

EXAMINATIONS – 2016

TRIMESTER 1

<p>SWEN 430</p> <p>COMPILER ENGINEERING</p>

Time Allowed: THREE HOURS

CLOSED BOOK

Permitted materials: No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.

Instructions: Answer all questions

You may answer the questions in any order. Make sure you clearly identify the question you are answering.

Question	Topic	Marks
1.	Grammars and Parsing	30
2.	Typing & Static Analysis	30
3.	Java Bytecode	30
4.	Machine Code	30
5.	Register Allocation	30
6.	Advanced Topics	30
Total		180

1. Grammars and Parsing

(30 marks)

A recursive descent parser is implemented as a set of recursive methods, one for each nonterminal in an LL(1) grammar. The method for nonterminal N attempts to parse an instance of N as a prefix of the input, based on the rule defining N .

- (a) **(4 marks)** Explain briefly why the LL(1) conditions are important to ensuring that a recursive descent parser works correctly.

- (b) Consider the following grammar, where nonterminals are in italics and all other symbols are terminals, and nonterminals other than *IfStmt* are assumed to be defined elsewhere.

$$\begin{aligned} \textit{IfStmt} & ::= \textit{if} (\textit{Exp}) \textit{Stmt} \\ & \quad | \textit{if} (\textit{Exp}) \textit{Stmt} \textit{else} \textit{Stmt} \end{aligned}$$

- i. **(6 marks)** Explain why this grammar violates the first LL(1) condition (also known as the Choice Condition), and why a parser based directly on this grammar would not work correctly.

- ii. **(4 marks)** Show how the grammar can be modified so that it satisfies the first LL(1) condition, and explain how this corrects the problem with the parser.

- (c) Consider the following grammar, where again nonterminals are in italics and all other symbols are terminals, and nonterminals other than *Exp* are assumed to be defined elsewhere.

$$\textit{Exp} ::= \textit{Exp op Term} \mid \textit{Term}$$

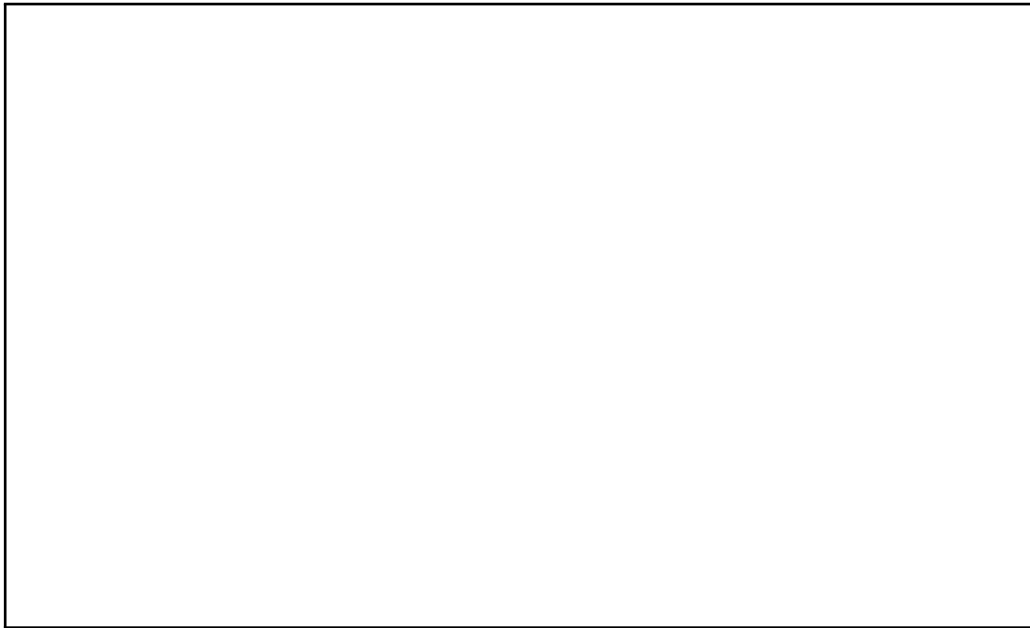
- i. **(6 marks)** Explain why this grammar violates the first LL(1) condition, and why a parser based directly on this grammar would not work correctly.

- ii. **(5 marks)** One way to avoid the above problem is to rewrite the grammar as:

$$\begin{aligned} \textit{Exp} & ::= \textit{Term op Exp} \mid \textit{Term} \\ \textit{op} & ::= + \mid - \end{aligned}$$

Explain why this solution is often undesirable when a parser builds a parse tree based on the grammar, which is then used for generating code.

- iii. **(5 marks)** Explain how an alternative solution can be obtained by basing the parser on an Extended BNF grammar, and how this avoids the problem encountered with the solution described in part **(ii)**.



Student ID:

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

2. Typing & Static Analysis

(30 marks)

- (a) Consider a simplified version of the λ_W language, where a program is a sequence of function definitions followed by an expression to be evaluated, and the body of a function is just an expression (i.e. there are no statements). The syntax for this language, and most of its type system are shown below.

p	$::=$	$f_1 \dots f_n e$	<i>programs</i>
f	$::=$	$T_1 n_1 (T_2 n_2) \{ e \}$	<i>functions</i>
e	$::=$		<i>expressions</i>
		b	<i>logical constants</i>
		c	<i>numeric constants</i>
		n	<i>variables</i>
		$e_1 \text{ op } e_2$	<i>binary</i>
		$n(e)$	<i>application</i>
b	$::=$	$\text{true} \mid \text{false}$	<i>truth values</i>
c	$::=$	$\dots \mid -1 \mid 0 \mid 1 \mid \dots$	<i>numeric values</i>
T	$::=$	$\text{bool} \mid \text{int}$	<i>types</i>
op	$::=$	$'=='$ $'!=='$ $'+'$ $'-'$ $'*'$	<i>operators</i>
		$'<'$ $'<='$ $'>'$ $'>='$	

$\frac{}{\Gamma \vdash b : \text{bool}}$	(T-Bool)	$\frac{}{\Gamma \vdash c : \text{int}}$	(T-Num)	$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$	(T-Var)
$\frac{\Gamma \vdash e_1 : \text{int}, \Gamma \vdash e_2 : \text{int}, \text{op} \in \{'+', '- ', '*'\}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{int}}$ (T-AOp)					
$\frac{\Gamma \vdash e_1 : \text{int}, \Gamma \vdash e_2 : \text{int}, \text{op} \in \{ '<', '<=', '>', '>=' \}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{bool}}$ (T-Rel)					
$\frac{\Gamma \vdash e_1 : T, \Gamma \vdash e_2 : T, \text{op} \in \{ '==', '!==' \}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{bool}}$ (T-Eq)					
$\frac{\Gamma \cup \{n_2 : T_2\} \vdash e : T_1}{\Gamma \vdash T_1 n_1 (T_2 n_2) \{e\} \text{ OK}}$ (T-Fun)					
$\frac{\Gamma \vdash \bar{f} \text{ OK}, \Gamma \cup \{\bar{f}\} \vdash e : T}{\Gamma \vdash \bar{f} e : T}$ (T-Prog)					

- i. (6 marks) What is missing from the type system is a rule for typing function calls. Complete the following rule for typing function calls, and explain the meaning of the premises you add.

$\frac{\quad}{\Gamma \vdash n(e) : T}$
--

- ii. (4 marks) Suppose the language is extended to include a notion of subtyping, for example we might treat Booleans as a subtype of integers. In this case, we want to modify the above rule for typing function calls so that subtypes or supertypes can be used where appropriate. Briefly explain how the above rule would be modified to support subtyping.

--

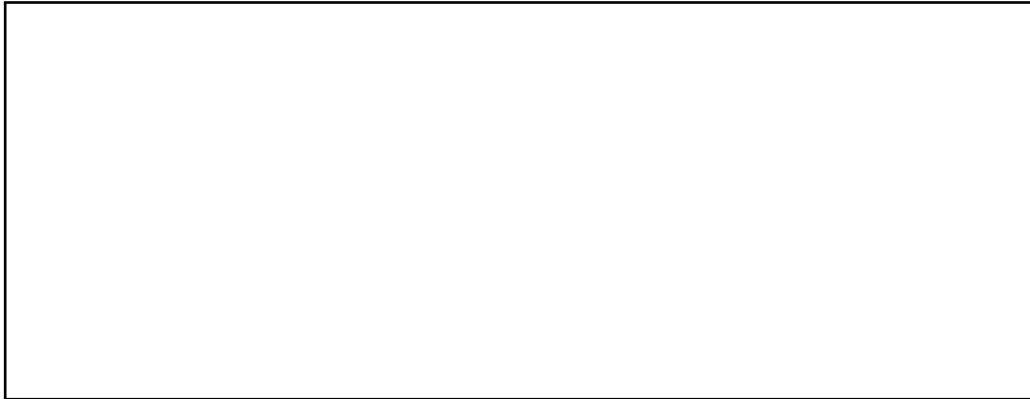
(b) This part is concerned with the process of detecting *dead code*.

- i. **(4 marks)** Explain, using an example, what is meant by *dead code*, and why it is helpful for a compiler to detect dead code.

- ii. **(6 marks)** Explain, using an example, what is meant by a *Control Flow Graph*, and how a simple depth-first traversal algorithm can be used to detect dead code.

- iii. **(5 marks)** Explain briefly, using an example, why no algorithm can be guaranteed to detect all cases of dead code.

- iv. **(5 marks)** Explain briefly, using an example, how the algorithm you described in part **(iii)** can be extended to detect uninitialised variables.



3. Java Bytecode

(30 marks)

(a) Consider the following method written in Java bytecode:

```
public List create(java.lang.String);
  new class java/util/ArrayList
  dup
  invokespecial java/util/ArrayList."<init>":()V
  astore 2
  aload 2
  aload 1
  invokeinterface java/util/List.add:(Ljava/lang/Object;)Z
  pop
  aload 2
  areturn
```

i. (5 marks) In the box below, give Java source code equivalent to the bytecode above.

ii. (2 marks) Briefly, outline one way the above bytecode could be rewritten to an equivalent with fewer bytecodes.

iii. (3 marks) What is the *maximum stack height* of the above method? Be sure to show your working.

(b) Consider the following method implemented in Java:

```
int sum(int[] xs) {  
    int r = 0;  
    int i = 0;  
    while(i < xs.length) {  
        r = r + xs[i];  
        i = i + 1;  
    }  
    return r;  
}
```

i. (6 marks) In the box below, translate the above method into Java bytecode.

- (c) The following outlines a method for translating *integer* expressions represented in a simple Abstract Syntax Tree (AST) to Java Bytecode:

```
void translate(Expr expr, Map<String,Integer> env) { ... }
```

For simplicity, the method prints its translation to `System.out` in format similar to that found on Page 10. The `env` argument identifies the allocated JVM register of each variable. All variables correspond to the JVM type `int`.

In the boxes below, complete the methods for translating pieces of our AST into Java bytecode.

- i. (2 marks)

```
interface Variable extends Expr {
    public String getName();
}
```

```
void translate(Expr.Variable e, Map<String,Integer> env) {
```

- ii. (4 marks)

```
enum BinOp { ADD, SUB, MUL, DIV }
```

```
interface BinaryOp extends Expr {
    public BinOp getOperator();
    public Expr getLhs();
    public Expr getRhs();
}
```

```
void translate(Expr.BinaryOp e, Map<String,Integer> env) {
```

We now consider the related method for translating statements into Java Bytecode:

```
void translate(Stmt stmt, Map<String,Integer> env) { ... }
```

In the boxes below, complete the methods for translating statements into Java bytecode. You may use labels to identify branch destinations as follows:

```
...
goto label1
...
label1:
...
```

iii. (3 marks)

```
interface Assign extends Stmt {
    public Expr.Variable getLhs();
    public Expr getRhs();
}
```

```
void translate(Stmt.Assign s, Map<String,Integer> env) {
```

iv. (5 marks)

```
interface If extends Stmt {
    public Expr getCondition();
    public List<Stmt> getTrueBranch();
    public List<Stmt> getFalseBranch();
}
```

```
void translate(Stmt.If s, Map<String,Integer> env) {
```

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

4. Machine Code

(30 marks)

Consider the following program written in WHILE:

```
int multiply(int x, int y) {  
    int i = 0;  
    int r = 0;  
    while(i < y) {  
        r = r + x;  
        i = i + 1;  
    }  
    return r;  
}
```

- (a) **(10 marks)** In the box below, translate the above program into X86_64 machine code. You should assume: **(1)** parameters *x* and *y* are passed in the `%rdi` and `%rsi` registers respectively; **(2)** the return value is passed in the `%rax` register; **(3)** all other registers are *callee-saved*.

NOTE: the Appendix on page 25 provides an overview of x86_64 machine instructions for reference.

- (b) **(5 marks)** During execution of a machine code program, a *stack frame* is created at runtime to hold critical information. Briefly, discuss the layout of a stack frame on x86_64. You may use your answer from (a) above to illustrate.

- (c) **(5 marks)** When generating machine code, a *calling convention* is needed to coordinate between *caller* and *callee*. Briefly, discuss what a calling convention is. You may use your answer from (a) above to illustrate.

- (d) **(5 marks)** The special *flags register* is an unusual feature of the `x86_64` architecture. Briefly, explain what this register does. You may use your answer from **(a)** above to illustrate.

- (e) **(5 marks)** The GNU C Compiler (GCC) supports the `-fomit-frame-pointer` command-line switch. When enabled, GCC will avoid storing the frame pointer in the `rbp` (or any other) register *when it is safe to do so*. As such, this optimisation can free up the `rbp` register for general use. Briefly, discuss the situations in which this optimisation can be applied.

5. Register Allocation

(30 marks)

(a) Register allocation is an important process within a modern compiler.

i. (5 marks) Briefly, discuss what *register allocation* is.

ii. (5 marks) Briefly, discuss why *register allocation* is important for the performance of compiled programs.

(b) An integral component of any register allocation mechanism is *live variables analysis*.

i. (5 marks) Briefly, discuss what a live variable is.

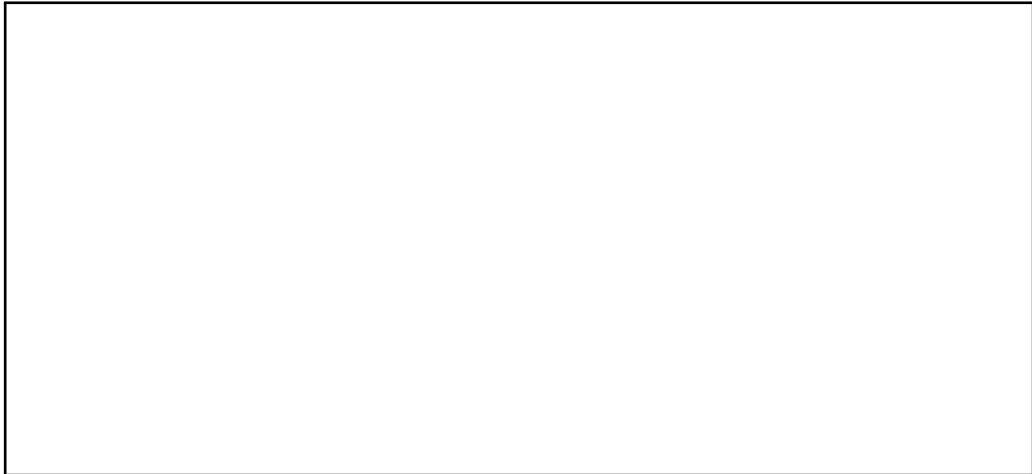
- ii. (5 marks) Briefly, discuss how live variable information helps with register allocation.

- iii. (3 marks) Draw the *interference graph* for the following method written in WHILE:

```
int[] trim(int[] ls, int n) {  
    int[] rs = [0;n];  
    int i = 0;  
  
    while(i < |rs|) {  
        rs[i] = ls[i];  
        i = i + 1;  
    }  
    return rs;  
}
```

- iv. (2 marks) Give a *minimal colouring* for your interference graph.

- v. (5 marks) Briefly, discuss how *graph colouring* helps with register allocation.



6. Advanced Topics

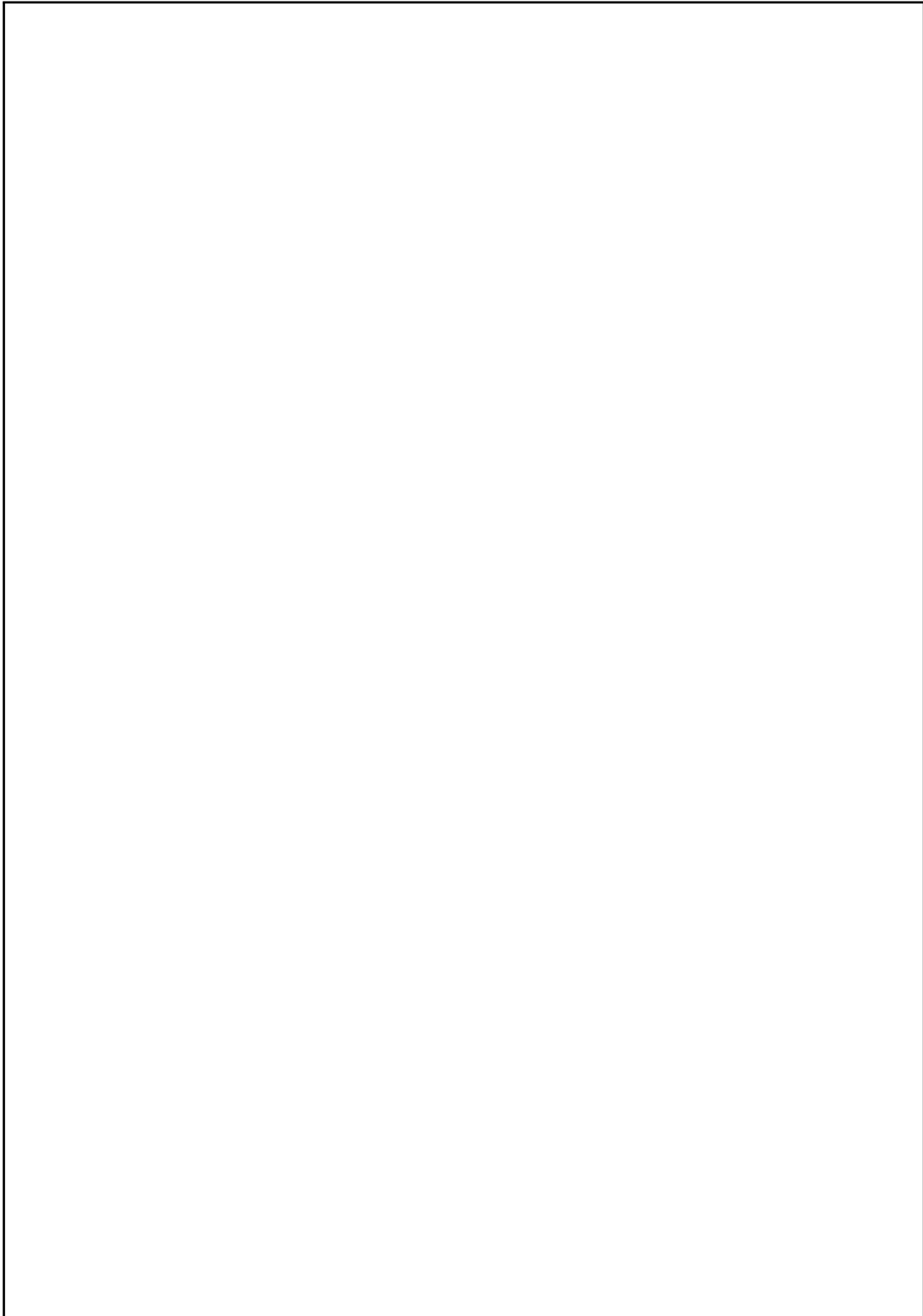
(30 marks)

Pick one of the research papers discussed and presented in the last two weeks of the course and answer the following questions.

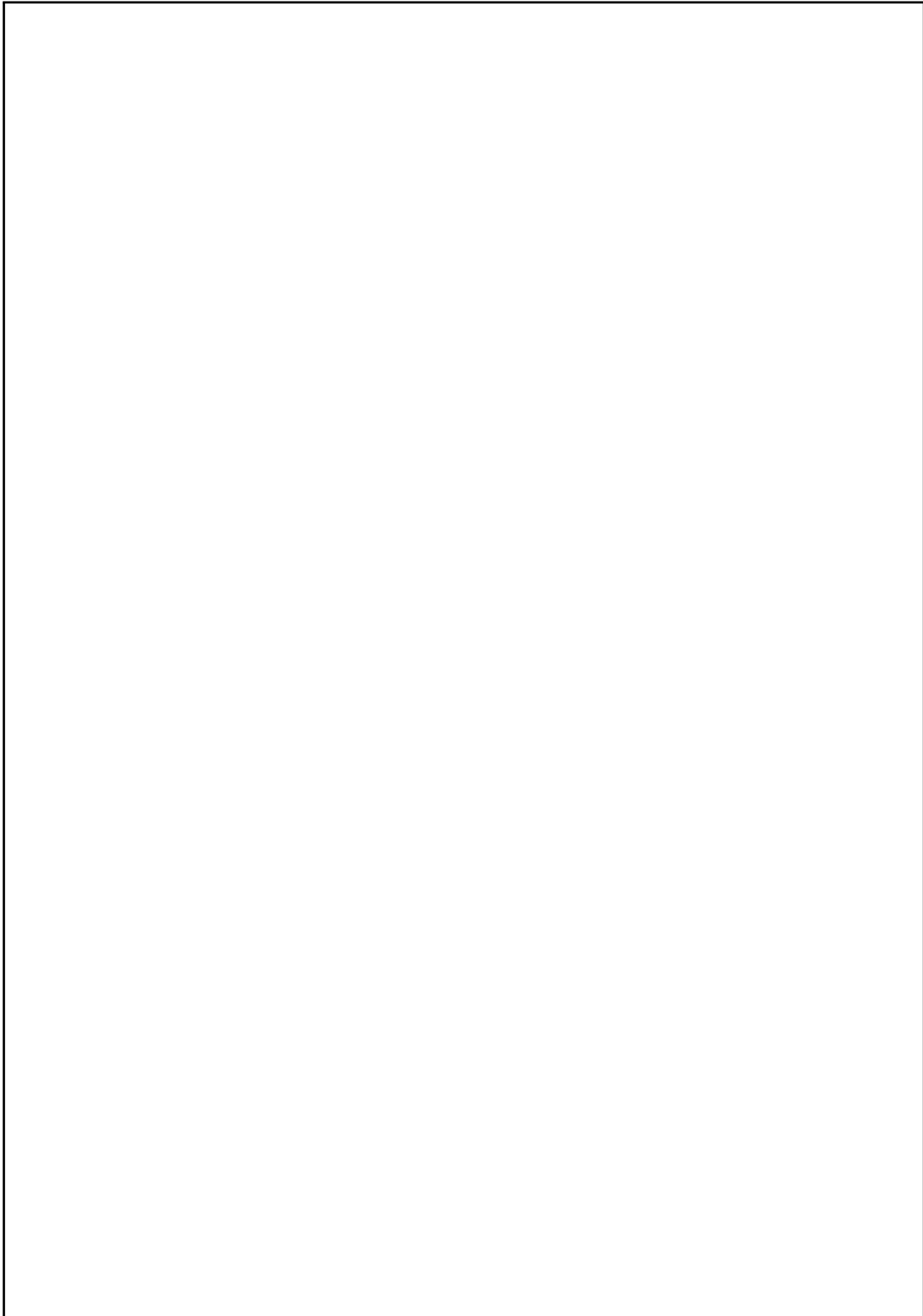
Note that this is a 30 mark (or 30 minute) question and thus each of these parts should take you around 10 minutes to answer - a one paragraph answer is NOT going to get many marks!

- (a) **(10 marks)** Describe in your own words what was the paper about and what compiler-related issue or technique it presented.

(b) **(10 marks)** Give an example of how you would use such a technique in real life compiler.

A large, empty rectangular box with a thin black border, intended for the student to write their answer to the question.

(c) **(10 marks)** Describe how the contribution of the paper improved on the state of the art.



Appendix: Overview of x86_64 Machine Instructions

<code>movq \$c, %rax</code>	Assign constant <code>c</code> to <code>rax</code> register
<code>movq %rax, %rdi</code>	Assign register <code>rax</code> to <code>rdi</code> register
<code>addq \$c, %rax</code>	Add constant <code>c</code> to <code>rax</code> register
<code>addq %rax, %rbx</code>	Add <code>rax</code> register to <code>rbx</code> register
<code>subq \$c, %rax</code>	Subtract constant <code>c</code> from <code>rax</code> register
<code>subq %rax, %rbx</code>	Subtract <code>rax</code> register from <code>rbx</code> register
<code>cmpq \$0, %rdx</code>	Compare constant <code>0</code> register against <code>rdx</code> register
<code>cmpq %rax, %rdx</code>	Compare <code>rax</code> register against <code>rdx</code> register
<code>movq %rax, (%rbx)</code>	Assign <code>rax</code> register to dword at address <code>rbx</code>
<code>movq (%rbx), %rax</code>	Assign <code>rax</code> register from dword at address <code>rbx</code>
<code>movq 4(%rsp), %rax</code>	Assign <code>rax</code> register from dword at address <code>rsp+4</code>
<code>movq %rdx, (%rsi, %rbx, 4)</code>	Assign <code>rdx</code> register to dword at address <code>rsi+4*rbx</code>
<code>pushq %rax</code>	Push <code>rax</code> register onto stack
<code>pushq %c</code>	Push constant <code>c</code> onto stack
<code>popq %rdi</code>	Pop qword off stack and assign to register <code>rdi</code>
<code>jz target</code>	Branch to <code>target</code> if zero flag set.
<code>jnz target</code>	Branch to <code>target</code> if zero flag not set.
<code>jl target</code>	Branch to <code>target</code> if less than (i.e. sign flag set).
<code>jle target</code>	Branch to <code>target</code> if less than or equal (i.e. sign or zero flags set).
<code>ret</code>	Return from function.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.