TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2010

END-YEAR

SWEN 430

Compiler Engineering

Time Allowed: Three Hours

Instructions:

- *Read each question carefully before attempting it.*
- This examination will be marked out of **180** marks.
- Answer all questions.
- You may answer the questions in any order. Make sure you clearly identify the question you are answering.
- Non-electronic foreign-language English dictionaries are permitted.
- Reference material is NOT PERMITTED.
- Calculators are NOT PERMITTED.
- Use of mobile phones, laptop computers, PDAs or other electronic devices is NOT PERMITTED.

Questions		**Marks**
1.	Parsing and Grammars	[30]
2.	The Typing Stage	[25]
3.	Intermediate Representation	[15]
4.	Deadcode Elimination	[20]
5.	Definite Unassignment	[30]
6.	Java Bytecode	[30]
7.	Interprocedural Analysis	[30]

Syntax for a subset of Java Intermediate Language (JIL)

Types:

$$
\begin{array}{rcl}
\texttt{T}_\texttt{P} & \longrightarrow & \texttt{void} \mid \texttt{boolean} \mid \texttt{int} \\
\texttt{T}_\texttt{C} & \longrightarrow & \texttt{C} \\
\texttt{T} & \longrightarrow & \texttt{T}_\texttt{P} \mid \texttt{T}_\texttt{C} \mid \texttt{null} \mid \bot \mid \top \\
\texttt{T}_\texttt{F} & \longrightarrow & (\overline{\texttt{T}}) \longrightarrow \texttt{T}
\end{array}
$$

Syntax:

$$
\begin{array}{rcl}
\texttt{D} & \longrightarrow & \texttt{class}\ \texttt{T}_\texttt{C}\ [\texttt{extends}\ \texttt{T}_\texttt{C}]\ \{\ \overline{\texttt{T}\ \texttt{v}_1 = \texttt{v}_2\texttt{;}}\ \overline{\texttt{M}}\ \} \\
\texttt{M} & \longrightarrow & \texttt{T}\ \texttt{m}(\overline{\texttt{T}\ \texttt{x}})\ [\texttt{throws}\ \overline{\texttt{T}_\texttt{C}}]\ \{\ \texttt{var}\ \overline{\texttt{v}}\ \texttt{;}\ \overline{\texttt{R}}\ \} \\
\texttt{R} & \longrightarrow & \texttt{L:} \mid \texttt{nop;} \mid \texttt{goto L;} \mid \texttt{S}\ [\texttt{,}\ \overline{\texttt{T}_\texttt{C}\ \texttt{goto L}}]\texttt{;} \\
\texttt{S} & \longrightarrow & \texttt{return e} \mid \texttt{throw e} \mid \texttt{v} = \texttt{e} \mid \texttt{e}_1\texttt{.v} = \texttt{e}_2 \mid \texttt{if(e) goto L} \\
\texttt{e} & \longrightarrow & \texttt{e}_1\ \texttt{bop}\ \texttt{e}_2 \mid \texttt{e.m[}\texttt{T}_\texttt{F}\texttt{]}(\overline{\texttt{e}}) \mid (\texttt{T})\ \texttt{e} \mid \texttt{e.v} \mid \texttt{t} \\
\texttt{t} & \longrightarrow & \texttt{new}\ \texttt{T}_\texttt{R}() \mid \texttt{v} \mid \texttt{i} \mid \texttt{b} \mid \texttt{null} \\
\texttt{v} & \longrightarrow & [\texttt{a-zA-Z\_}]^+[\texttt{a-zA-Z0-9\_}]^* \\
\texttt{i} & \longrightarrow & \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \\
\texttt{b} & \longrightarrow & \texttt{true,false} \\
\texttt{bop} & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid < \mid \leq \mid \geq \mid > \mid == \mid \texttt{!}= \\
\end{array}
$$

Question 1. Parsing and Grammars [30 marks]

(a) [3 marks] Describe the difference between a *scanner* and a *parser*.

(b) [2 marks] Define the terms *Abstract Syntax Tree* and *Concrete Parse Tree*.

(c) [5 marks] For the first project you had a choice of implementing the checks at the type checking stage, working with *Abstract Syntax Tree*, or at a later stage working with JKit's *Intermediate Language*. State the advantages and disadvantages of working with the *Abstract Syntax Tree* in the context of your project.

(d) [10 marks] Write an easy to read grammar for a language described as follows.

> The language describes commands that can be given to lego robots. It has basic commands like *turn*, *move*, *drop*, *pickup* where turning and moving commands can be given an integer to specify the amount to turn or move by.
>
> There are balls around the room, in which these lego robots roam, that they can pickup and drop as long as they are *touching* it and a robot's sensor can return a boolean value *touching-ball*.
>
> Thus the primitive types are numbers (integers) and booleans and additional values that can be numbers or booleans as returned by sensors include: *distance-to-closest-ball*, *distance-to-closest-robot*, *touching-robot*, *touching-other-obstacle*.
>
> The robot language supports the normal *while* loop and *if* statement with the body separated by parenthesis '{' and '}' and condition inside brackets: '(' and ')'.
>
> The language should have support for variables (both declaring and assigning them). All the instructions are separated by semicolons: ';'.

(e) [10 marks] Re-write your grammar above in a form that would be acceptable by the LL(1) parser.

Question 2. The Typing Stage [25 marks]

Consider the following type system describing a trivial subset of the Java language. Its syntax is as follows:

$$\texttt{L ::= class C } \{\overline{\texttt{C f;}}\ \texttt{K}\} \qquad \texttt{K ::= C}(\overline{\texttt{C f}})\{\overline{\texttt{this.f = f;}}\}$$
$$\texttt{e ::= x | new C}(\overline{\texttt{e}}) \texttt{ | e.f | e.f = e'.f'}$$

L is a class declaration for a class named C and $\overline{\texttt{C f}}$ stands for a list of fields each having a type C and a name f. K describes a constructor that must initialise all fields in the class. There are no methods or casts in our language and only four possible expressions: variable (x), class instantiation ($\texttt{new C}(\overline{\texttt{e}})$), field access (e.f) and field assignment.

We provide the following expression type checking rules (where the $fields(\texttt{C})$ lookup function provides a list of fields given class C):

$$\frac{\Gamma \vdash \texttt{K} = \texttt{C}(\overline{\texttt{C f}})\{\overline{\texttt{this.f = f;}}\}}{\Gamma \vdash \texttt{class C}\ \{\overline{\texttt{C f;}}\ \texttt{K}\}\ \texttt{OK}}\ \text{(T-Class)} \qquad \Gamma \vdash \texttt{x} : \Gamma(\texttt{x})\ \text{(T-Var)}$$

$$\frac{fields(\texttt{C}) = \overline{\texttt{C f}} \quad \Gamma \vdash \overline{\texttt{e : C}}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) : \texttt{C}}\ \text{(T-New)} \qquad \frac{\Gamma \vdash \texttt{e}_0 : \texttt{C}_0 \quad fields(\texttt{C}_0) = \overline{\texttt{C f}}}{\Gamma \vdash \texttt{e}_0.\texttt{f}_i : \texttt{C}_i}\ \text{(T-Field)}$$

A program in our language is a list of class declarations followed by an expression. Each class declaration should meet the T-Class rule and the expression should meet the expression type rules. Assume that class `Object` is pre-defined. Also assume that a class table CT maps class names C to class declarations L and an environment Γ maps variables to their types.

(a) [5 marks] One of the type checking rules is missing. Please write the missing type rule.

(b) [5 marks] For the following program that doesn't use the missing rule show the expression derivation if it type checks or show why it does not type check. Assume the following class declaration:

```
class Pair {
  Object first;
  Object second;
  Pair(Object first, Object second) {
    this.first = first; this.second = second;
  }
}

new Pair(new Object(), new Object()).first;
```

(Question 2 continued on next page)

(Question 2 continued)

(c) [5 marks] Consider the following ownership extension to the language above described using the following type rule (that can be applied *in addition* to the rules above). Assume that the syntax has been extended to allow each field to have an optional *owned* declaration:

$$\texttt{L} ::= \texttt{class C} \left\{ \overline{\texttt{C} \left[\texttt{owned}\right] \texttt{f;}} \texttt{K} \right\}$$

For the rule below describe what it checks for and describe the purpose behind every clause of the rule.

$$\frac{\Gamma \vdash \texttt{e}_0 \ : \ \texttt{C}_0 \qquad \texttt{C}_i \ \texttt{owned} \ \texttt{f}_i \in \mathit{fields}(\texttt{C}_0) \qquad \texttt{e}_0 = \texttt{this}}{\Gamma \vdash \texttt{e}_0.\texttt{f}_i \ : \ \texttt{C}_i} \quad \text{(T-RULE-1)}$$

(d) [5 marks] The extension above does not account for the missing rule in subquestion (a) above. Please extend it in the appropriate manner.

(e) [5 marks] In your own words, briefly discuss the purpose of a *type system*. You may choose to refer to rules and examples given in this question.

Question 3. Intermediate Representation [15 marks]

The following question is concerned with the *Java Intermediate Language (JIL)*. The syntax for the subset of JIL being considered here is given on Page 2.

(a) [15 marks] Translate each of the following three Java programs into JIL, using the syntax given above.

(i)

```
class Foo {
  boolean test(int x, int y) {
    if (x <= y) { return true; }
    while (x-- > y) {
      test(1,2);
    }
    return false;
  }
}
```

(ii)

```
class Foo {
  int countrecursively(int x) {
    int i = 0;
    int c = 0;
    while (i < x) {
      c = countrecursively(i++);
    }
    return c;
  }
}
```

(iii)

```
class Bar {
  boolean doIt(String arg1, String arg2) {
    try {
      if (arg1.equals(arg2)) { return true; }
      if (arg2.equals(arg1)) { return true; }
    } catch (NullPointerException e) { return false; }
    return false;
  }
}
```

Question 4. Deadcode Elimination [20 marks]

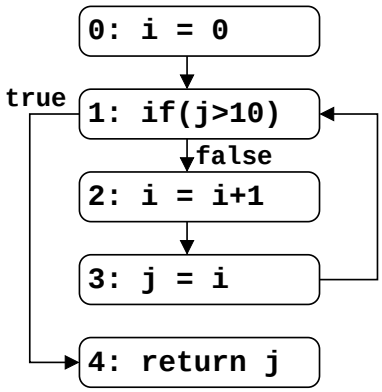
(a) [5 marks] Define the terms *control-flow graph*, *dead code*, *execution path*, *normal edge*, and *exceptional edge*.

(b) [15 marks] Imagine that you are implementing a deadcode elimination stage inside JKit. Outline your approach in English and write a detailed pseudocode algorithm that realises your approach.

Question 5. Definite Unassignment [30 marks]

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

```
int aMethod() {
   var i;
   final var j;
   i = 0;
 lab1:
   if j > 10 goto lab2;
   i = i+1;
   j = i;
   goto lab1;
 lab2:
   return j;
}
```



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

(a) [20 marks] Design a simple dataflow analysis for determining whether a variable is *definitely unassigned* before a given statement. This should operate on JIL code (recall the syntax on Page 2). Be sure to describe the following parts of the analysis: lattice of values (including $\top$ and $\bot$), the dataflow equations and any helper functions required.

(b) [10 marks] Illustrate how your definite unassignment analysis solves the above JIL program using a *forward iterative solver*. Present your answer in a tabular form, where successive columns represent the solution after each stage has completed.

Question 6. Java Bytecode [30 marks]

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

```
Number method(Integer, Double);
   0:   aload_1
   1:   ifnonnull 9
   4:   aload_2
   5:   astore_3
   6:   goto    11
   9:   aload_1
   10:  astore_3
   11:  aload_3
   12:  areturn
```

(i) [3 marks] What is the *maximum stack height* of this method?

(ii) [5 marks] Give Java source code which is equivalent to the above method.

(iii) [5 marks] Rewrite the above method to an equivalent which uses as few bytecodes as possible.

(b) Consider the following Java bytecode, which fails *bytecode verification*:

```
  boolean method(int);
     0: iconst_2
     1: istore_2
     2: iload_2
     3: aload_1
     4: if_icmpeq 13
     7: iconst_0
     8: dup
     9: goto 14
    13: iconst_1
    14: pop
    15: pop
    16: ireturn
```

(i) [6 marks] Briefly discuss the different ways in which this method fails bytecode verification.

(ii) [6 marks] The bytecode verifier employs a *least upper bound* operation on types. Discuss what this means, and how it relates to bytecode verification. Use examples to illustrate as necessary.

(iii) [5 marks] A bytecode method can fail verification because the verifier claims it uses types incorrectly, when in fact it does not. Using an example to illustrate, discuss how this can occur.

Question 7. Interprocedural Analysis [30 marks]

(a) Consider the following class:

```
public class Link {
  private Link next;

  public Link(Link next) { this.next = next; }

  public int empty_count() { return 0; }

  public int count() {
     if(next == null) {
       return empty_count();
     } else {
       return 1 + next.count();
     }
  }

  public static void main(String[] args) {
    new Link(new Link(null)).count();
  }
}

public class SpecialLink extends Link {
  public int count() {
   ...
  }
}
```

(i) [5 marks] Draw a *static call graph* for the above program.

(ii) [10 marks] *Class Hierarchy Analysis* is a technique for approximating a program's call graph. Using examples where appropriate, discuss how this works.

(iii) [5 marks] Class Hierarchy Analysis often provides an imprecise approximation of a program's call graph. Using an example, discuss how this imprecision can arise.

(iv) [10 marks] Outline a technique for approximating a program's call graph which is more precise than Class Hierarchy Analysis.

****