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:

For any two productions $N \rightarrow \alpha$ and $N \rightarrow \beta$ from the same non-terminal $N$, we must have $\text{first}(\alpha) \cap \text{first}(\beta) = \emptyset$ (otherwise the grammar is ambiguous)

Condition 2:

For any non-terminal $N$ which has a production $N \rightarrow \epsilon$, we must have $\text{first}(N) \cap \text{follow}(N) = \emptyset$ (otherwise the grammar is ambiguous)

(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.

- Grammar has ambiguity around productions for *RPart* because we have first(RPart) $\cap$ follow(RPart) $= \{\text{id}\}$. This violates condition 2. This is a problem for a recursive descent parser as it will need to make a decision when parsing *RPart* which production to use. For example, it might greedily consume an `id` and never choose the empty production.
- Grammar has ambiguity around productions for *APart* because they have identical first() sets. This violates condition 1. A recursive descent parser could work around this, however, as having parsed an `id` it can use a lookahead to see whether a comma follows.

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

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

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)**

(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.

Yes, can be detected by type checker because addition operator expects operands of type integer.

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

Yes, can be detected by type checker because it must know which function or method is being called, hence it must know how many parameters are required.

(iii) Division by zero.

No, this cannot be detected by a type checker. This is because the integer type does not contain enough information to tell us whether a variable can be zero or not

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

Yes, can be detected by type checker since the type of the operand will tell us what methods may be invoked.

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

In principle, this could be detected by a type checker. However, generally speaking it is not detected due to the very large number of cases (e.g. for variable of integer type).

(vi) Dereferencing a `null` pointer.

In a language like Java, this cannot be detected by the type checker because a variable of e.g. type `String` can be a valid reference or **`null`** and the type checker cannot distinguish these cases. However, in other languages (e.g. WHILE) it can be detected by the type checker.

(b) **(8 marks)**

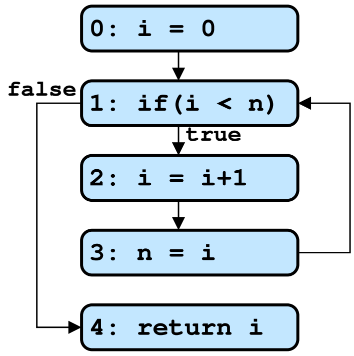
Adding union types to a programming language increases the expressiveness of the language, but makes type checking more complicated. Discuss the main issues that arise in testing for type equivalence and subtype compatibility in the presence of union types.

Union types introduce a separation between the *meaning* of a type and its *syntax*. For example, the type `int` is expressed differently from $\texttt{int} \vee \texttt{int}$ but has the same meaning. Developing an algorithm to do this correctly is challenging and hard to express using type rules alone. This results in a gap between what the algorithm can do, and what we would ideally like it to do. We say that such an algorithm is *sound* if, when it claims $\texttt{T}_1 \leq \texttt{T}_2$ for some types $\texttt{T}_1$ and $\texttt{T}_2$, this is always true. Likewise, such an algorithm is *complete* if, whenever it is true that $\texttt{T}_1$ is a subtype of $\texttt{T}_2$ the algorithm can conclude that $\texttt{T}_1 \leq \texttt{T}_2$.

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:

```
1  int aMethod(final int n) {
2    int i = 0;
3    while(i < n) {
4      i = i + 1;
5      n = i;
6    }
7    return i;
8  }
```




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.

Static analyses cannot reason with perfect precision, and must draw safe (i.e. conservative) conclusions. In definite unassignment analysis, for example, the analysis may not know for sure whether a variable has been defined or not. But if it thinks it might be, then it must assume it has been. For example, consider program:

```
final int p;

if x >= 0 { p = 1; }
if x < 0 { p = 0; }
```

In this example, we know that `p` is never defined twice. But, our definite unassignment analysis cannot reason about conditions in this way.

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

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. Considering `aMethod()` above, a depth-first traversal of the CFG for this graph will, in essence, take two paths: $2 \rightarrow 3 \rightarrow 7$ and $2 \rightarrow 3 \rightarrow 4 \rightarrow 5$.
In both of these paths, variable `n` is defined at most once. In order to see that it could be defined more than once, we must propagate information coming out of 5 back around the loop so that it eventually propagates back into 5.

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

The analysis maintains the set of variables which are currently *undefined*. This is initialise with all variables in the method, except for the arguments. Whenever a variable is assigned (or declared with an initialise), it is removed from the set. At control-flow join points (e.g. after a conditional) we require that, for a variable to be still considered undefined, it must have been undefined on all incoming branches. The dataflow equations are given as follows:

$$UNDEF_{IN}(v) = \bigcap_{w \to v \in E} UNDEF_{OUT}(w)$$
$$UNDEF_{OUT}(v) = UNDEF_{IN}(v) - DEF_{AT}(v)$$

These questions make use of a function $\mathtt{DEF}_{\mathtt{AT}}(\mathtt{v})$ which returns the set of variables defined (e.g. assigned) at a given node `v` in the control-flow graph. This function was given in lectures for the definite assignment 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.

**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.

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

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
```

maxheight = 3

(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`"

A bytecode which expects at least one operand on the stack is being used on an empty stack. For example, if a method began with `istore_2` we might see this error.

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

A bytecode is reading a given register which has not yet been initialised. For example, if the first bytecode of method `f(int[])` was `iload_2`, this error would be generated

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

This occurs when, at a join point in the control-flow graph, the stack heights from incoming paths are not the same. For example, the error would be given for this program:

```
...
ifeq L1
iload_0
L2:
```

(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);
  }
}
```

```
public static int fib(int);
 iload_0
 ifeq L1
 iload_0
 iconst_1
 if_icmpne L2
L1:
 iload_0
 ireturn
L2:
 iload_0
 iconst_1
 isub
 invokestatic fib(int)
 iload_0
 iconst_2
 isub
 invokestatic fib(int)
 iadd
 ireturn
}
```

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.

Parameters are passed on the stack in this example. This is evident because of instructions such as "`movq 24(%rbp), %rbx`" which are loading values from locations *above* the frame pointer. These must identify parameters passed to the function.

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

```
int mul(int m, int n) {
    int r = 0;
    int i = 0;
    //
    while i < m {
        r = r + n;
        i = i + 1;
    }
    //
    return r;
}
```

(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.

The stack frame layout for `mul` looks as follows:

| | |
|---|---|
| **+32** | **int m** |
| **+24** | **int n** |
| **+16** | **return value** |
| **+08** | **return address** |
| **0** | **old frame pointer** |
| **-08** | **int r** |
| **-16** | **int i** |

(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.

The implementation of `mul` could be made more efficient by storing local variables in registers. For example, `r` and `i` could be stored in the `%rdi` and `%rsi` registers respectively. Depending on the calling convention being used, these might need to be saved on the stack at the beginning of the method so they could be recalled at the end.

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.

Method calls in object-oriented languages (e.g. Java) are normally implemented using a *virtual dispatch table* (or *vtable* for short). This allows methods to be *overriden* in subclasses. However, it also means that calling such a method requires first *reading* a function pointer from the vtable and then performing a indirect invocation on it. In contrast, C-like function calls are done using static invocations directly to the method being called.

(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.

A technique such as *Class Hierarchy Analysis (CHA)* can be used to help analyse method calls in object-oriented programs. This works by determining, for a variable of a given type, the set of possible methods that could be dispatched to based on the inheritance hierarchy. The analysis must then conservatively assume that any potential target could be called in practice. For example, consider this code:

```java
class A { int f() { return 0; } }
class B extends A { int f() { return 1; } }

public class Test {
   public static void main(String[] args) {
      A a = new A();
      B b = new B();
      a.f(); // call #1
      b.f(); // call #2
   }
}
```

In this case, CHA would conclude the targets for `a.f()` include both `A.f()` and `B.f()`, whilst for `b.f()` that `B.f()` is the only target.

(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 `int`s 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 `int`s on stack. | `...,int,int` ⇒ `...,int` |
| `iload` *n* | Load `int` from local variable *n* onto stack. | `...` ⇒ `...,int` |
| `imul` | Multiply two `int`s 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 `int`s 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. |

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.