The University of Melbourne
Semester 2 Assessment 2007

Department of Computer Science and Software Engineering

433–361 Programming Language Implementation

Reading Time: 15 minutes

Writing Time: 3 hours

This paper has 7 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.

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 seven 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 [10 marks]

Consider the following two-rule Lex (or flex) specification, from which a C program `p` is generated:

```
%%
[a-c]*bb[a-c]*  { printf("y"); }
[a-c]+          { printf("n"); }
%%
```

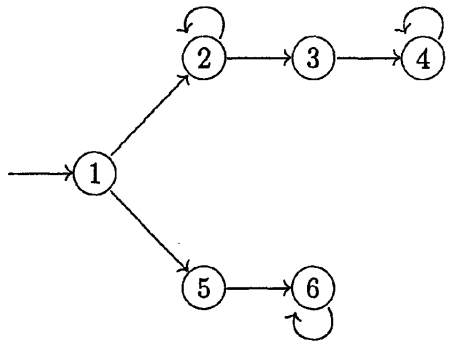
a. Given a three-line input file containing:

```
abbacababba
baccaba
yobbo
```

what will the output from `p` be? [1 mark.]

For the remaining questions, to simplify matters, assume the input alphabet is {a, b, c}.

b. In the process of converting the specification to a running program, Lex/flex produces an NFA of this shape:



Complete the NFA, by providing transitions and indicating which states are accept states. [2 marks.]

c. Using the subset construction method, convert to NFA into a DFA. Do not simplify the DFA at this stage—just follow the steps of the algorithm. [4 marks.]

d. Minimize the resulting DFA, keeping in mind that Lex/flex associates *actions* with accept states and uses these to distinguish states. [2 marks.]

e. With reference to the DFA construction, discuss which impact, if any, it will have if the programmer swaps the two rules in the Lex/flex specification. [1 mark.]

Question 2 [10 marks]

Consider the following context-free grammar, with start symbol S:

$$
\begin{array}{llll}
(r1) & S & \rightarrow & \mathtt{a}\ T\ U\ \mathtt{b} \\
(r2) & T & \rightarrow & \mathtt{c} \\
(r3) & T & \rightarrow & T\ \mathtt{d} \\
(r4) & U & \rightarrow & X\ Y
\end{array}
\qquad
\begin{array}{llll}
(r5) & X & \rightarrow & \epsilon \\
(r6) & X & \rightarrow & \mathtt{e} \\
(r7) & Y & \rightarrow & \epsilon \\
(r8) & Y & \rightarrow & \mathtt{f}
\end{array}
$$

a. Is the grammar ambiguous? Justify your answer. [1 mark.]

b. Calculate the *nullable*, *FIRST*, and *FOLLOW* sets for the grammar. [3 marks.]

c. Show that the grammar is not LL(1). (You may want to construct the look-ahead set for each rule, or you can give an informal explanation.) [3 marks.]

d. Modify the grammar *as little as possible* to turn it into an LL(1) grammar for the same language. [3 marks.]

Question 3 [15 marks]

Consider the (augmented) context-free grammar G_1 with start symbol S' and rules

$$
\begin{array}{llll}
(r0) & S' & \rightarrow & S \\
(r1) & S & \rightarrow & A\ \mathtt{;} \\
(r2) & A & \rightarrow & \mathtt{n} \\
(r3) & A & \rightarrow & A\ \mathtt{;}\ \mathtt{u} \\
(r4) & A & \rightarrow & B\ \mathtt{;}\ \mathtt{b} \\
(r5) & B & \rightarrow & A\ \mathtt{;}\ A
\end{array}
$$

(While you do not need this in order to solve the question, we observe that the generated language is reminiscent of languages for pocket calculators using reverse Polish notations, provided we interpret `n` and any number, `u` as any binary operator, and `b` as any binary operator.

a. Show the parse tree for `n ; n ; u ; b ; u ;`. [1 mark.]

b. Construct the LR(0) machine for G_1, that is, the finite-state machine having LR(0) sets-of-items as states and transitions that are determined by grammar symbols. [6 marks.]

c. Explain why G_1 is not SLR(1). [2 marks].

d. Is the following augmented context-free grammar G_2 (with start symbol A') equivalent, that is, does it generate the same language as G_1? Justify your answer. [1 mark.]

$$
\begin{array}{llll}
(r0) & A' & \rightarrow & A \\
(r1) & A & \rightarrow & \mathtt{n}\ \mathtt{;} \\
(r2) & A & \rightarrow & A\ \mathtt{u}\ \mathtt{;} \\
(r3) & A & \rightarrow & B\ \mathtt{b}\mathtt{;} \\
(r4) & B & \rightarrow & A\ A
\end{array}
$$

e. Construct the LR(0) machine for G_2, and use it to determine whether G_2 is SLR(1). [5 marks].

Question 4 [5 marks]

Consider the following (augmented) context-free grammar, with start symbol S:

$$\begin{array}{lrcl} (r0) & S & \rightarrow & A \\ (r1) & A & \rightarrow & \texttt{a} \\ (r2) & A & \rightarrow & A\ \texttt{[}\ A\ \texttt{]} \end{array}$$

A shift-reduce parser has been constructed for the language generated by S, with action and goto tables given below.

	a	[	]	$	A
0	shift 2				1
1		shift 3		accept	
2		reduce 1	reduce 1	reduce 1	
3	shift 2				4
4		shift 3	shift 5		
5		reduce 2	reduce 2	reduce 2	

Show in detail how the parser processes the input string

```
a[a[a]][a]$
```

including all the different states the parsing stack goes through. (As usual, we use '$' as an end-of-string marker.)

Question 5 [10 marks]

The following context-free grammar, with start symbol E, generates strings of the form s_1+s_2 where s_1 and s_2 are binary strings:

$$\begin{array}{lrcl} (r0) & E & \rightarrow & S + S \\ (r1) & S & \rightarrow & S\ B \\ (r2) & S & \rightarrow & B \\ (r3) & B & \rightarrow & \texttt{0} \\ (r4) & B & \rightarrow & \texttt{1} \end{array}$$

a. Add attributes and attribution rules to evaluate strings generated by E. For example, the string '`11101+001101`' should evaluate to 42. [6 marks.]

b. Discuss which changes would be needed in the attributed grammar if rule $r1$ instead was given as $S \rightarrow B\ S$. [4 marks.]

Question 6 [15 marks]

The abstract machine M07 has instruction set:

```
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, intconst           #     rN = integer
real_const        rN, realconst          #     rN = float
string_const      rN, stringconst        #     rN = string
add_int           rN, rI, rJ             #     rN = rI + rJ
add_real          rN, rI, rJ             #     rN = rI + rJ
sub_int           rN, rI, rJ             #     rN = rI - rJ
sub_real          rN, rI, rJ             #     rN = rI - rJ
mul_int           rN, rI, rJ             #     rN = rI * rJ
mul_real          rN, rI, rJ             #     rN = rI * rJ
div_int           rN, rI, rJ             #     rN = rI / rJ
div_real          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)
cmp_eq_real       rN, rI, rJ             #     rN = (rI == rJ)
cmp_ne_real       rN, rI, rJ             #     rN = (rI != rJ)
cmp_lt_real       rN, rI, rJ             #     rN = (rI <  rJ)
cmp_le_real       rN, rI, rJ             #     rN = (rI <= rJ)
int_to_real       rN, rI                 #     rN = (float) rI
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
call_builtin      builtin_function_name  # Builtin function call
return                                   # Procedure return
```

Consider the following program written in a procedural source language with variables of type `int` and `float`, and parameter passing by value and by reference:

```
proc main()
float x, result;
int n;
{   write "Float x: "; read x;
    write "Positive integer n: "; read n;
    power(x, n, result);
    write "x^n is: "; write result; write "\n"
}
```

```
proc power(val float x, val int n, ref float out)
float res;
{   if n = 1 then
        out := x
    else {
        power(x*x, n/2, res);
        if 2*(n/2) = n then
            out := res          # n was even
        else
            out := x * res      # n was odd
    }
}
```

Below is an initial segment of M07 code generated for this program.

```
        call proc_main
        halt
proc_main:
        push_stack_frame 3
        real_const r0, 0.000000
        store 0, r0
        real_const r0, 0.000000
        store 1, r0
        int_const r0, 0
        store 2, r0
        string_const r0, "Float x: "
        call_builtin print_string
        call_builtin read_real
        store 0, r0
        string_const r0, "Positive integer n: "
        call_builtin print_string
        call_builtin read_int
        store 2, r0
        load r0, 0
        load r1, 2
        load_address r2, 1
        call proc_power
        string_const r0, "x^n is: "
        call_builtin print_string
        load r0, 1
        call_builtin print_real
        string_const r0, "\n"
        call_builtin print_string
        pop_stack_frame 3
        return
proc_power:
```

Complete the code generation by providing a sequence of M07 instructions and labels which, when appended to the code above, will form a correct translation of the source program. You do not need to repeat the initial sequence up to label `proc_power`—just list the remaining M07 code.

Question 7 [10 marks]

The following is a fragment of code written in the language of the machine M07 introduced in the previous question:

```
        int_const r0, 0
        store 0, r0
        int_const r0, 0
        store 1, r0
        int_const r0, 0
        branch_on_false r0, label0
        int_const r0, 1
        store 0, r0
        branch_uncond label1
label0:
        int_const r0, 0
        store 0, r0
label1:
        int_const r0, 0
        branch_on_false r0, label2
        int_const r0, 1
        store 0, r0
        branch_uncond label3
        load r0, 1
        store 0, r0
label2:
        int_const r0, 0
        store 0, r0
label3:
label4:
        branch_uncond label5
label5:
        load r0, 0
        branch_on_false r0, label6
        load r0, 1
        store 1, r0
        int_const r0, 0
        store 0, r0
        branch_uncond label4
label6:
        int_const r0, 1
        int_const r1, 2
        add_int r0, r0, r1
        int_const r1, 3
        add_int r0, r0, r1
        call_builtin print_int
```

Identify opportunities for peephole optimization as well as other simplifications of the code, for example, based on constant propagation. Show the code that results after your optimizations.