Surname: . . . . **First Name:** . . . . **Student ID:** . . . .

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA

UNIVERSITY OF WELLINGTON

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.

A lexer is responsible for grouping characters into *tokens* that can then be fed into the parser.

ii. **(2 marks)** Parser.

A parser is responsible for reading a sequence of tokens and checking whether (or not) they adhere to the grammar of the language in question. A parser typically produces an abstract syntax tree representation.

iii. **(2 marks)** Abstract Syntax Tree.

An abstract syntax tree is a programmatic representation of source program arranged as a tree.

(b) Consider the following grammar:

$$\begin{array}{lcl} E & \longrightarrow & N \mid \boxed{(}\, E \,\boxed{,}\, E \,\boxed{)} \\ N & \longrightarrow & \boxed{0} \mid \boxed{1} \mid \boxed{2} \mid \boxed{3} \mid \boxed{4} \mid \boxed{5} \mid \boxed{6} \mid \boxed{7} \mid \boxed{8} \mid \boxed{9} \end{array}$$

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

`0`	YES
`(0,1)`	YES
`(00,1)`	NO
`((0,1))`	NO

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

```
interface Expr {}

class Tuple implements Expr {
  private final Expr lhs, rhs;
  public Tuple(Expr l, Expr r) { lhs = l; rhs = r; }
}

class Number implements Expr {
    private final int n;
    public Number(int n) { this.n = n; }
}
```

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; }

  public Expr parseExpr() {
    char c = input.charAt(offset);
    if(c == '(') { return parseTuple(); }
    else { return parseNumber(); }
  }
  public Expr parseTuple() {
    match('(');
    Expr l = parseExpr();
    match(',');
    Expr r = parseExpr();
    match(')');
    return new Tuple(l,r);
  }
  public Expr parseNumber() {
    char c = input.charAt(offset);
    return new Number(Integer.parseInt("" + c));
  }
  public void match(char c) {
    if(input.charAt(offset++) != c) {
      throw new RuntimeException("error");
    }
  }
}
```

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.

$\texttt{s} ::=$ (Statements) $\mid \texttt{T x = e}$ (Declarations) $\mid \texttt{x = e}$ (Assignments) $\mid \texttt{s ; s}$ (Sequence) $\mid \texttt{e}$ (Expressions) $\texttt{e} ::=$ (Expressions) $\mid \texttt{c}$ (Integers) $\mid \texttt{x}$ (Variables) $\mid *\texttt{e}$ (Dereference) $\mid \texttt{new e}$ (Allocation) $\texttt{T} ::=$ (Types) $\mid \&\texttt{T}$ (Reference type) $\mid \texttt{int}$ (Int type) $\mid \texttt{void}$ (Void type)	$\dfrac{}{\vdash \texttt{c} : \texttt{int}}$ (T-INT) $\quad \dfrac{\texttt{x} : \texttt{T} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}}$ (T-VAR) $\dfrac{\Gamma \vdash \texttt{e} : \&\texttt{T}}{\Gamma \vdash *\texttt{e} : \texttt{T}}$ (T-Deref) $\quad \dfrac{\Gamma \vdash \texttt{e} : \texttt{T}}{\Gamma \vdash \texttt{new e} : \&\texttt{T}}$ (T-New) $\dfrac{\Gamma \vdash \texttt{x} : \texttt{T} \quad \Gamma \vdash \texttt{e} : \texttt{T}}{\Gamma \vdash \texttt{T x = e} : \texttt{void}}$ (T-Decl) $\dfrac{\Gamma \vdash \texttt{x} : \texttt{T} \quad \Gamma \vdash \texttt{e} : \texttt{T}}{\Gamma \vdash \texttt{x = e} : \texttt{void}}$ (T-Assign) $\dfrac{\Gamma \vdash \texttt{s}_1 : \texttt{T}_1 \quad \Gamma \vdash \texttt{s}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{s}_1 \texttt{ ; } \texttt{s}_2 : \texttt{T}_2}$ (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.

$\Gamma \vdash \texttt{x = 1 : void}$

$\Gamma = \{\texttt{x} \mapsto \texttt{int}\}$

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

None exists because y cannot be both &**int** and **int**

$\Gamma \vdash \texttt{x = new y ; *x : int}$

$\Gamma = \{\texttt{x} \mapsto \&\texttt{int}, \texttt{y} \mapsto \texttt{int}\}$

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

$\Gamma = \{\texttt{x} \mapsto \&\texttt{int}, \texttt{y} \mapsto \&\texttt{int}\}$.

$\Gamma \vdash \texttt{x = y ; y = *x : void}$

None exists because no valid type for x exists (e.g. not &**int**, nor &&**int**, etc)

(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.

$$\frac{\Gamma \vdash \texttt{e} : \texttt{\&T}}{\Gamma \vdash \texttt{delete}\ \texttt{e} : \texttt{void}}$$

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.

A program is stuck if it cannot continue executing, but has not yet reduced to a value. For example, the sequence "`&int p` $=$ `new 1 ; delete p ; delete p`" will become stuck on the last statement since the memory referred to by `p` was already deallocated.

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.

The progress theorem states that a well-typed program is not stuck (either its a value or it can reduce). Unfortunately, the `delete` statement means that well-typed programs *can* get stuck. For example, the program "`&int p` $=$ `new 1 ; delete p ; delete p`" is well typed but gets stuck. The reason this happens is that the typing environment doesn't include information about whether heap data has been deallocated or not.

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:

```
&int p = new 123;
&int q;
// p is defined, q is undefined
if x >= 0 {
   q = p;
   // p is undefined, q is defined
}
// 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.

Static analyses cannot reason with perfect precision, and must draw safe (i.e. conservative) conclusions. In uniqueness analysis, for example, the analysis may not know for sure whether a variable was moved or not. But if it thinks it could be, then it must assume it has moved. For example, consider this variation on the program above:

```
assert x < 0;
...
&int p = new 123;
&int q;
if x >= 0 { q = p; }
```

In this example, we know that q is never moved and, hence, is defined after the last statement. But, our uniqueness analysis cannot reason about conditions in this way.

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

A depth-first traversal visits every node in the control-flow graph exactly once. However, this is not sufficient for tracking uniqueness information around loops. Consider the following:

```
1 &int p = new 123;
2 while x < n {
3    &int q = p;
4 }
5 return p;
```

A depth-first traversal of the CFG for this graph will, in essence, take two paths: $1 \rightarrow 2 \rightarrow 5$ and $1 \rightarrow 2 \rightarrow 3$. In both of these paths, uniqueness information appears correct. In order to catch the problem, we must propagate information coming out of 3 back around the loop so that it eventually propagates into 5.

(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`.

```
consume(assert e)  =  ∅
consume(delete e)  =  consume(e)
   consume(x = e)  =  consume(e)

  consume(x == y)  =  ∅
   consume(new e)  =  consume(e)
       consume(x)  =  ∅, if typeOf(x)=int
                   =  {x}, if typeOf(x)=&int
```

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

$$
\begin{aligned}
UNIQ_{IN}(0) &= ARGS(0) \\
UNIQ_{IN}(v) &= \bigcap_{w \to v \in E} UNIQ_{OUT}(w) \\
UNIQ_{OUT}(v) &= (UNIQ_{IN}(v) - \texttt{consumed}(v)) \;\cup\; DEF_{AT}(v)
\end{aligned}
$$

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.

```java
public boolean contains(int[] items, int item) {
   int i = 0;
   while(i < items.length) {
      if(items[i] == item) {
         return true;
      }
      i= i + 1;
   }
   return false;
}
```

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

Relative addressing means that the operand in the instruction identifies the target address as an offset from the current bytecode. The alternative is *absolute addressing* where the full address is encoded in the instruction.

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

Short circuiting is where the right-hand side of a logical operator (e.g. `&&`) is not executed when the outcome of the operator is already known. For example, consider this program:

```
public int filter(int item) {
   if(item < 0 || item > 16) { return 0; }
   else { return item; }
}
```

This would be translated into something like this:

```
 0: iload_1
 1: iflt 10
 4: iload_1
 5: bipush 16
 7: if_icmple 12
10: iconst_0
```

We see that if the first condition is true, the second condition is not even evaluated.

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

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

```
public void fill(int[], int);
 iconst_0
 istore_3
.L1
 iload_3
 aload_1
 arraylength
 if_icmpeq L2
 aload_1
 iload_3
 iload_2
 iastore
 iload_3
 iconst_1
 iadd
 istore_3
 goto L1
.L2
 return
```

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:

```
int max(int x, int y) {
  if(x < y) { return y; }
  else { return x; }
}
```

(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.

```
max:
        cmpl %rdi, %rsi
        jge .L2
        movl %rdi, %eax
        ret
.L2:
        movl %rsi, %eax
        ret
```

(b) On `X86_64`, the `rbp` register normally holds the *frame pointer*.

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

The frame pointer points to the start of the method's stack frame, and is used to access local variables to the method stored on the stack frame. Space may also be used for parameters and return values if/when these are not being passed in registers.

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

A framepointer is not required for method `max()` because all local variables can be stored in registers and, hence, a stack frame is not required.

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

Register allocation is important because reading / writing values from registers is much faster than from main memory. Thus, allocating variables into registers improves overall performance, especially if those variables are accessed many times. For example, if there is a loop then it is desirable to have all variables used in that loop stored in registers.

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;`

No. Either `r1 = x` or `r2 = y` is always executed before the other instructions and, hence, either `r1` or `r2` must be zero.

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;`

Yes. Assume the sequence `y=1 ; x=1; r1=x; r2=y;`. Under TSO, both writes to `x` and `y` maybe stuck in the store buffer when both `r1=x` and `r2=y` are executed.

(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?

No, at least one write is required for a data race.

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

If a variable is read and written at the same time, this can result in *tearing*. Specifically, where the value read contains part of the old value and part of the new value.

(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()`.

```
class Channel {
   private int value = 0;
   private volatile boolean ready = false;

   public void write(int v) {
      value = v;
      ready = true;
   }
   public int read() {
      while(!ready) { }
      return value;
} }
```

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

No. Since `ready` is marked **`volatile`** this has a *synchronising* effect in Java 5 or later. In effect, this means any access to this variable introduces a memory barrier which synchronises all cached variables with main (i.e. shared) memory. For example, on X86 this would result in the store buffer for each processor being flushed.

* * * * * * * * * * * * * * *

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

Appendix B: x86_64 Machine Instructions

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