



TESTS – 2021

TRIMESTER 2

SWEN 430

COMPILER ENGINEERING

Time Allowed: TWO 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	20
2.	Types and Type Checking	20
3.	Static Analysis	20
4.	Java Bytecode	20
5.	Machine Code	20
6.	Memory Models	20
Total		120

1. Grammars and Parsing

(20 marks)

(a) Briefly, describe the following components of a compiler.

i. **(2 marks)** Lexer.
ii. **(2 marks)** Parser.
iii. **(2 marks)** Abstract Syntax Tree.

(b) Consider the following grammar:

$$\begin{aligned}
 E &\rightarrow N \mid (E , E) \\
 N &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
 \end{aligned}$$

i. **(4 marks)** For each of the following inputs, state whether it would be accepted or not by the grammar:

- ii. **(4 marks)** Provide suitable Java classes for an Abstract Syntax Tree representation of the grammar from page 2,

- iii. (6 marks) Complete the following class `Parser` which should implement a *recursive descent parser* for the grammar given on page 2:

```
public class Parser {  
    private final String input;  
    private int offset = 0;  
  
    public Parser(String input) { this.input = input; }  
  
}
```

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. Types and Type Checking

(20 marks)

Consider the following simple imperative language and its corresponding typing rules.

$s ::=$ (Statements) $\quad \text{Tx} = e$ (Declarations) $\quad x = e$ (Assignments) $\quad s ; s$ (Sequence) $\quad e$ (Expressions)	$\frac{}{\vdash c : \text{int}}$ (T-INT) $\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$ (T-VAR)
$e ::=$ (Expressions) $\quad c$ (Integers) $\quad x$ (Variables) $\quad *e$ (Dereference) $\quad \text{new } e$ (Allocation)	$\frac{\Gamma \vdash e : \&T}{\Gamma \vdash *e : T}$ (T-Deref) $\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{new } e : \&T}$ (T-New)
$T ::=$ (Types) $\quad \&T$ (Reference type) $\quad \text{int}$ (Int type) $\quad \text{void}$ (Void type)	$\frac{\Gamma \vdash x : T \quad \Gamma \vdash e : T}{\Gamma \vdash \text{Tx} = e : \text{void}}$ (T-Decl) $\frac{\Gamma \vdash x : T \quad \Gamma \vdash e : T}{\Gamma \vdash x = e : \text{void}}$ (T-Assign) $\frac{\Gamma \vdash s_1 : T_1 \quad \Gamma \vdash s_2 : T_2}{\Gamma \vdash s_1 ; s_2 : T_2}$ (T-Seq)

- (a) (5 marks) For each of the following typing judgements identify a suitable typing environment Γ , or explain why no such typing environment exists.

 $\Gamma \vdash x = 1 : \text{void}$
 $\Gamma \vdash \&\text{int } x = \text{new } y ; z = *y : \text{void}$
 $\Gamma \vdash x = \text{new } y ; *x : \text{int}$
 $\Gamma \vdash y = x ; \&\text{int } x = \text{new } 1 : \text{void}$
 $\Gamma \vdash x = y ; y = *x : \text{void}$

- (b) Suppose the language were extended with a statement “delete e” which deallocates memory as in WHILE. For example, “delete p” deallocates the memory referred to by p.

- i. **(4 marks)** Provide a suitable typing rule for this statement.

- ii. **(5 marks)** Executing “`&int p = new 1 ; ... ; delete p`” can result in a *stuck* program. Briefly, discuss what this means using an example to illustrate.

- iii. **(6 marks)** Introducing the `delete` statement means the simple *progress theorem* shown in lectures no longer holds for our language. Briefly, discuss what this means.

3. Static Analysis

(20 marks)

This question concerns the *uniqueness analysis* developed for WHILE which determines, at each point, whether or not a variable is *defined*. A variable is defined after it has been assigned a value, but may become *undefined* if its value is *consumed* (e.g. moved to another variable). For simplicity, assume all references `&T` are *unique references*. For example, `&int` is a reference to an `int` variable and, furthermore, must be the *only* reference to that variable. The following illustrates:

```

1  &int p = new 123;
2  &int q;
3  // p is defined, q is undefined
4  if x >= 0 {
5      q = p;
6      // p is undefined, q is defined
7  }
8  // p and q are undefined

```

- (a) **(5 marks)** Explain briefly, using an example, why any algorithm for uniqueness analysis must be *conservative* (i.e. imprecise) in some way.

- (b) **(5 marks)** Using examples to illustrate, explain briefly why a depth-first traversal algorithm is *insufficient* for implementing the uniqueness analysis.

(c) A variable `x` is *consumed* by a statement if it must be undefined *after* that statement to preserve uniqueness. The method `consume(s)` returns the set of variables consumed by evaluating statement `s`.

i. (6 marks) Sketch an implementation of `consume(s)` for statements `x = e`, `assert e`, `delete e`, expressions `x`, `x == y`, `new e` and types `int`, `&int`. You may assume `typeof(x)` returns the declared type of a variable `x`.

- ii. **(4 marks)** Using `consume(s)`, give appropriate *dataflow equations* for the uniqueness analysis.

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.

4. Java Bytecode

(20 marks)

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

```
boolean f(int[], int);  
  0: iconst_0  
  1: istore_3  
  2: iload_3  
  3: aload_1  
  4: arraylength  
  5: if_icmpge 24  
  8: aload_1  
  9: iload_3  
10: iaload  
11: iload_2  
12: if_icmpne 17  
15: iconst_1  
16: ireturn  
17: iload_3  
18: iconst_1  
19: iadd  
20: istore_3  
21: goto 2  
24: iconst_0  
25: ireturn
```

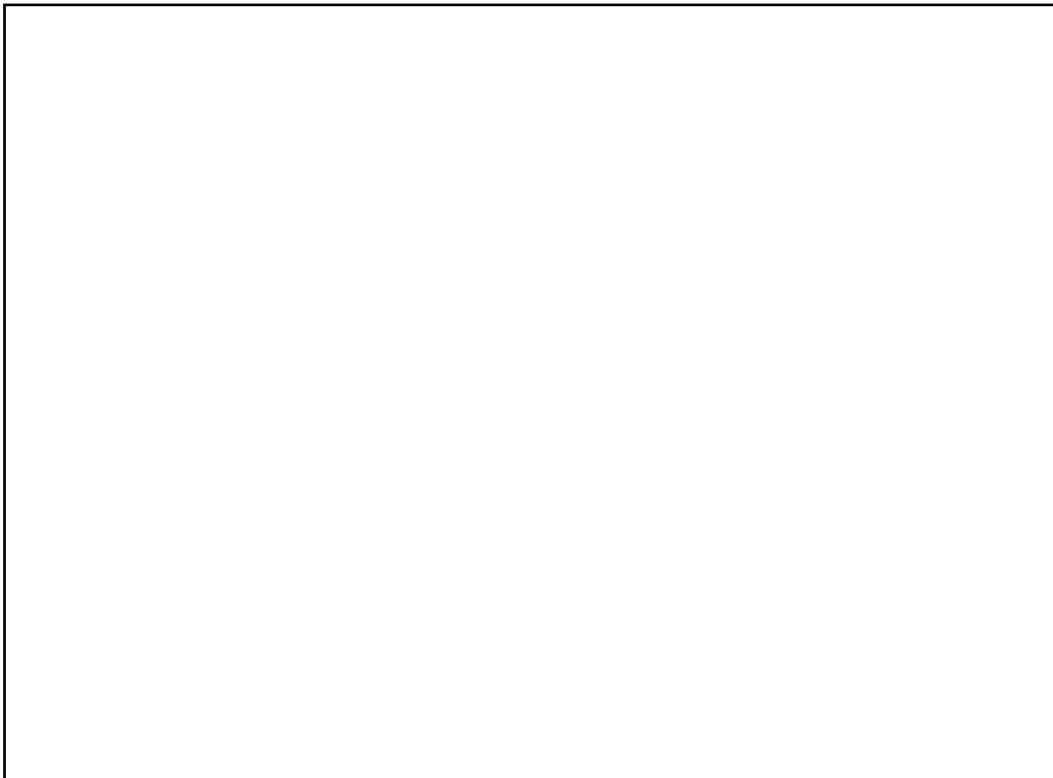
i. (5 marks) In the box below, give Java source code equivalent to the bytecode above:

NOTE: Appendix A on p21 provides an overview of bytecode instructions for reference.

- (b) **(2 marks)** Branch instructions in Java bytecode use *relative addressing*. Briefly, explain what this means.

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

- (c) **(6 marks)** Using an example to illustrate both Java source and the generated bytecode, explain what is meant by the term *short circuiting*.

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

(d) (7 marks) Translate the following method into Java bytecode:

```
1    public void fill(int[] items, int item) {  
2        for(int i=0;i!=items.length;i=i+1) {  
3            items[i] = item;  
4        }  
5    }
```

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.

5. Machine Code

(20 marks)

Consider the following program written in WHILE:

```
1  int max(int x, int y) {  
2    if(x < y) { return y; }  
3    else { return x; }  
4  }
```

- (a) (6 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: Appendix B on page 22 provides an overview of x86_64 instructions for reference.

(b) On x86_64, the `rbp` register normally holds the *frame pointer*.

i. **(4 marks)** Briefly, discuss what the frame pointer is used for.

ii. **(4 marks)** Briefly, discuss whether a frame pointer is needed for method `max()`.

(c) **(6 marks)** Briefly, discuss why *register allocation* is important for the performance of compiled programs.

6. Memory Models

(20 marks)

(a) In the following *litmus tests*, x and y are *shared* variables, whilst $r1$ and $r2$ are *local* variables. Assume all variables are initialised with 0.

i. (5 marks) Under the *Sequential Consistency* model, can executing following program ever leave both $r1=1$ and $r2=1$ at the end? Justify your answer.

Thread 1	Thread 2
$r1 = x;$	$r2 = y;$
$y = 1;$	$x = 1;$

ii. (5 marks) Under the *Total Store Ordering (TSO)* model, can executing the following program ever leave both $r1=0$ and $r2=0$ at the end? Justify your answer.

Thread 1	Thread 2
$y = 1;$	$x = 1;$
$r1 = x;$	$r2 = y;$

(b) A *data race* can occur when two threads access the same shared variable at the same time.

i. (2 marks) Can a data race occur if both threads *read* from the shared variable?

ii. (2 marks) Briefly, discuss how data races can cause variables to be assigned unexpected values.

(c) (6 marks) Let `c` be an instance of `Channel` (defined below) and suppose **Thread 1** repeatedly calls `c.write(1)` and **Thread 2** repeatedly calls `c.read()`.

```

1 class Channel {
2     private int value = 0;
3     private volatile boolean ready = false;
4
5     public void write(int v) {
6         value = v;
7         ready = true;
8     }
9     public int read() {
10        while(!ready) { }
11        return value;
12    } }

```

On Java 5 (or later) can **Thread 2** ever read the value `0`? Justify your answer.

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.

Appendix A: Java Bytecodes

aaload	Load reference element from array onto stack.	..., aref, index \Rightarrow ..., ref
aastore	Store reference element into array from stack.	..., ref, index, val \Rightarrow ...
aload <i>n</i>	Load reference from local variable <i>n</i> onto stack.	... \Rightarrow ..., ref
areturn	Return reference from method.	..., ref \Rightarrow ...
arraylength	Push array length on stack.	..., aref \Rightarrow ..., int
astore <i>n</i>	Store reference into local variable <i>n</i> from stack.	..., ref \Rightarrow ...
bipush <i>c</i>	Load integer byte constant <i>c</i> onto stack.	... \Rightarrow ..., int
dup	Duplicate top item on stack.	..., val \Rightarrow ..., val, val
iadd	Add two ints on stack.	..., int, int \Rightarrow ..., int
iaload	Load int element from array onto stack.	..., aref, index \Rightarrow ..., int
iastore	Store int element into array from stack.	..., aref, index, val \Rightarrow ...
iconst_c	Load integer constant <i>c</i> onto stack.	... \Rightarrow ..., int
idiv	Divide two ints on stack.	..., int, int \Rightarrow ..., int
iload <i>n</i>	Load int from local variable <i>n</i> onto stack.	... \Rightarrow ..., int
imul	Multiply two ints on stack.	..., int, int \Rightarrow ..., int
ineg	Negate int on stack.	..., int \Rightarrow ..., int
invokeinterface	Invoke interface method.	..., aref[val, [val, ...]] \Rightarrow [val]
invokespecial	Invoke special instance method (e.g. initialisation).	..., aref[val, [val, ...]] \Rightarrow [val]
invokestatic	Invoke static method.	... [val, [val, ...]] \Rightarrow [val]
invokevirtual	Invoke instance method.	..., aref[val, [val, ...]] \Rightarrow [val]
ireturn	Return int from method.	..., int \Rightarrow ...
istore <i>n</i>	Store int into local variable <i>n</i> from stack.	..., int \Rightarrow ...
isub	Subtract two ints on stack.	..., int, int \Rightarrow ..., int
if<cond>	Branch if int comparison with zero succeeds.	..., int \Rightarrow ...
if_acmp<cond> <i>d</i>	Branch to <i>d</i> if reference comparison succeeds.	..., ref, ref \Rightarrow ...
if_icmp<cond> <i>d</i>	Branch to <i>d</i> if int comparison succeeds.	..., int, int \Rightarrow ...
ldc <i>c</i>	Load constant (e.g. integer or string) <i>c</i> on stack.	... \Rightarrow ..., int
new <i>C</i>	Create a new object of class <i>C</i> \Rightarrow ..., ref
goto <i>d</i>	Branch unconditionally to <i>d</i> \Rightarrow ...
pop	Pop top item off stack.	..., val \Rightarrow ...
return	Return from method.	... \Rightarrow ...
sipush <i>c</i>	Load integer word constant <i>c</i> onto stack.	... \Rightarrow ..., int

Appendix B: 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.