 6 Question 6

Compiler Construction

Explain a possible implementation technology for Java classes and objects. Your answer should focus on storage layout for objects and on how class variables and methods are accessed—it is not necessary to explain access qualifiers such as `public` and `private`. Illustrate your answer with the following program; in particular indicate its eventual output.

```
class test {
public int n;
public static int s = 100;
public void f(int x) { System.out.println("f1 " + (x+n)); }
public static void main(String args[]) {
        test p = new test();
        test2 q = new test2();
        test r = q;
        p.n = 4;
        q.n = 5;
        q.m = 6;
        r.n = 7;
        p.f(p.s);
        q.f(p.s);
        r.f(q.s);
     }
}

class test2 extends test {
public int n, m;
public static int s = 200;
public void f(int x) { System.out.println("f2 " + (x+n+m)); }
}
```

[20 marks]

2003 Paper 4 Question 1

Compiler Construction

A programming language has commands C and expressions E which may involve the terminals I (identifiers) and N (integer constants). Its grammar $\mathcal{G}$ (with start symbol S) is given by

$$
\begin{aligned}
S &::= C \ \texttt{eof} \\
C &::= I = E \mid \texttt{if}\ E\ \texttt{then}\ C \mid \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C \\
E &::= I \mid N \mid E + I
\end{aligned}
$$

Construct

(*a*) a recursive descent parser, and [8 marks]

(*b*) the characteristic finite state machine (CFSM) of a LR(k), SLR(k) or LALR(k) parser, [12 marks]

explaining carefully whether there are any problems in the grammar $\mathcal{G}$ (and if so how you resolved them) and in adapting the grammar for recursive descent parsing and in adopting the CFSM so as to be suitable for LR(k), SLR(k) or LALR(k) parsing. It is not necessary to consider how the corresponding parse tree is constructed.

2003 Paper 5 Question 6

Compiler Construction

(*a*) A Java static method is defined in `class C` by

```
class C {
  public static int f(int x, int y) { int z = x; ...; return x+y*z;
                                    }
}
```

where '`...`' represents commands the details of which are not important to this question. It is called in an expression *e* of the form

```
  f(f(1,2), f(3,4))
```

Give JVM (or other stack machine) code corresponding to the expression *e* and explain how this is derived from the syntax tree for *e*. [6 marks]

(*b*) Explain how the body of `f` above is mapped into JVM (or other stack machine) code, explaining the rôle of the registers FP and SP (precise details are not important, but their rôle should be well explained). You may write '`...`' for the translation of the '`...`' in `f`. [6 marks]

(*c*) Consider the Java class definitions:

```
class A {
  public int a1, a2;
  public void m() { println("I am an A with " + a1 + " and " + a2);
                  }
}
class B extends A {
  public int b1, b2;
  public void m() { println("I am a B with " + a1 + " and " + a2 +
                             " also with " + b1 + " and " + b2);
                  }
}
```

Describe the run-time storage layout for objects of `class A` and for those of `class B`, particularly noting the size and offsets of members and how a *cast* of an object of type `class B` to one of `class A` can be achieved.

Explain how calls to `m()` work, particularly in code like:

```
  public static void g(B x) { h(x); }
  public static void h(A x) { x.m(); }
```

[8 marks]

2003 Paper 6 Question 6

Compiler Construction

Each of the following statements may be true, false, or nonsensical. Indicate which and (respectively) provide a (one-sentence) justification of why it holds, a counter-example or other explanation of why it fails, or corrected statement.

(*a*) In a language with stack-allocated free variables, the static chain pointer always points to the caller's stack frame.

(*b*) If each procedure sets up one exception handler whose scope includes all of the procedure calls it makes, then the entries in the exception handler stack and those in the dynamic chain will be in 1–1 correspondence.

(*c*) Fortran gives semantic meaning to parentheses, i.e. `a+b+c` permits compiler re-arrangement whereas `(a+b)+c` does not. This matters for floating point. However, since Fortran '`+`' is left associative the above two expressions yield the same parse tree, and we have a contradiction.

(*d*) Two alternatives to conservative garbage collection are liberal garbage collection or new labour style de-allocation.

(*e*) Any type 3 (regular) grammar is also a type 2 (context-free) grammar. Hence any lexer generated by `lex` (or `JLex`) could instead be generated by `yacc` (or `Cup`).

(*f*) When a compiler for a language like Java compiles e_1+e_2, it computes the types of e_1 and e_2 so that it can treat it as e_1+`(float)`e_2 if e_1 is of type `float` and e_2 is of type `int`.

(*g*) A syntax-tree interpreter can fail (give an error) when evaluating a variable name which does not appear in the current environment. Any program which fails using static scoping will fail using dynamic scoping.

(*h*) A syntax-tree interpreter can fail (give an error) when evaluating a variable name which does not appear in the current environment. Any program which fails using dynamic scoping will fail using static scoping.

(*i*) A dynamically-typed language is one in which the type of a value is carried around at run-time; a type error is given at run-time if values are used inappropriately. Such languages must be dynamically linked otherwise type errors will occur.

(*j*) Java `.class` files and ELF-style `.o` (or `.obj`) object files represent similar information, i.e. compiled code, a list of symbols made available to other functions, and a list of undefined symbols which the file expects to be defined elsewhere.

[2 marks each]

2004 Paper 4 Question 1

Compiler Construction

(*a*) A context-free grammar can be formally defined as a 4-tuple. Give a precise statement of what the components are. [2 marks]

(*b*) Explain the difference between a grammar and the language it generates. [2 marks]

(*c*) Explain what makes a grammar ambiguous, with reference to the grammar which may be commonly expressed as a "rule"

$$E ::= 1 \mid 2 \mid X \mid E + E \mid E * E \mid -E$$

where X is an identifier. [2 marks]

(*d*) For the "rule" in part (*c*), give a formal grammar containing this "rule" and adhering to your definition in part (*a*). [2 marks]

(*e*) Give non-ambiguous grammars each generating the same language as your grammar in part (*d*) for the cases:

(*i*) "$-$" is most tightly binding and "$+$" and "$*$" have equal binding power and associate to the left.

(*ii*) "$-$" is most tightly binding and "$+$" and "$*$" have equal binding power and associate to the right.

(*iii*) "$-$" binds more tightly than "$+$", but less tightly than "$*$", with "$+$" left-associative and "$*$" right-associative so that "$-a + -b * c * d + d$" is interpreted as "$((-a) + (-(b * (c * d)))) + d$".

[2 marks each]

(*f*) Give a simple recursive descent parser for your grammar in part (*e*)(*iii*) above which yields a value of type `ParseTree`. You may assume operations *mkplus*, *mktimes*, *mkneg* acting on type `ParseTree`. [6 marks]

2004 Paper 5 Question 7

Compiler Construction

You have been provided with the description of a programming language, J, intended for scripting applications. Its syntax is similar to a cut-down version of Java in that it consists of function definitions which have bodies containing if-then-else, while-do, assignments and (typed) declarations of variables. Only one statement or keyword may occur on a line so that it is sufficient to describe the start of a loop iteration with its line number. You need to explain to your boss the alternatives for implementing this so that a decision may be made as to the best implementation strategy.

The choice is between:

(*a*) compiling J to machine code;

(*b*) compiling J to "interpreted byte code", and then interpreting this;

(*c*) parsing J to a syntax tree representation and then interpreting this using a function which walks the tree;

(*d*) keeping J in a text file and then interpreting it by reading each line (and acting on it) as and when the line is required.

For *each* of (*a*)–(*d*), (*i*) summarise the main phases of work that are done *before* execution in each case, giving a brief explanation of the main actions of the main interpreter loop (if any) *during* execution, and (*ii*) for each of the following possible erroneous forms, explain whether the error would be found before or during execution: malformed syntax, undeclared variable, type error, division by zero.

[5 marks each]

You are not expected to argue for or against any of the alternatives.

2004 Paper 6 Question 8

Compiler Construction

(*a*) Explain the differences (illustrating each with a small program) between

(*i*) static and dynamic binding (scoping); [4 marks]

(*ii*) static and dynamic typing. [2 marks]

(*b*) Java is sometimes said to be "dynamically typed" in that a variable whose type is (class) C can be assigned a value of (class) D provided that D extends C; conversely a variable of type D can be assigned a value of type C using a cast. By considering storage layouts, explain why the former assignment is always valid and the latter *sometimes* invalid. [4 marks]

(*c*) A new programming language has the notion of "statically scoped exceptions" in which the program

```
exception foo;
void f()
{   try
    {   void g() { raise foo; }
        try {
          g();
        }
        except (foo) { C2 }
    }
    except (foo) { C1 }
}
```

would execute `C1` rather than `C2` as the former was in scope at the `raise` point. By analogy with statically scoped variables, or otherwise, explain how such exceptions might be implemented on a stack. [10 marks]

2005 Paper 4 Question 1

Compiler Construction

(*a*) A (phrase-structured) grammar is often defined to be a 4-tuple (N, T, R, S) where R is a set of production rules. Explain what the other components of the 4-tuple are. Explain also the (most general form of) production rules, how these are conventionally restricted and why one might wish to restrict them. [6 marks]

(*b*) Give a grammar which is ambiguous. [2 marks]

(*c*) Give a grammar which is not a regular grammar but which generates a regular language containing an infinite number of strings. [2 marks]

(*d*) Is it possible to write a grammar which generates the strings $\{aa, aaa, aaaaa, \ldots, a^p, \ldots\}$ where p is prime? (A general argument for or against suffices.) [2 marks]

(*e*) It is desired to construct a simple "pocket-calculator" program using `yacc` and `lex` (or other similar automated tools of your choice) which can parse strings such as "`1+(10-5-3)*5+2=`" and print the result, 13 in this case. Outline the overall structure of your program components. Give full details of the input to `yacc` and `lex` (or equivalent). (Precise syntactic details are not important, but your answer should show an understanding of the principles involved.) [8 marks]

2005 Paper 5 Question 6

Compiler Construction

(*a*) Explain how a parse tree representing an expression can (*i*) be converted into stack-oriented intermediate code and then (*ii*) be translated into simple machine code for a register-oriented architecture (e.g. ARM or IA32) on an instruction-by-instruction basis. Also indicate how this code might be improved to remove push–pop pairs introduced by (*ii*). Your answer need only consider expression forms encountered in the expression:

```
h(a, g(b), c) * 3 + d
```

[12 marks]

(*b*) In Java, expressions are evaluated strictly left-to-right. Consider compiling the function `f` in the following Java class definition:

```
class A
{
    static int a,b;
    void f() { ... <<C>> ... }
    int g(int x) { ... a++; ... }
};
```

Indicate what *both* the intermediate code *and* (improved as above) target code might be for `<<C>>` for the cases where `<<C>>` is:

(*i*) `b = g(7) + a;`

(*ii*) `b = a + g(7);`

(*iii*) `b = (-g(7)) + a;`

(*iv*) `b = a - g(7);`

Comment on any inherent differences in efficiency at both the intermediate code and target code levels.

[8 marks]

2005 Paper 6 Question 5

Compiler Construction

(*a*) Give a program which gives different results according to whether dynamic or static binding is used. [3 marks]

(*b*) Summarise two different methods of implementing functions which have free variables statically bound in an outer nested function, paying particular attention to (*i*) the lifetime of any storage used and (*ii*) any language restrictions which the implementation requires. Note also, for each case, what the effect is of assignment to such free variables, e.g. when calls to `g()` and `h()` are interleaved in

```
f(int x)
{   int a = x+7;
    int g() { a++; }
    int h(int y) { return a+y; }
    ...
}
```

[8 marks]

(*c*) Give a program in which a function is called recursively without any value or function being defined recursively. [3 marks]

(*d*) Give a program which produces three distinct results according to whether parameters are passed by value, by reference, or by value-result. [3 marks]

(*e*) Explain the likely storage layouts for Java objects *p* and *q* respectively of classes `P` and `Q` defined by

```
class P           { int a,b; int f() { <body1> } }
class Q extends P { int c; int f() { <body2> }
                           int g() { <body3> } }
```

[3 marks]

2006 Paper 4 Question 2

Compiler Construction

Consider the following grammar.

$$E ::= \mathrm{NUM} \mid E \times E \mid E/E \mid E + E \mid E - E \mid (E)$$

(*a*) Show that this grammar is ambiguous. [2 marks]

(*b*) Rewrite the grammar to eliminate ambiguity. [4 marks]

(*c*) Transform the grammar of part (*b*) into an equivalent one that could be used for recursive descent parsing. Explain your answer. [5 marks]

(*d*) Write a recursive descent parser for your grammar. [9 marks]

2006 Paper 5 Question 6

Compiler Construction

(*a*) Consider the grammar

$$S ::= (L) \mid a$$
$$L ::= L, S \mid S$$

(*i*) Present a right-most derivation for the string $(a,\ ((a,\ a),\ (a,\ a)))$. [3 marks]

(*ii*) Present a left-most derivation for the same string $(a,\ ((a,\ a),\ (a,\ a)))$. [3 marks]

(*b*) Automatic garbage collection is an important technique for the implementation of many programming languages. Define each of the following variations:

(*i*) Mark and Sweep; [3 marks]

(*ii*) Copy Collection; [3 marks]

(*iii*) Generational Collection. [3 marks]

(*c*) Write a small program that will produce different values depending on which kind of variable scoping mechanism is used, static or dynamic. Explain your answer. [5 marks]

2006 Paper 6 Question 6

Compiler Construction

(*a*) Describe a difference and a similarity between the notions of *overloading* and *polymorphism*. [2 marks]

(*b*) Define the notion of *type safety* in a programming language. [2 marks]

(*c*) Describe the linking phase and the difference between static and dynamic linking. [2 marks]

(*d*) Suppose that a programming language allows nested functions. How might a stack-based implementation of `f` access the value associated with `a`?

```
int g(int a)
{
    int f(int x) {
      return a + x;
    }
    ...
    f(2);
    ...
}
```

[4 marks]

(*e*) Describe the low-level code that could be generated by compiling the following code fragment for a stack-based target machine.

```
int f(int a, int b)
{
  int z = a * b;
  return a + z;
}
...

  f(1, 2) * f(3, f(4, 5));

...
```

[10 marks]

2007 Paper 4 Question 4

Compiler Construction

(a) Lexical analysis is an important first step in compilation.

(i) Define in detail the abstract "lexing problem" that is solved by a lexical analyser. [4 marks]

(ii) Describe in detail how a lexical analyser can be automatically constructed from a list of regular expressions. [2 marks]

(b) Explain why LL(1) parsing is associated with left-most derivations of parse trees. [6 marks]

(c) Context-free grammars and ambiguity.

(i) Define when a context-free grammar is *ambiguous.* [2 marks]

(ii) A *left-recursive* context-free grammar production has the form $A \to A\alpha$. In the same way, a *right-recursive* production has the form $B \to \beta B$. (Both α and β are assumed to be non-empty sequences.) Suppose we have a grammar that contains both a left- and a right-recursive production for the same non-terminal, such as $A \to A\alpha$ and $A \to \beta A$. Is such a grammar always ambiguous? [6 marks]

2007 Paper 5 Question 6

Compiler Construction

(*a*) Describe how a stack is used to implement procedures and functions. [6 marks]

(*b*) Suppose a language allows the creation of pointers. How does this complicate the use of stacks as described in part (*a*)? [2 marks]

(*c*) How does the Java language deal with the problem described in part (*b*)? [2 marks]

(*d*) Consider the following ML-like program containing the function `g` that returns a function as a result.

```
let a = 17 in
let g b = (let h c = a + b + c in h) in
let f1 = g 21 in
let f2 = g 33 in
let v = f1(3) + f2(57) in
...
...
```

Explain carefully how such a program can be compiled. In particular, pay special attention to how the code for the body of the function `h` can access the values of `a`, `b`, and `c`. [10 marks]

2007 Paper 6 Question 6

Compiler Construction

(*a*) Garbage Collection.

(*i*) Explain how it is possible to "leak memory" using a reference counting garbage collector. [3 marks]

(*ii*) Describe any technique that might be used to address this problem. [3 marks]

(*b*) Explain in detail how we might translate code generated for a stack-only machine (such as the JVM) to a register-based machine (such as ARM). [6 marks]

(*c*) Describe in detail how static and dynamic methods are compiled differently for object-oriented languages with single inheritance. [8 marks]

2008 Paper 4 Question 3

Compiler Construction

(*a*) Given the following program fragment in C or Java

```
static int a = 3;
static int f(int x, int y) { int z = a+x; ...; return ... }
```

explain how the four variables (`a`, `x`, `y` and `z`) are accessed from within `f` at the instruction level for an architecture such as MIPS, ARM or x86. Pay particular attention as to how and when storage is allocated for these variables, and which system components of a standard compile-link-and-execute model are involved in selecting the instruction and determining the run-time address calculation. Your answer should briefly explain what occurs when `f` makes a recursive call to itself. [8 marks]

(*b*) Suppose we extend the language to allow nested function definitions:

```
static int a = 3;
static int f(int x) {
  static int g(int y) { int z = a+x; ...; return ... }
  return g(7);
}
```

(*i*) Complete the definition of `g` in such a way that, during execution of a call to `f`, the difference between the addresses allocated to `x` and `y` varies. [2 marks]

(*ii*) Explain a possible run-time mechanism that allows all four variables to be accessed from inside `g`. [6 marks]

(*c*) Suppose function-valued variables are added to the language, e.g.:

```
funvar double v(double);
v = (...) ? sin : cos;
```

and a pointer to code or to data is represented as a 32-bit value. How many bits might naturally be used to hold `v`? Justify your answer, emphasising the way in which it might differ according to whether the language is as given in part (*a*) or part (*b*) above. [4 marks]

2008 Paper 5 Question 6

Compiler Construction

Consider the following grammar for expressions (where `Id` is a terminal symbol representing an identifier resulting from lexical analysis):

```
Expr  ::=  1 | 2 | Id | Expr + Expr | Expr / Expr |
                        Expr ^ Expr | (Expr)
```

(*a*) Explain in what principal respect this grammar is unsatisfactory. [1 mark]

(*b*) Assuming further that `+` is to be left-associative, `^` is to be right-associative and `/` is to be *non-associative* (i.e. `2/2/2` is forbidden but `(2/2)/2` and `2/(2/2)` are allowed), re-write the grammar to reflect this. [4 marks]

(*c*) List the terminal symbols and non-terminal symbols, and count the production rules both in the original grammar and in the grammar in your answer to part (*b*). Indicate the *start symbol* in both grammars. [2 marks]

(*d*) Define a type or types (in C, Java, or ML) suitable for holding an abstract syntax tree resulting from your answer to part (*b*). [2 marks]

(*e*) Give a brief and elementary explanation of the principles of how the grammar resulting from part (*b*) might be used to create a syntax analyser taking a token stream as input (via calls to function `lex()`) and giving as output an abstract syntax tree corresponding to part (*d*). Mention both hand-written and automatically-generated syntax analysers. [8 marks]

(*f*) Summarise any issues related to left- or right-associative operators in the two techniques (in implementing the parser and in constructing the tool) you outlined in part (*e*). [3 marks]

Compiler Construction

(*a*) Languages like Lisp, Prolog and Python are said to be *dynamically typed.* Explain this concept and its implications for the size of a run-time storage cell needed to hold a value which may be an integer or floating-point value. [4 marks]

(*b*) Consider the following object-oriented program in Java style:

```
class A { int a,b; };
class B extends A { int c,d1,d2,d3,d4,d5,d6,d7,d8,d9; };
...
static void f(A x) { x.a = 1; }
static void g(B x) { x.c = 2; }
static void h() { A p = new A(); f(p); g(p); }
static void k() { B p = new B(); f(p); g(p); }
static void main() { h(); k(); }
```

(*i*) Explain the run-time structure of values of type `A` and `B`. Indicate a constraint on the layout of these structures needed to support *inheritance.* [3 marks]

(*ii*) Indicate why the above program would not compile in Java and insert a single *cast* to make it compile. Why are two casts not required? [2 marks]

(*iii*) What happens when your Java program in part (*ii*) is executed? [2 marks]

(*iv*) Make an analogy to part (*a*) to argue why a Java value of type `A` requires more storage than that required for two integers. [2 marks]

(*v*) C++ traditionally allows values of type `A` to occupy just the space required by two integers. Comment on the implications for safety if this were allowed in Java. [2 marks]

(*c*) Explain where storage for `new` comes from. Some languages have a primitive `dispose` which de-allocates space allocated by `new`, but Java does not. Explain the implications of this for Java implementation, particularly how a program can perform `new A()` every millisecond but never run out of memory. Suppose that, while executing such a program, a `new B()` is executed. Explain, giving reasons, whether this is guaranteed to succeed in a situation where exactly half the memory available for `new` is in use. [5 marks]

2009 Paper 5 Question 1

Compiler Construction

(*a*) Give *two* reasons why a compiler is typically broken down into *passes* (or *phases*) and indicate a typical set of passes and the data structures passed between them. [4 marks]

(*b*) Explain one way in which the process of compiling a Java applet, to be stored on a web page and run within many browsers, might differ from that of compiling an application and running it multiple times on a single machine. [2 marks]

(*c*) Explain the difference between a file containing textual assembly code and a file containing an object module. Indicate the essential structure of an object file, for example ELF, highlighting the purpose of each section. Give assembly-level or C-level code that produces entries in each of these sections. [7 marks]

(*d*) Explain the actions of a *linker*, the role of libraries and what makes certain object files directly executable. [4 marks]

(*e*) Explain the difference between *static* and *dynamic linking* giving one advantage of each. Which is most appropriate for an applet stored on a web site to be loaded into a browser? [3 marks]

2009 Paper 5 Question 2

Compiler Construction

Consider an ML-like language in which the set of values includes functions and these functions may have nested definitions.

(*a*) Explain how a function value may be represented at run-time, both in a syntax-tree interpreter and in compiled code. What is the importance of the word "nested" above? [3 marks]

(*b*) Give an example program that gives different results in static and dynamic scoping. In each case, explain how an interpreter or compiled code might perform function application. [3 marks]

(*c*) Considering only static scoping from now on, explain what restrictions are necessary to implement function values without using an auxiliary heap. Explain the notions of static and dynamic chains, giving an example of a situation in which they differ (i.e. reference to the wrong one would access the wrong variable). Illustrate how updating (by assignment to) a free variable can be implemented. [5 marks]

(*d*) Give an alternative heap-based implementation for function values when the restrictions in part (*c*) do not hold. Explain how free variables (and particularly their update) can now be implemented. [4 marks]

(*e*) Java normally holds local variables in stack frames and instance variables on the heap. Consider a Java-like language with nested classes, and a possible program of the form:

```
class C {
  int f(int x)
   { class D { int addx(int y) { return x+y; }
               void updtx(int y) { x = y; }
             }
      ...
   }
};
```

Explain a possible implementation of such nested classes based on your answers above; also give and justify a restriction on methods like `updtx` that eases the cost of implementation. [5 marks]

2010 Paper 3 Question 3

Compiler Construction

(*a*) What are the principal features of a regular language and of a context-free language and their respective parsers? [4 marks]

(*b*) When implementing a compiler, why is a regular language commonly used for lexical analysis and why are context-free grammars commonly used for the main syntax analysis phase? [3 marks]

(*c*) Give **two** main features that distinguish a recursive-descent parser from an SLR(k) parser (or similar). [2 marks]

(*d*) The C++ language uses the "`>>`" character sequence to denote the right-shift operator. This character sequence can also appear when a template (generic) type takes another template as its argument, as in "`stack<list<int>>`" that is supposed to denote a stack of integer lists. What problem can this cause? [1 mark]

Comment on how you might solve this problem

(*i*) in the syntax analyser; and [2 marks]

(*ii*) in the lexical analyser. [2 marks]

Comment on the relative elegance of the two solutions. [1 mark]

(*e*) A common language construct is "`T v;`" to declare a variable called "`v`" or type "`T`". What parsing problem can arise if "`T`" is a user-defined type? Explain how the lexical analyser could benefit from feedback from the syntax analyser in this situation. [5 marks]

2010 Paper 3 Question 4

Compiler Construction

(*a*) How is inheritance of class variables typically implemented by a compiler for an object-oriented language in which each class has exactly one parent? [3 marks]

(*b*) When supporting multiple inheritance, what extra run-time complexity arises when casting or coercing an object handle up or down and why is this avoided using the Java concept of an interface? [3 marks]

(*c*) When supporting multiple inheritance, what identifier clash problem can arise and what are the possible solutions? [4 marks]

(*d*) Explain, using an example, the potential for an erroneous downcast in an object-oriented language such as Java, C++ or C#. [4 marks]

(*e*) Sketch assembly-level code that

(*i*) detects an erroneous downcast;

(*ii*) finds an exception handler for it; and

(*iii*) correctly jumps to the handler.

Assume that the handler is already registered with the run-time system and that there are no arguments to the exception. Use any well-known implementation techniques, such as keeping an object's identity in its virtual method table and stacking exception handlers on an extra run-time stack. [6 marks]

2011 Paper 3 Question 4

Compiler Construction

(*a*) In a stack-based runtime system, what problem does the *static link method* attempt to solve, and how does it work? [4 marks]

(*b*) Can static linking be used to implement a language with first-class functions? If yes, then explain how. If no, give an example and explain how static linking fails. [6 marks]

(*c*) Explain how exceptions (ML-like `raise` and `handle`) could be implemented with a stack-oriented machine. [5 marks]

(*d*) A program may evaluate to an exception that has been raised all the way to the top-level and never handled. Discuss how you might modify your implementation in part (*c*) to dump debugging information when such top-level exceptions are raised. The debugging information should include some description of the state of the computation *just before the top-level exception was raised.* [5 marks]

2011 Paper 3 Question 5

Compiler Construction

Consider a simple grammar for arithmetic expressions:

$$E ::= n \mid x \mid -E \mid E\ +\ E \mid (\ E\ )$$

with n ranging over integer constants and x ranging over variables. We want to compile expressions to code for a simple stack-based machine with the following instruction set.

instruction	meaning
`pushvar` k	push the value of the k-th variable on top of stack
`push` n	push n on top of stack
`add`	replace the top two stack items with their sum
`neg`	replace the top stack item with its negation

For this problem, we will not worry about how variables are bound to values nor how abstract syntax trees are produced.

(*a*) How will your compiler generate code from expressions of the form $-E$? [4 marks]

(*b*) How will your compiler generate code from expressions of the form $E_1\ +\ E_2$? [4 marks]

(*c*) What code will your compiler generate for the expression `-(-x + (17 + y))`? [4 marks]

(*d*) Suppose we now want to extend the language of integer expressions with multiplication

$$E ::= \cdots \mid E\ *\ E$$

but we cannot extend the machine with an instruction for multiplication.

Can you implement this extended language directly with the machine instruction set presented above? If not, suggest a *minimal* extension to the instruction set that allows for the implementation of multiplication using the addition from the instruction set. Explain the semantics of your extensions and how you would use them to implement multiplication. [8 marks]

4 Compiler Construction (DJG)

(*a*) Define the following terms used when discussing a grammar:

(*i*) a non-terminal symbol [1 mark]

(*ii*) an ambiguous grammar [1 mark]

(*iii*) a production [1 mark]

(*iv*) a context-free grammar [2 marks]

(*v*) a regular grammar [2 marks]

(*b*) The following grammar defines a language where expressions are strings or integers. A type error arises when an integer is added to a string.

```
Var -> x | y
Exp -> Var | 0 | 1  | "cat" | "dog" | Exp + Exp
S   -> Var := Exp | S S
```

(*i*) Give a syntactically-correct sentence of the language that contains a type error. [1 mark]

(*ii*) What phase (or pass) of a typical, simple compiler would detect such a type error? Sketch the fragment of code that actually spots the error. [3 marks]

(*iii*) Provide a modified grammar such that type errors are also syntax errors. [3 marks]

(*iv*) Why is such a modified grammar generally impractical? [1 mark]

(*c*) Certain operators, such as **logical and**, which is commonly denoted with **&&** use short-circuit evaluation.

(*i*) Define short-circuit evaluation. [2 marks]

(*ii*) Give two reasons why it is useful. [1 mark]

(*iii*) Describe the problem, and its solution, that arises when a short-circuit operator is encountered during a simplistic compilation of a syntax tree to stack machine code. [2 marks]

5 Compiler Construction (DJG)

(*a*) Define the following forms of variable: [1 mark each]

(*i*) statically allocated global variable

(*ii*) local variable (to a function)

(*iii*) free variable

(*b*) In an object-oriented language, which of the above three terms best describes a field? [1 mark]

(*c*) How are storage addresses typically allocated for each of these types of variable? Describe the stage/pass of the compiler that makes the allocation and also describe any changes made by the operating system linker or loader. [3 marks]

(*d*) What addressing mode or sequence of instructions is typically used in compiled code to read (or write) each of the variable forms from part (*a*)? [1, 1, 4 marks]

(*e*) With dynamic storage allocation, how does the memory manager 'new' operator know how much memory to allocate? Discuss both compilation to machine code and execution on a VM. [2 marks]

(*f*) Why is garbage collection easier to implement in strongly-typed languages (compared with weakly-typed languages)? [1 mark]

(*g*) What overheads does allocating arrays and class objects on the stack cause compared with placing them on the heap? What are the advantages and disadvantages of each method? [4 marks]

4 Compiler Construction (TGG)

(*a*) Describe the costs and benefits of performing inline expansion of functions during compilation. [4 marks]

(*b*) Describe what is meant by *eliminating tail recursion*, when such an optimization can be applied and why it is a benefit. [4 marks]

(*c*) Consider the following ML-like program where the function `f` returns a function:

```
let val a = 99 in
let fun  f b = let g c = a + b + c in g end
let val f1 = f 17 in
let val f2 = f 33 in
let val v = (f1 a) + (f2 a) in
...
...
```

Describe carefully how this program fragment could be compiled. Explain how the expression

```
(f1 a) + (f2 a)
```

would be evaluated by your compiled code. [12 marks]

5 Compiler Construction (TGG)

(*a*) When is it useful to eliminate left-recursion from a grammar and why? [2 marks]

(*b*) Write a recursive descent parser for the language generated by the following grammar.

```
E ::= E + F
      |  E - F
      |  F

F ::= NUM
      | ID
      | ( E )
```

[6 marks]

(*c*) This section deals with how object-oriented classes are typically implemented by a compiler when only simple inheritance (each class has exactly one parent) is supported.

(*i*) Describe in detail how objects are represented in memory and how this representation captures inheritance of attributes. [6 marks]

(*ii*) Discuss how virtual methods can be implemented. [6 marks]

4 Compiler Construction (TGG)

This question concerns the run-time call stack.

(*a*) What is a *run-time stack* and why is it important to a compiler writer? [3 marks]

(*b*) The implementation of a run-time call stack typically uses a *stack pointer* and a *frame pointer*. What are their roles and why do we need two pointers? [3 marks]

(*c*) For some compilers the activation records (stack frames) contain *static links*. What problem are static links used to solve and how do they solve this problem? [3 marks]

(*d*) (*i*) Consider a programming language that does not allow functions to be returned as results, but does allow the nesting of function declarations. Using ML-like syntax, we have the following code in this language.

```
let fun f(x) =
    let
        fun h(k) = k * x

        fun g(z) = h(x + z + 1)
    in
       g(x + 1)
    end
in
    f(17)
end
```

Draw a diagram illustrating the call stack from the call of `f` up to and including the call of function `h`. Make sure all function arguments are included in the diagram and clearly indicate static links. [5 marks]

(*ii*) Using your diagram, explain how the code generated from the body of function `h` can access the values associated with the variables `k` and `x`. In each case make it clear what information is known at compile-time and what information is computed at run-time. [6 marks]

5 Compiler Construction (TGG)

Functional programmers will often rewrite a recursive function such as

```
fun fact1 n =
    if n <= 1
    then 1
    else n * (fact1 (n -1))
```

to one such as

```
fun fact2 n =
let fun aux (m, a) =
        if m <= 1
        then a
        else aux(m-1, m * a)
in aux (n, 1) end
```

using an accumulator (the parameter `a` of `aux`) and *tail recursion.*

(*a*) Clearly explain the optimisation such programmers are expecting from the compiler and how that optimisation might improve performance. [4 marks]

(*b*) The desired optimisation can be performed by a compiler either directly on the source program or on lower-level intermediate representations. Treating it as a source-to-source transformation, rewrite `fact2` to ML code that has been transformed by this optimisation. You will probably use references and assignments as well as the construct `while EXP do EXP`. [8 marks]

(*c*) Suppose that the programmer used instead a function as an accumulator.

```
fun fact3 n =
let fun aux (m, h) =
        if m <= 1
        then h(1)
        else aux(m-1, fn r => m * (h r))
in aux (n, fn x => x) end
```

Will your optimisation still work in this case? Explain your answer in detail. [8 marks]

3 Compiler Construction (TGG)

Consider the following context-free grammar of expressions

$$E ::= n \mid (E,\ E)$$

where n ranges over integers.

(a) Present a right-most derivation of the expression ((21, 18), 17). [2 marks]

(b) List the LR(0) items for this grammar. [2 marks]

(c) Describe the states of the deterministic finite automata associated with an LR(0) parser for the grammar presented above. Explain your method of constructing these states. [4 marks]

(d) Describe the calculation of the *goto* function associated with an LR(0) parser for the grammar above. How is the goto function used by the parser? [4 marks]

(e) Carefully describe the LR(0) parsing associated with your derivation in (a). That is, show each transition of the parser and how it performs *shift* and *reduce* operations. [8 marks]

4 Compiler Construction (TGG)

(*a*) Consider a programming language with nested function declarations that allows only *first-order* functions. That is, functions are not treated as values and can neither be passed as arguments nor returned by functions.

Lambda lifting and *static links* are two common methods of implementing such a language using a run-time stack. Describe these methods and discuss their advantages and disadvantages. [6 marks]

(*b*) Now suppose we are dealing with a programming language that supports higher-order functions that can be passed as arguments and returned as results. Give an example, in ML-like pseudo-code, where the techniques that you have described in (*a*) can no longer be used. Justify your answer. [6 marks]

(*c*) Carefully explain the techniques you might use to compile the example that you presented in (*b*). [8 marks]

3 Compiler Construction (TGG)

Programming answers should be written in some notation approximating SML or OCaml.

(*a*) Describe what is meant by *tail recursion*. [4 marks]

(*b*) Eliminate tail recursion from `foldl` given below. Explain your answer.

```
(*
    foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
*)
let rec foldl f accu l =
  match l with
      [] -> accu
  | a::l -> foldl f (f accu a) l
```

[8 marks]

(*c*) Eliminate tail recursion from the following mutually tail-recursive functions. Explain your answer.

```
let rec is_even n =
    if n = 0
    then true
    else is_odd (n - 1)

and is_odd n =
    if n = 0
    then false
    else is_even(n - 1)
```

[8 marks]

4 Compiler Construction (TGG)

Consider writing a compiler for a simple language of expressions given by the following grammar,

$$\begin{array}{rcll} e & ::= & n & \text{(integer)} \\ & \mid & ? & \text{(read integer input from user)} \\ & \mid & e + e & \text{(addition)} \\ & \mid & e - e & \text{(subtraction)} \\ & \mid & e * e & \text{(multiplication)} \\ & \mid & (e, e) & \text{(pair)} \\ & \mid & \texttt{fst}\ e & \text{(first projection)} \\ & \mid & \texttt{snd}\ e & \text{(second projection)} \end{array}$$

(*a*) Describe the tasks that should be carried in implementing a front end for this language and any difficulties that might be encountered. [5 marks]

(*b*) Suppose that the target virtual machine is stack-oriented and that the stack elements are integer values, and addresses can be stored as integers. Explain which other features are required in such a virtual machine. Invent a simple language of instructions for such a machine and show how it would be used to implement each of the expressions. [10 marks]

(*c*) Suppose that the following rules are proposed as possible optimizations to be implemented in your compiler.

expression	simplifies to	expression
$(\texttt{fst}\ e,\ \texttt{snd}\ e)$	$\rightarrow$	e
$\texttt{fst}\ (e_1,\ e_2)$	$\rightarrow$	e_1
$\texttt{snd}\ (e_1,\ e_2)$	$\rightarrow$	e_2

Describe how you could implement these rules so that the simplifications are made *only* when the program's semantics is correctly preserved. [5 marks]

3 Compiler Construction (TGG)

(*a*) Explain why some programming languages require automatic memory management ("garbage collection") for program execution. [4 marks]

(*b*) At a given point in the execution of a program, what can be considered as garbage? How can garbage be located in memory? [4 marks]

(*c*) Suppose a programmer is implementing garbage collection using reference counting. Discuss whether or not they need to consider the possibility of a reference count overflowing when incremented. [4 marks]

(*d*) Suppose we are writing a compiler for an ML-like language. We want to employ the equation

```
(map f) o (map g) = map (f o g)
```

as a left-to-right rewrite rule for optimisation. The symbol `o` represents function composition — for any value `v` the expression `(f o g) v` evaluates to the value of `f(g v)`.

Discuss the merits of this idea. Is it always correct? [8 marks]

4 Compiler Construction (TGG)

Consider the following simple evaluator for a language of expressions written in OCaml.

```
type expr =
   | Integer of int            (* integer                *)
   | Pair of expr * expr       (* pair                   *)
   | Apply of string * expr    (* apply a named function *)

type value =
     | INT of int
     | PAIR of value * value

(* eval : expr -> value *)
let rec eval = function
   | Integer n      -> INT n
   | Pair (e1, e2)  -> PAIR (eval e1, eval e2)
   | Apply (f, e)   -> eval_function(f, eval e)
```

In this code the function `eval_function` has type `string * value -> value` and is used to evaluate some "built in" functions. For example,

```
eval_function("add", PAIR(INT 10, INT 7))
```

could return the value `INT 17`.

(*a*) Rewrite the `eval` function in continuation passing style (CPS) to produce a function `eval_cps` so that the function

```
let eval_2 e = eval_cps (fun x -> x) e
```

will produce the same results as the function `eval`. [10 marks]

(*b*) Eliminate higher-order continuations from your `eval_cps` function. That is, introduce a data type `cnt` to represent continuations and write functions of type

```
eval_cps_dfn : cnt -> expr -> value
apply_cnt    : cnt * value -> value
eval_3       : expr -> value
```

using the technique of defunctionalisation. Note that functions `eval_cps_dfn` and `apply_cnt` will be mutually recursive. [10 marks]