This is a collection of past 2010 exam papers. The list of exercises presented here DOES NOT cover the whole range of topics taught. Please DO NOT make the WRONG assumption that only the following topics will be asked in the exam!

Exercise

1. Consider this fragment of Java code:

```
public int max(int i, int j) {
      if (i<j)
            return j;
      else
            return i;
}
```

   a. List the keywords, special characters and identifiers that a lexical analyser should recognise for the above code.

   [3 marks]

   b. Describe the concepts of token, token type and token value providing examples from the code above.

   [3 marks]

2. Given the regular expression:

```
(00|11)*
```

   a. Describe (in plain English) the language denoted by the regular expression, and provide examples of strings belonging to the language.

   [2 marks]

   b. Construct the non-deterministic finite-state automata (NFA) for the above regular expression, and then build the deterministic finite state automata (DFA) from the NFA.

   [8 marks]

   c. Show the sequence of moves made by the automata in processing the input strings `001111` and `001`. Comment on the results.

   [4 marks]

[TOTAL: 20 marks]

Exercise

1. Consider the following context-free grammar G for boolean expressions:

   B → B `&&` B
   B → B `||` B
   B → `(` B `)`
   B → `id`

   where `&& || ( )` and `id` are tokens, while B is a non-terminal.

   a. Demonstrate that the above grammar is ambiguous, by providing two different left-most derivations for the expression:

   ```
   id || id && id && id
   ```

   Discuss why ambiguity is a problem, by showing the abstract syntax trees (ASTs) associated with the two derivations previously built.

   [8 marks]

   b. Fully parenthesise the expression based on the meaning of each of the derivations previously provided.

   [2 marks]

   c. Illustrate how ambiguity problems can be resolved in CUP using precedence and associativity rules.

   [3 marks]

2. Consider the following context-free grammar G1, being an unambiguous version of G:

   B → B `||` A
   B → A
   A → A `&&` R
   A → R
   R → `(` B `)`
   R → `id`

   a. Demonstrate that there exists only one left-most derivation in G1 for the expression: `id || id && id && id`

   [4 marks]

   b. Extend G1 to an unambiguous grammar G2 that includes the relational operators: `==` `!=` `<` `>` `<=` `>=`. Assume these operators all have the same priority, which is higher than both `&&` and `||`.

   [8 marks]

[TOTAL: 25 marks]

Exercise

Consider the following context-free grammar G for C-style variable assignment:

E → L = R
E → R
L → * R
L → id
R → L

where = * and id are tokens, while E, L and R are non-terminals.

a. Demonstrate that G is not LR(0).

[8 marks]

b. Discuss the term shift-reduce conflict. Explain when and where the conflict arises in the given grammar. Provide an example of input that would incur into such a parsing conflict. Manually modify the table to resolve the conflict so that your input can be successfully parsed.

[6 marks]

c. Explain why LR(0) grammars often give rise to shift/reduce conflicts. Describe the advantages and drawbacks of LR(1) grammars compared to LR(0). In particular, discuss why LR(1) grammars avoid parsing action conflicts in most cases and why, although being so powerful, they are hardly ever used in practise.

[6 marks]

[TOTAL: 20 marks]

Exercise

Consider the fragment of Java code:

```
i=10;
while (i < 0) {
      j = j + (j - i) * l;
      i = (j * 5) / h - 10;
}
```

a. Translate the above Java code into three-address code.

[5 marks]

b. Assume that (part of) the context-free grammar that generates the above code is the following:

stmt_list → stmt_list stmt
| stmt
stmt → while_stmt
| assign_stmt
assign_stmt → `ID BECOMES` expr `SEMI`
while_stmt → `WHILE LPAREN` expr `RPAREN` block
block → `LBRACE` stmt_list `RBRACE`
expr → expr `PLUS` expr
| expr `TIMES` expr
| `ID`
| `NUMBER`

Write a CUP-like specification to implement the syntax-directed definition for translating the above grammar for statements and expressions to three-address code. State clearly your assumptions about the type of the attribute associated with non-terminals where your intermediate code is stored, and about auxiliary functions you may use.

[10 marks]

c. Consider extending the above grammar to include the `for' statement:

for_stmt ::= FOR (expr; expr; expr) block_stmt

Show how the for statement can be translated to three-address code, making use of the basic three-address code operators. You may assume that the three expressions that form the for-condition must appear.

[10 marks]

[TOTAL: 25 marks]

Exercise

Consider the following context-free grammar G:

S → A`b`
S → B`c`
A → `a`A
A → ε
B → `a`B
B → ε

where `a` `b` and `c` are tokens, while S, A and B are non-terminals (S is the start symbol of the grammar).

d. Demonstrate that G is not LR(0).

[8 marks]

e. Demonstrate that G is LR(1)

[12 marks]

[TOTAL: 20 marks]

Exercise

Consider the fragment of Java code:

```
if (i != 0) {
      j = (j + k) / i * l;

} else {
      j = (j + k) / (i + 1) * l;
}
```

a. Translate the above Java code into three-address code.

[5 marks]

b. Assume that (part of) the context-free grammar that generates the above code is the following:

stmt_list → stmt_list stmt
| stmt
stmt → if_stmt
| assign_stmt
assign_stmt → `ID BECOMES` expr `SEMI`
if_stmt → `IF LPAREN` expr `RPAREN` block `ELSE` block
block → `LBRACE` stmt_list `RBRACE`
expr → expr `DIVIDE` expr
| expr `TIMES` expr
| `ID`
| `NUMBER`

Write a CUP-like specification to implement the syntax-directed definition for translating the above grammar for statements and expressions to three-address code. State clearly your assumptions about the type of the attribute associated with non-terminals where your intermediate code is stored, and about auxiliary functions you may use.

[10 marks]

c. Consider extending the above grammar to include the `factorial' operator:

```
a! = a * (a-1) * (a-2) * … * 1
0!=1
```

Show how the factorial operartor can be translated to three-address code, making use of the basic three-address code operators. You may assume that `a` is a non-negative integer.

[10 marks]

[TOTAL: 25 marks]

Exercise

1. Given the regular expression:

   `1*(00|11)0*`

   d. Describe (in plain English) the language denoted by the regular expression, and provide examples of strings belonging to the language.
   [2 marks]

   e. Construct the non-deterministic finite-state automata (NFA) for the above regular expression, and then build the deterministic finite state automata (DFA) from the NFA.
   [8 marks]

   f. Show the sequence of moves made by the automata in processing the input strings `100` and `010`. Comment on the results.
   [4 marks]

2. Regular expressions are used to specify the lexical tokens of a programming language. However, regular expressions alone may be ambiguous. Illustrate two different cases of ambiguity, and discuss how lexical analysers resolve them.
   [6 marks]

[TOTAL: 20 marks]

Exercise

Both regular expressions and context-free grammars can be used to define languages.

1. Define precisely what a regular expression is. Provide examples of regular expressions and of the languages they define.

   [5 points]

2. Define precisely what a context-free grammar is. Provide examples of context-free grammars and of the languages they define.

   [5 points]

3. Demonstrate that context-free grammars are more expressive than regular expressions, by showing that any language that can be defined by a regular expression can also be defined by a context-free grammar. Hint: demonstrate by induction on the structure of a regular expression.

   [10 points]

4. Provide an example of a language that can be defined using a context-free grammar but not by a regular expression.

   [5 points]

[TOTAL: 25 points]

Exercise

Consider the following context-free grammar G that generates regular expressions:

S → S | S
S → S . S
S → S*
S → (S)
S → a
S → b

where a b ( ) * . | are tokens, while S is a non-terminal.

1. Provide two examples of token streams that can be derived using the above grammar.

   [2 marks]

2. Demonstrate that the above grammar is ambiguous, by providing two different left-most derivations for the same stream of tokens. Draw the abstract syntax trees corresponding to the two derivations and discuss their different interpretations.

   [8 marks]

3. Disambiguate the above grammar; remember that the star * operator has higher precedence than the concatenation . operator and that the concatenation . operator has higher precedence than the alternation | operator. Also remember that all these operators associate to the left.

   [10 marks]

[TOTAL: 20 marks]

Exercise

The third phase of a compiler is semantic analysis.

1. Discuss concisely the two main tasks of semantic analysis.

[6 marks]

2. Consider the following fragment of java code:

```
1:  public class Rectangle {
2:    int width;
3:    int height;
4:
5:    public int perimeter () {
7:      int p;
8:      if ((width > 0) && (height >0)) {
9:          int tmp = width + height;
10:      } else {
11:         int tmp=0;
12:      }
13:      p = tmp * 2;
14:      return p;
15:    }
16: }
```

   a. Define what a lexical scope is, and provide 2 different examples taken from the code above.

[4 marks]

   b. Scope semantic checks are performed by means of the information contained in a symbol table. Illustrate the content of the symbol table for the fragment of code above.

[6 marks]

   c. Discuss why semantic analysis returns a scope error at line 13: what scope rule has been violated, and how does the symbol table help in detecting the error?

[4 marks]

3. Assume that lexical and syntax analysis have been performed on your input program, and that an abstract syntax tree has been created. Describe how scope semantic checks can be implemented, by means of a visit to the tree.

[5 marks]

[TOTAL: 25 marks]

Exercise

a. **Bottom-up Parsing.** Consider the following context-free grammar G:

S→aAd

S→aec

S→bAc

S→bed

A→e

where S and A are non-terminals, S is the start symbol of the grammar, and `a b c d e` are terminals.

- Demonstrate that the grammar G is not SLR.
- Based on the result above, and without further computations, discuss whether each of the following statements is true or false. Explain why.

"The grammar G is LR(0)"

"The grammar G is LR(1)"

[20 marks]

b. **Derivations.** Given a context-free grammar G and a list of token s, you can demonstrate that s is syntactically correct by building a derivation in G for s.

- Define what a derivation is.
- Discuss the difference between a left-most and a right-most derivation.
- Discuss how left-most derivations can be used to demonstrate that a grammar is ambiguous.

[10 marks]

[TOTAL: 30 marks]

Exercise

a. **Syntax Analysis.** There exist two basic techniques to perform syntax analysis: top-down and bottom-up. Illustrate similarities and differences between the two techniques focusing on:

- The type of derivation they build, and the order in which such a derivation is constructed;
- The classes of grammars they parse. For each class, indicate what kind of manipulations a context-free grammar must undergo to fall into these classes and why;
- The behaviour of the parser. More specifically, indicate what kind of actions the parser may perform; illustrate briefly how the parsing table is consulted to decide the next action; summarize in a few words how the parsing table is built.

[15 marks]

b. **Semantic Analysis.** One of the main goals of semantic analysis is scoping.

- Spell out the two basic rules for scope semantic checks;
- Discuss how scope rules can be checked by means of a hierarchy of symbol tables. In particular, discuss how the hierarchy of symbol tables solves the problem of name collision; also, explain how the hierarchy is traversed to ensure scope rules are respected and to catch potential scope errors. Refer to the fragment of code shown below to illustrate your argument;

```
public void foo() {
  int x = 5;
  for (int i=0; i<10; i++) {
    int x = i;
    int y = x/5;
   }
  for (int i=0; i<25; i++) {
    x = x * y;
  }
}
```

- The construction of the symbol table and scope semantic checks can be performed in parallel, unless the language allows forward references. Illustrate with an example why it is so.

[15 marks]

[TOTAL: 30 marks]

Exercise

Run-time organisation. When a routine, or an object method, is invoked, a new environment called an activation record (or frame) is created.

a. Describe in detail the typical content of an activation record.

[15 marks]

b. Illustrate the allocation scheme of activation records on an abstract machine (assume that the language does not support concurrency, so that only one activation record represents a running routine). Discuss the role of the Frame Pointer (FP) and of the Stack Pointer (SP) in this allocation scheme.

[10 marks]

c. Given the routine:

```
void A(int a) {
    void B() {
        print(a);
    }
    void C (int c) {
        if (c==0) B(); else C(c-1);
    }
    C(3);
}
```

- Draw the stack of activation records that results from calling A(2).
- For each of the activation records depicted at the previous step, illustrate where the dynamic link pointer and the static link pointer point.

[15 marks]

[TOTAL: 40 marks]

Exercise

AST and Intermediate Code Generation. The last phase of the front-end of a compiler is intermediate code generation.

a. Discuss in detail the advantages of generating intermediate code instead of immediately creating target code.

[10 marks]

b. Translate the following fragment of Java code to 3-address code.

```
a=10;
while(a>0) {
      b[a-1]=a*10-7/a;
      a=a-1;
}
```

[10 marks]

c. Assume that (part of) the grammar that generates the above code is the following:

stmt_list → stmt stmt_list
| ε
stmt → while_stmt
| assign_stmt
assign_stmt → ID BECOMES expr SEMI
| ID LSQUARE expr RSQUARE BECOMES expr SEMI
while_stmt → WHILE LPAREN expr RPAREN block
block → LBRACE stmt_list RBRACE
expr → expr PLUS expr
| expr TIMES expr
| expr MINUS expr
| expr DIVIDE expr
| expr GREATER expr
| ID
| NUMBER

Illustrate with a detailed UML class diagram the data structures (and their relationship) you would need to build the abstract syntax tree (AST) for programs written according to the above grammar. For each class, illustrate the instance variables you need.

[20 marks]

[TOTAL: 40 marks]

Exercise

a. **Semantic Analysis.** Consider the following fragment of Java code:

```
public class A {
    int i;
    double d;
    public double sum(int i, double d) {return i+d;}
    public double sum() {return i+d;}
}
```

- Provide a type expression for class `A`.
- Illustrate in detail the content of the symbol table for the fragment of code listed above. If more than one table is used, illustrate the parent-child relationship among these tables.

[15 marks]

b. **Type Checking.** Consider the following fragment of pseudo code:

```
public class MySum {
    public int sum (int a, int b) {return a+b;}
    public double sum (double d, double f) {return d+f;}
}
…
MySum m=new MySum();
double d1=m.sum(10, 20);          (1)
double d2=m.sum(5.6, 10+7.8);     (2)
int i=m.sum(5+6,7);               (3)
```

List the type rules that a plausible type checker would check when dealing with statements (1), (2) and (3). For each statement, state also whether the type checker would fire a type checking error and why.

[15 marks]

[TOTAL: 30 marks]

Exercise

a. **Parameters Passing.** When a routine, or an object method, is invoked, parameters can be passed in one of 4 possible ways: call-by-value, call-by-reference, call-by-name, copy-restore.

- Describe in detail how each of these techniques work. Illustrate what information is placed in the parameters space of the activation record upon routine invocation.
- For each technique, illustrate what `print(a)` and `print(b)` would print after calling `swap(a,b)` in the pseudo code below.

```
void swap(int i,int j) {
   int tmp=i;
   i=j;
   j=tmp;
}
…
int a=10;
int b=20;
swap(a,b);
print(a);
print(b);
```

[20 marks]

b. **Activation Records.** Given the routine:

```
void A(int a) {
    void B() {
        print(a);
    }
    void C (int c) {
          void D (int d) {
              if (d==0) B(); else C(d-1);
          }
          D(c);
    }
    C(3);
}
```

- Draw the stack of activation records that results from calling A(5).
- For each of the activation records depicted at the previous step, illustrate where the dynamic link pointer and the static link pointer point.
- Briefly describe static and dynamic link pointers. Discuss why it is important to maintain these pointers (what are they used for at run-time?).

[20 marks]

[TOTAL: 40 marks]

Exercise

a. **Intermediate Code Generation.** The last phase of the front-end of a compiler is intermediate code generation. Translate the following fragment of Java code to 3-address code.

```
if(a>b) {
      if (b!=10) {
            c[a-b]=a+5*b-10;
      } else {
            c[10]=a*b;
      }
}
```

[10 marks]

b. **AST.** Consider the following grammar for variable declarations:

<VarDecList> → <SingleVarDec> <VarDecList>
| ε
<SingleVarDec> → <BasicTypeDecl>
| <ArrayTypeDecl>
<BasicTypeDecl> <NameType> `ID ;`
| <NameType> `ID =` <Expr> `;`
<ArrayTypeDecl> → <NameType> `[ INTNUM ] ID ;`
| <NameType> `[ INTNUM ] ID = [` <Expr> <ExprList> `];`
<NameType> → `int` | `float` | `bool`
<ExprList> → `,` <Expr> <ExprList>
|ε
<Expr> → `ID` | `INTNUM`

- Illustrate with a detailed UML class diagram the data structures (and their relationships) you would need to build the abstract syntax tree (AST) for variable declarations written according to the above grammar. For each class, illustrate the instance variables you need.

[20 marks]

- According to the data structures defined above, illustrate the AST for the following declaration:
  int[3] intArray = [10,20,30];

[10 marks]

[TOTAL: 40 marks]

Exercise

1. Translate the following fragment of code to 3-address code:

```
while (a*b < 100 && a/b>=c || d)
   a = a-1/2*3;
```

[10 marks]

2. Provide a 3-address code translation scheme for the following statements and operators:
   a. Repeat-until

```
repeat {
  <body>
} until ( <expr> )
```

   whose meaning is: repeat <body> as long as <expr> is false. Stop when <expr> evaluates to true.

   b. Exponentiation operator

```
a^b = a * a * ... * a (b times)
```

   You may assume that a,b are positive integers

[10 marks]

3. There exist three different ways to implement three-address statements. Name them, illustrate their characteristics, and discuss their strengths and limitations.

[10 marks]
[TOTAL: 30 marks]

Exercise

Given the following fragment of source code:

```
if (b > 0) {
        exp = 1;
        while (b > 0) {
                exp = exp * a;
                b = b - 1;
        }
}
```

1. Translate it to 3-address code. Partition the generated list of 3-address statements into basic blocks. Represent this sequence of blocks with a directed flow graph.

   [15 marks]

2. Given the assignment statement:

   a = (b+c) * (b+-c) * (-b*c)

   a. Translate it to 3-address code and compute next-use information of all variables and temporaries.
   b. Based on the computed next-use information, generate intermediate code that optimises the usage of temporaries.

   [15 marks]

3. Describe briefly how Peephole optimisation works. Discuss 4 different program transformations that can be performed as part of this optimisation technique; provide examples for each of these transformations.

   [10 marks]

[TOTAL: 40 marls]