The University of Melbourne
Semester 2 Assessment 2008

Department of Computer Science and Software Engineering

433–361 Programming Language Implementation

Reading Time: 15 minutes

Writing Time: 3 hours

This paper has 8 pages, including this front page.

Identical Examination Papers: None

Common Content Papers: None

Authorised Materials:
This is a closed book exam. Electronic devices, including calculators and laptop computers are **not** permitted.

Calculators:
No calculators are permitted.

Instructions to Invigilators:
Students should be provided with a script book. Students may take the exam paper out of the exam room once examination finishes, but not before.

Instructions to Students:
This examination counts for 75% of the total assessment in the subject (25% being allocated to written assignments). There are 9 questions—all should be attempted. Make sure that your answers are *readable*. Any unreadable parts will have to be considered wrong. For each question and sub-question, the weight is indicated. Be careful to allocate your time according to the value of each question. The marks add to a total of 75.

Question 1 [6 marks]

Consider the language L consisting of strings of the form *cmd options*, where *cmd* consists of the strings `c0`, `c1`, ..., `c9`, and *options* consists sequences of "options", where an option is a dash (`-`) followed by a lower-case letter. Options can be given in any order. However, no option may be repeated. These are examples of strings in L:

```
c4 -a -l -s
c7 -z -y -d -k -s -c -t
c2
```

whereas the following string is *not* in L (because '`-d`' is repeated):

```
c7 -z -y -d -k -s -c -t -d -l -a -w
```

Outline how you would implement L, including which program generation tools you might use. Do not make any assumptions about available tools or library functions (such as `getopt`).

Question 2 [6 marks]

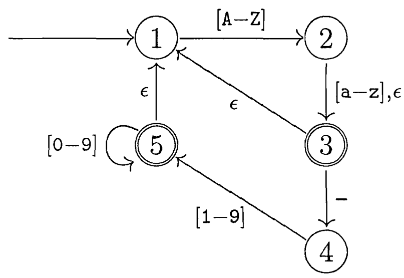
Consider the following `lex/flex` specification:

```
stuff   [a-z]+

%%

do      { return DO; }
{stuff} { return STUFF; }
docile  { return DOCILE; }
.       { return WHATEVER; }

%%
```

Give the token sequence returned by `yylex` for the input "whodovoodoo??" and also for the input "docile dodos do". (In each case the input is the string between the quotation marks.)

Question 3 [11 marks]

Chemical compounds are denoted by strings such as $CaSiO_3$ and $C_6H_{12}O_6$. There are more than 100 known elements in the periodic table, including H, He, Li, Be, B, C, N, and O. For simplicity we shall assume that any single upper-case letter, as well as any upper-case letter followed by a single lower-case letter, can denote an element. We shall also assume that subscripts in strings such as our two examples are captured by prefixing the subscript with an underscore, '_', so that the two examples are written `CaSiO_3` and `C_6H_12O_6`, respectively. Thus the following NFA captures the language of chemical compounds:



(Here [A − Z] is used as a short-hand for the set of upper-case letters, and similarly for the other intervals.)

a. Give a regular expression for the language given by the NFA above. [2 mark.]

b. Use the subset construction method to turn the NFA into an equivalent DFA. [5 marks.]

Chemists use directed equations such as

$$C_6H_{12}O_6 \rightarrow CO_2 + C_2H_5OH \tag{1}$$

to denote chemical reactions—in this case saying that the compound called glucose ($C_6H_{12}O_6$) can turn into the compounds carbon dioxide (CO_2) and ethyl alcohol (C_2H_5OH).

The following context-free grammar, **G**, with start symbol *eq*, captures this language of directed equations:

$$\begin{array}{llll} \mathbf{G}: & eq & \rightarrow & exp \texttt{ -> } exp \\ & exp & \rightarrow & exp \texttt{ + } comp \mid comp \\ & comp & \rightarrow & elt \texttt{ \_ } num \mid elt \mid comp\ elt \texttt{ \_ } num \mid comp\ elt \end{array}$$

Here *num* is the syntactic category of natural numbers, and *elt* contains the two-letter combinations described above, that is, the name of an element is an upper case letter, possibly followed by a lower case letter. An example of a string derived from *eq* is

```
C_6H_12O_6  ->  CO_2 + C_2H_5OH
```

corresponding to (1) above.

c. **G** is not LL(1). Perform grammar transformations that will produce an equivalent LL(1) grammar. [3 marks.]

d. Is the set of strings that can be derived from *eq* a regular language? Justify your answer (a simple 'yes' or 'no' will not attract a mark). [1 mark.]

Question 4 [14 marks]

Consider the following (augmented) context-free grammar:

$$\begin{array}{lllll}
(r0) & S' & \rightarrow & S \\
(r1) & S & \rightarrow & \texttt{id} \texttt{ := } C \texttt{ ;} \\
(r2) & C & \rightarrow & A\ E \\
(r3) & A & \rightarrow & \epsilon \\
(r4) & A & \rightarrow & A\ \texttt{id :=} \\
(r5) & E & \rightarrow & E\ \texttt{+}\ P \\
(r6) & E & \rightarrow & P \\
(r7) & P & \rightarrow & \texttt{id} \\
(r8) & P & \rightarrow & \texttt{(}\ C\ \texttt{)}
\end{array}$$

a. Compute the FIRST and FOLLOW sets for the grammar's non-terminals [3 marks.]

b. Construct the LR(0) machine for the grammar, that is, the finite-state machine having LR(0) sets-of-items as states and transitions that are determined by grammar symbols. [8 marks.]

c. Based on the LR(0) machine, construct the action and goto table for an SLR parser recognising the set of strings derivable from S'. [3 marks.]

Question 5 [6 marks]

a. Give a function definition and a call of that function which will compute one result with call by reference and a different result with call by value-result. [3 marks.]

b. The layout of runtime stack frames is influenced by whether or not a programming language supports nested functions. How? [3 marks.]

Question 6 [8 marks]

Consider these rules for syntax-directed generation of code that utilises short-circuit evaluation of Boolean expressions:

$S \rightarrow$ `while` B `do` S_1
$S.begin$:= $newlabel()$;
$B.true$:= $newlabel()$;
$B.false$:= $S.next$;
$S_1.next$:= $S.begin$;
$S.code$:= $gen(S.begin$ ':') || $B.code$ || $gen(B.true$ ':') || $S_1.code$ || $gen($'`goto`' $S.begin)$

$S \rightarrow$ **id** := E
$S.code$:= $E.code$ || $gen($**id**$.place$ ':=' $E.place)$

$E \rightarrow E_1$ + E_2
$E.place := newtemp()$;
$E.code := E_1.code$ || $E_2.code$ || $gen(E.place$ ':=' $E_1.place$ '+' $E_2.place)$

$E \rightarrow$ **id**
$E.place :=$ **id**$.place$;
$E.code :=$ ' '

$B \rightarrow B_1$ && B_2
$B_1.false := B.false$;
$B_1.true := newlabel()$;
$B_2.false := B.false$;
$B_2.true := B.true$;
$B.code := B_1.code$ || $gen(B_1.true$ ':') || $B_2.code$

$B \rightarrow$! B_1
$B_1.true := B.false$;
$B_1.false := B.true$;
$B.code := B_1.code$

$B \rightarrow$ **id**$_1$ < **id**$_2$
$B.code := gen($'`if`' **id**$_1.place$ '<' **id**$_2.place$ '`goto`' $B.true)$ || $gen($'`goto`' $B.false)$

Show the *code* attribute for

```
while (u<y && ! u<v) do u = u + y
```

Assume the *S.next* attribute for the topmost node in the syntax tree is (the label) 15, and that the next available label is number 20.

Question 7 [8 marks]

The machine T08 (from this year's project) has the following instructions, amongst others:

```
push_stack_frame  framesize           # Reserve stack slots
pop_stack_frame   framesize           # Free stack slots
                                      # C analogies:
load              rN, slotnum         #     rN =  x
store             slotnum, rN         #      x = rN
load_address      rN, slotnum         #     rN = &x
load_indirect     rN, slotnum         #     rN = *x
store_indirect    slotnum, rN         #     *x = rN
int_const         rN, const           #     rN = const
add_int           rN, rI, rJ          #     rN = rI + rJ
add_offset        rN, rI, rJ          #     rN = rI + rJ
sub_int           rN, rI, rJ          #     rN = rI - rJ
sub_offset        rN, rI, rJ          #     rN = rI - rJ
cmp_eq_int        rN, rI, rJ          #     rN = (rI == rJ)
cmp_ne_int        rN, rI, rJ          #     rN = (rI != rJ)
cmp_lt_int        rN, rI, rJ          #     rN = (rI <  rJ)
cmp_le_int        rN, rI, rJ          #     rN = (rI <= rJ)
move              rN, rI              #     rN = rI
branch_on_true    rN, label           # if (rN)   goto label
branch_on_false   rN, label           # if (! rN) goto label
branch_uncond     label               #           goto label
call              label               # Procedure call
return                                # Procedure return
```

A T08 code generator has been given the Kate08 program shown below, on the left, and so far it has generated the code shown on the right (initialising variables to 0).

```
proc main()
    int a[0..9];
    int i;

    i := 0;
    while i < 10 do
        a[i] := i;
        i := i+1;
    od
end
```

```
    call proc_main
    halt
proc_main:
    push_stack_frame 11
    int_const r0, 0
    store 0, r0
    store 1, r0
    store 2, r0
    store 3, r0
    store 4, r0
    store 5, r0
    store 6, r0
    store 7, r0
    store 8, r0
    store 9, r0
    store 10, r0
```

Assuming that the array elements get stored in stack slots 0–9, complete the code generation by giving a sequence of T08 instructions and labels which, when appended to the above, yields a correct translation of the source program. Do *not* perform runtime array bounds checking.

Question 8 [10 marks]

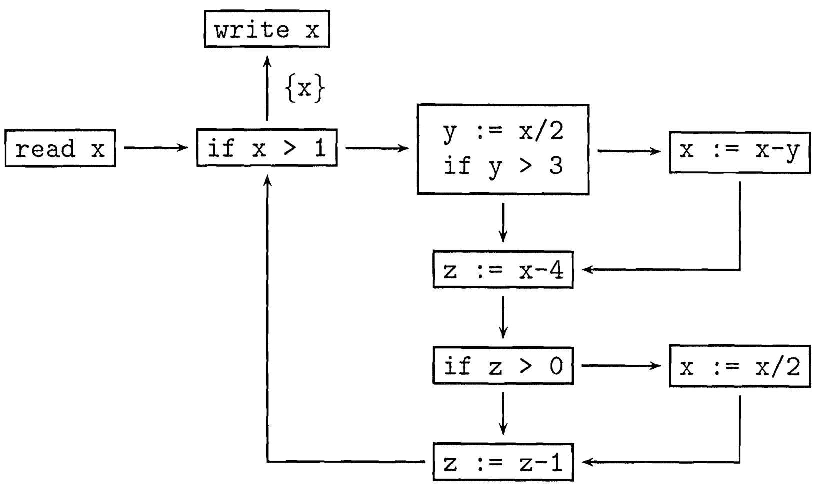
The following is a fragment of code written in T08 (from this year's project) except targets for jumps are instruction numbers, not symbolic labels:

```
83   load r0, 0
84   load r1, 1
85   cmp_ge_int r0, r0, r1
86   branch_on_false r0, 97
87   branch_uncond 92
88   load r0, 0
89   int_const r1, 1
90   sub_int r0, r0, r1
91   store 0, r0
92   load r0, 0
93   load r1, 1
94   cmp_gt_int r0, r0, r1
95   branch_on_true r0, 88
96   branch_uncond 99
97   load r0, 0
98   store 1, r0
99   load r0, 1
```

a. Draw the corresponding control-flow graph. For each basic block in the graph, just indicate its line numbers; there is no need to repeat the code. [4 marks.]

b. From the code above, give one example of a possible peephole optimization. [1 marks.]

Now consider the following control-flow graph G for a C-like language:



The program point just before '`write x`' has been annotated with liveness information, to indicate that only x is live at that point.

c. Copy the graph to your script book and propagate liveness information through the graph, to show which variables are live at which program points. [3 marks.]

d. Identify at least one optimisation of the code of G that is justified by liveness information. [2 marks.]

Question 9 [6 marks]

Consider the straight-line code

```
read a
read c
read e
b := a + c
d := b + c
a := 10
c := a - d
e := e + d
write d
write e
halt
```

a. Identify which variables are live at each program point and draw, based on the liveness information, the register interference graph for the variables. [3 marks.]

b. Given the following register interference graph, give a register allocation for all variables, using as few registers as possible. Use register names $R0$, $R1$, and so on. [3 marks.]