TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2019

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.	Advanced Topics	20
	Total	120

1. Grammars and Parsing **(20 marks)**

(a) **(6 marks)**

Briefly describe the *two* conditions a context-free grammar must satisfy in order to be considered LL(1) (i.e. suitable for a recursive descent parser).

Condition 1:

Condition 2:

(b) Consider the following grammar, where nonterminals are in italics, terminals are enclosed in double quotes, id denotes an identifier, and $\langle empty \rangle$ denotes an empty string.

Header	::=	*RPart* id "(" *APart* ")"
RPart	::=	id | $\langle empty \rangle$
APart	::=	id | id "," *APart*

i. **(8 marks)** Explain the ways in which this grammar violates the LL(1) conditions, and how they would affect the behaviour of a recursive descent parser based on this grammar.

ii. **(6 marks)** Write an equivalent LL(1) grammar.

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)**

(a) **(12 marks)**

For each of the following kinds of errors, say whether that kind of error can be detected by a type checker in a strongly typed language, and **explain your answer**.

(i) Adding an integer to a Boolean value.

(ii) Calling a function or method with the wrong number of arguments.

(iii) Division by zero.

(iv) Calling a non-existent method on an object.

(v) Missing case label in a `switch` statement.

(vi) Dereferencing a `null` pointer.

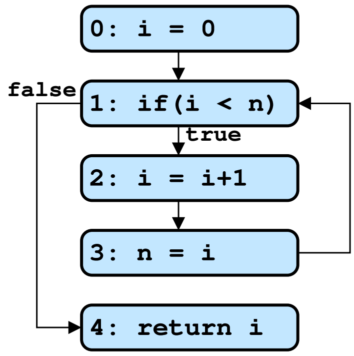
(b) **(8 marks)**

Adding union types to a programming language increases the expressiveness of the language, but makes type checking more complicated. Discuss the main issues that arise in testing for type equivalence and subtype compatibility in the presence of union types.

3. Static Analysis **(20 marks)**

The *definite unassignment* phase is used in Java to check that **`final`** variables are only assigned once. The following illustrates:

```java
int aMethod(final int n) {
  int i = 0;
  while(i < n) {
    i = i + 1;
    n = i;
  }
  return i;
}
```



The above method fails definite unassignment because the **`final`** variable `n` *may be assigned more than once*. The definite unassignment algorithm determines, at each point, which variables *may have been assigned* at that point.

(a) **(5 marks)** Explain briefly, using an example, why no algorithm accurately can detect *all cases* of definite unassignment.

(b) **(5 marks)** Using the `aMethod()` example above, explain briefly why a depth-first traversal algorithm is *insufficient* for checking definite unassignment.

(c) **(10 marks)** Briefly, outline how an algorithm for detecting definite unassignment would work. You may give the *dataflow equations* if this helps.

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. Java Bytecode **(20 marks)**

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

```
public int f(int[]);
   0: iconst_0
   1: istore_2
   2: iconst_0
   3: istore_3
   4: iload_2
   5: aload_1
   6: arraylength
   7: if_icmpge 23
  10: iload_3
  11: aload_1
  12: iload_2
  13: iaload
  14: iadd
  15: istore_3
  16: iload_2
  17: iconst_1
  18: iadd
  19: istore_2
  20: goto 4
  23: iload_3
  24: ireturn
```

i. **(5 marks)** In the box below, give Java source code equivalent to the bytecode above:
NOTE: Appendix A on p19 provides an overview of bytecode instructions for reference.

ii. **(3 marks)** What is the *maximum stack height* of the above method? Be sure to show your working by indicating below the height at each point.

```
public int f(int[]);
   0: iconst_0
   1: istore_2
   2: iconst_0
   3: istore_3
   4: iload_2
   5: aload_1
   6: arraylength
   7: if_icmpge 23
  10: iload_3
  11: aload_1
  12: iload_2
  13: iaload
  14: iadd
  15: istore_3
  16: iload_2
  17: iconst_1
  18: iadd
  19: istore_2
  20: goto 4
  23: iload_3
  24: ireturn
```

(b) For each of the following JVM error messages, briefly discuss what might have caused the problem. You may use examples to illustrate as necessary.

i. **(2 marks)** "`Unable to pop operand off an empty stack`"

ii. **(2 marks)** "`Accessing value from uninitialized register`"

iii. **(2 marks)** "`Inconsistent stack height`"

(c) **(6 marks)** Translate the following method into Java bytecode:

```
public static int fib(int n) {
  if(n == 0 || n == 1) { return n; }
  else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

5. Machine Code **(20 marks)**

Consider the following function, `mul`, written in `x86_64` assembly language:

```
mul:
        pushq %rbp
        movq %rsp, %rbp
        subq $16, %rsp
        movq $0, %rax
        movq %rax, -8(%rbp)
        movq $0, %rax
        movq %rax, -16(%rbp)
L1:
        movq -16(%rbp), %rax
        movq 32(%rbp), %rbx
        cmpq %rbx, %rax
        jge L2
        movq -8(%rbp), %rax
        movq 24(%rbp), %rbx
        addq %rbx, %rax
        movq %rax, -8(%rbp)
        movq -16(%rbp), %rax
        movq $1, %rbx
        addq %rbx, %rax
        movq %rax, -16(%rbp)
        jmp L1
L2:
        movq -8(%rbp), %rax
        movq %rax, 16(%rbp)
        movq %rbp, %rsp
        popq %rbp
        ret
```

NOTE: the Appendix on page 20 provides an overview of `x86_64` machine instructions for reference.

(a) **(5 marks)** Function parameters are normally passed *on the stack* or *in registers*. How are parameters passed in the above function? Justify your answer.

(b) **(5 marks)** Translate the `mul` function into WHILE.

(c) **(5 marks)** During execution, *stack frames* are created to hold critical information. Briefly, discuss the *stack frame layout* for the `mul` function using diagrams to illustrate.

(d) **(5 marks)** The implementation of `mul` is not efficient. For example, it uses more machine instructions than necessary. Briefly, discuss how it can be rewritten to improve efficiency.

6. Advanced Topics **(20 marks)**

(a) **(10 marks)**

(i) Briefly explain how implementing method calls in an object-oriented language differs from implementing function calls in a language like C, and why method calls can potentially be less efficient than C-like function calls.

(ii) Discuss how static analysis techniques can be used to analyse method declarations and calls in an object-oriented program, and use this information to improve the efficiency of method calls.

(b) **(10 marks)**

Programmers tend to think of their programs as executing on a relatively simple computer, such as a PDP11, and many compiler optimisations are based on similar assumptions.

Discuss some of the ways in which modern machines differ from this simple model, and the impact that this has for code generation and optimisation in a compiler.

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

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.