The University of Melbourne
Department of Computer Science
Semester 2 Assessment 2003

433-361 Programming Language Implementation

Time Allowed: 3 Hours
Reading Time: 15 minutes

Authorized materials: Books and calculators are not permitted.

Instructions to Invigilators: One 14 page script. Exam paper may leave the room.

Instructions to students:
This exam counts for 75% of your final grade. There are 8 pages and 6 questions for a total of 75 marks. Attempt to answer all of the questions. Values are indicated for each question and subquestion — be careful to allocate your time according to the value of each question.

Question 1 [14 marks]

A small language for controlling a laser printer head has commands:

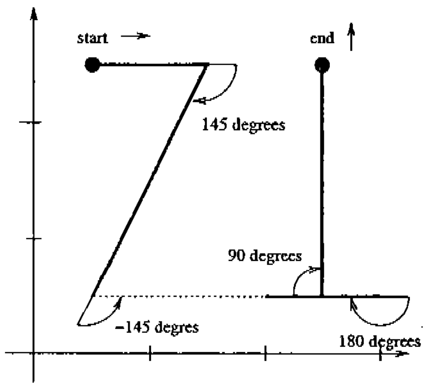
`down`	turn on laser drawing (pen down)
`up`	turn off laser drawing (pen up)
`forw` *d*	move laser forward distance *d*
`turn` *a*	change direction by (clockwise) angle *a*

Distances *d* are positive numbers with optional floating point, while angles are positive or negative integers of at most 3 digits. Sequences of tokens are separated by whitespace.

For example the sequence of instructions:

```
down forw 1 turn 145 forw 2.23 up
turn -145 forw 1.5 down turn 0 forw
1
turn 180 forw 0.5 turn 90 forw 2
```

draws the number 71 as shown to the left.



(a) Identify the lexemes that make up the tokens of the following input:

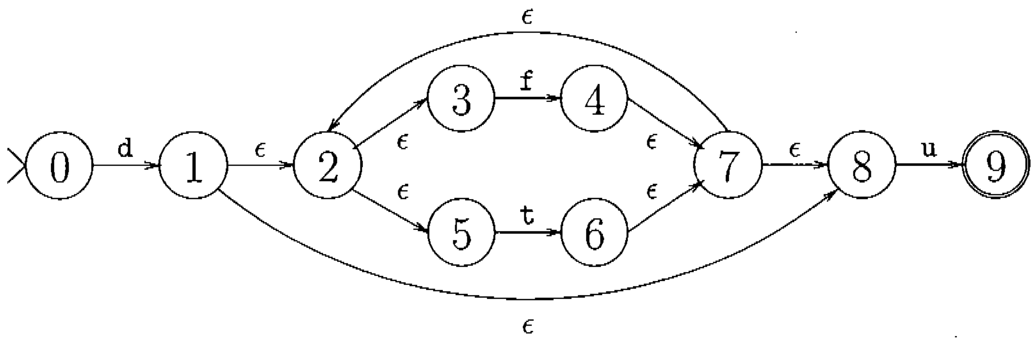
```
down forw 2.25 turn -45 forw 3 up
```

Give a list of lexeme, token and (some appropriate representation of the) token value. [1 mark].

(b) Give a LEX program that returns a sequence of tokens for an input consisting of a sequence of commands to the printer head. [3 marks].

(c) In a *valid* sequence of instructions, the pen is never placed down when it is already down, or up when it is up. The pen is considered to start up. Additionally the `forw` and `turn` instructions must alternate, that is there must be `turn` instructions in between any two `forw` instructions, and similarly there must be `forw` instructions in between any two `turn` instructions. Give an incomplete DFA that will recognize all *valid* sequences of instructions types, where instruction types are `d`, `u`, `f` and `t`. For example the *valid* sequence of instruction types corresponding to the sequence of instructions for the 71 example above is `dftfutfdtftftf`. [5 marks].

(d) Convert the following NFA to a DFA where the states are labelled by sets of states of the NFA. [4 marks].



(e) Give a regular expression which defines the same language as that accepted by the NFA in the previous question. [1 mark].

Question 2 [13 marks]

The following grammar G_1 is a simplified grammar for code statements.

	$S' \rightarrow S$
r1	$S \rightarrow$ `stmt`
r2	$S \rightarrow$ `ift` $S\ E$
r3	$S \rightarrow \{\ L\ \}$
r4	$L \rightarrow \epsilon$
r5	$L \rightarrow S\ L$
r6	$E \rightarrow \epsilon$
r7	$E \rightarrow$ `else` S

(a) Build the FIRST, and FOLLOW sets for the grammar G_1 and use these to build the lookahead sets for each rule above and determine whether the grammar is LL(1) or not. [4 marks].

(b) The *action* and *goto* tables for the LALR parser for the grammar G_1 above are given below:

	stmt	ift	{	}	else	\$	S	L	E
0	s2	s3	s4				1		
1						acc			
2	r1	r1	r1	r1	r1	r1			
3	s2	s3	s4				5		
4	s2	s3	s4	r4			7	11	
5	r6	r6	r6	r6	s9	r6			6
6	r2	r2	r2	r2	r2	r2			
7	s2	s3	s4	r4				8	
8				r5					
9	s2	s3	s4				10		
10	r7	r7	r7	r7	r7	r7			
11				s12					
12	r3	r3	r3	r3	r3	r3			

Show the execution of the parser for the input

`{ ift stmt else stmt }`

by giving the stack of (states + symbols) together with remaining input after each reduction step. Give the parse tree and rightmost derivation that results from the parsing. [3 marks].

(c) Give a compressed form of the action and goto table. [2 marks].

(d) *Warning HARD question, skip if unsure.* For each empty entry in the action table in rows 0, 3, 4, 7 and 9 add an error action. For each error action give a description of the error message, and actions to the parse stack and remaining input to continue parsing. [4 marks].

Question 3 [11 marks]

The following grammar G_2 defines a part of the grammar for a Prolog variant defining clauses and commands. The tokens `atom` stands for a Prolog atom, `g` stands for "`:-`", `c` stands for "`,`", `e` stands for "`.`", and q stands for "?".

$$S \rightarrow C$$

(r1)	$C \rightarrow$ `atom e`
(r2)	$C \rightarrow$ `atom g` B `e`
(r3)	$C \rightarrow$ g B q
(r4)	$B \rightarrow$ `atom c` B
(r5)	$B \rightarrow$ `atom`

(a) Build the LALR machine for G_2. Determine if G_2 is LALR. [9 marks].

(b) Without necessarily building it explain the differences between the LALR machine for G_2 and the LR(1) machine for G_2. [2 marks].

[please turn over ...]

Question 4 [13 marks]

The following grammar G_3 is part of a grammar for evaluating conditional expressions in a subset of C.

$$
\begin{aligned}
&S \rightarrow \texttt{if (} B \texttt{) } M\ A\\
&A \rightarrow id = E\ ;\\
&M \rightarrow \epsilon\\
&B \rightarrow E\\
&B \rightarrow E < E\\
&B \rightarrow E > E\\
&B \rightarrow E\ != E\\
&B \rightarrow E >= E\\
&B \rightarrow E <= E\\
&B \rightarrow E == E
\end{aligned}
$$

Write a YACC specification based on this grammar to output correct conditional evaluation code in a stack based language. Each expression E is an integer expression.

You can assume that the YACC code for E emits into the global code store code that places the integer value of expression E on the stack. Hence when we are about to do the reduction $B \rightarrow E_1 < E_2$ we know that the code will make the value of E_2 on top of the stack and value of E_1 just below it on the stack.

You can assume that the *id* token returns as its value the memory location where it will be stored.

The role of M is simply to have a marker non-terminal available. You may not modify the grammar and insert other marker non-terminals.

For example

```
if (4 + x > 3 - y) y = 7 + x;
```

might result in output code shown below (other outputs are also correct!).

```
push 4 # (!)  push 4 onto stack
dref 1 # (!)  push value of x (loc 1) on stack
plus   # (!)  result of E1=4+x on top of stack
push 3 # (!)  push 3 onto stack
dref 2 # (!)  push value of y (loc 2) on stack
sub    # (!)  value of E2=3-y on top of stack
psub
zero 1 # goto label 1 if zero
push 7 # (!)  push 7 onto stack
dref 1 # (!)  push x (loc 1) onto stack
plus   # (!)  value of 7+x on top of stack
```

```
asg  2 # assign to y (loc 2)
labl 1 # labl 1
```

You can make use of procedures

newlabel()	returns a new label.
emit_swap()	outputs `swap` instruction which swaps top two stack elements.
emit_sub()	outputs `sub` which subtracts top of stack from second top and replaces both by result.
emit_psub()	outputs `psub` which subtracts top of stack from second top and replaces both by result, unless the result is negative in which case it is replaced by 0.
emit_copy()	output `copy` which copies top of stack.
emit_push(i)	output `push` i which pushes the integer i on top of the stack.
emit_asg(r)	output `asg` r which pops top of stack and moves it into memory location r.
emit_labl(l)	output `labl` l which is a label instruction.
emit_goto(l)	output `goto` l which jumps to label l.
emit_zero(l)	output `zero` l which pops the top of stack and jumps to label l if it was 0.

Note that you are not responsible for emitting the expression calculation code (marked) (!) in the example above. [13 marks].

Question 5 [10 marks]

This question is about code generation. Consider the 3 address code fragment below:

```
(1) x := t - y
(2) y := x * x
(3) z := t - y
(4) u := y + z
(5) x := u * u
```

Assume that registers R0–R7 are available.

(a) Assuming that `x` is the only variable live at the end of the code, determine the set of variables live at each point before the instructions (1)–(5). [2 marks].

(b) Draw the register interference graph for this code fragment. [1 marks].

(c) Assuming a register interference graph was as shown in Figure 1, give an optimal assignment of registers to variables. Note that this graph is *not* the correct answer for the previous part. [2 marks].

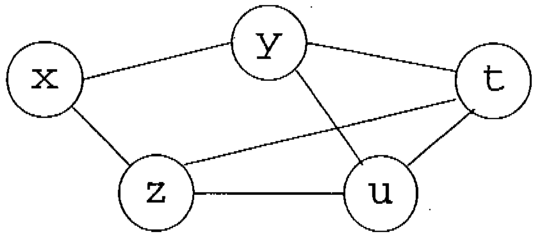


Figure 1: An example register interference graph.

(d) Assuming the register allocation strategy of placing x and z in R0, t and u in R1, and y in R2, give assembly code for this fragment. Assume that initially t is in R1 and y in R2. Note that this is *not* the correct answer to any previous part. [3 marks].

(e) Give the shortest code (in terms of words of storage) you can generate for the same fragment, assuming that initially t is in R1 and y in R2. [2 marks].

Question 6 [14 marks]

Consider the code in Figure 2, for counting the number of one bits in the binary representation of an integer x. The parse tree is given to the right. Each assignment statement is numbered.

(a) Give sample 3 address code that might be generated from this code, and define its basic blocks and flow graph. [2 marks].

(b) Generate the gen and kill sets for the total statement S. Recall the syntax directed definition is

$$\begin{array}{ll} S \rightarrow (d)\ id := E & S.gen := \{d\}; S.kill := D_{id} - \{d\} \\ S \rightarrow \texttt{if}\ (E)\ S_1 & S.gen := S_1.gen; S.kill := \emptyset \\ S \rightarrow \texttt{do}\ S_1\ \texttt{while}\ E & S.gen := S_1.gen; S.kill := S_1.kill \\ S \rightarrow S_1\ S_2 & S.gen := S_2.gen \cup (S_1.gen - S_2.kill); \\ & S.kill := S_2.kill \cup (S_1.kill - S_2.gen); \end{array}$$

where d is the number of the assignment statement, and D_{id} is the set of all assignment statement numbers with id on the left hand side. Note you may take shortcuts rather than laboriously generate the *gen* and *kill* sets for each statement S in the parse tree.[5 marks].

(c) Generate the in and out sets of reaching assignments for each numbered assignment

```
(1) y := 0
(2) x := 101
(3) z := 0
    do
(4)     t := x/2
        if (x != t*2)
            if (z != 0)
(5)             z := 1
(6)         y := y + 1
(7)     x := x/2
(8)     z := 0
    while x > 0
```

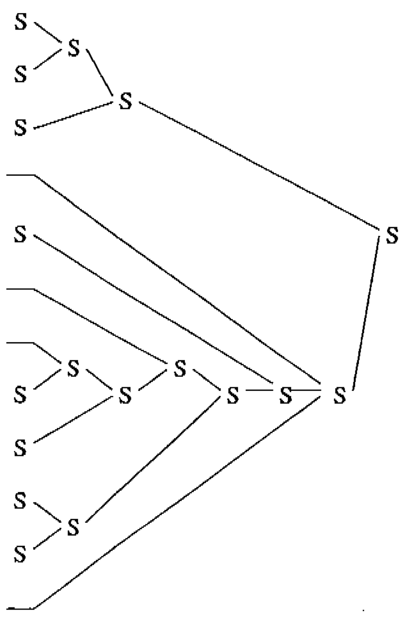


Figure 2: Code for bit counting.

statement. Recall that the syntax directed definition is:

$S \rightarrow (d)\ id := E$	$S.out := (S.in - D_{id}) \cup \{d\}$
$S \rightarrow$ `if` $(E)\ S_1$	$S_1.in := S.in; S.out := S_1.out$
$S \rightarrow$ `do` S_1 `while` E	$S_1.in := S.in \cup S_1.gen; S.out := S_1.out$
$S \rightarrow S_1\ S_2$	$S_1.in := S.in; S_2.in := S_1.out; S.out := S_2.out$

[6 marks].

(d) Give an example optimization to this code that might be made using reaching definitions information. Do not give an optimization which would require other information to be guaranteed to be correct.[1 marks].