The University of Melbourne
Semester 2 Assessment 2002

Department of Computer Science and Software Engineering

433–361 Programming Language Implementation

Reading Time: 15 minutes

Exam Duration: 3 hours

This paper has 8 pages, including this front page.

Identical Examination Papers: None

Common Content: None

Calculators: Not permitted

Authorised materials:
No books or notes are authorised. Calculators 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.

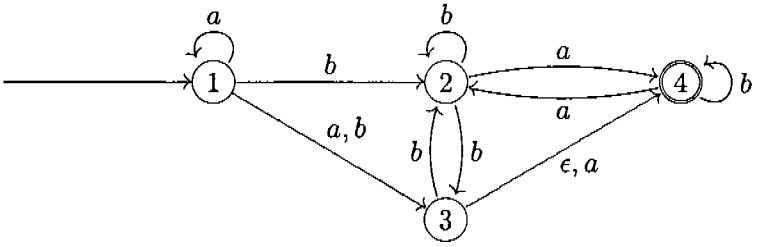
Instructions to Students:
This examination counts for 70% of the total assessment in the subject (30% being allocated to programming assignments). All questions should be attempted. Make sure that your answers are *readable*. Any unreadable parts will have to be considered wrong. For programming tasks, marks are allocated not only for correctness, but also for elegance and readability. There are 6 questions. Be careful to allocate your time according to the value of each question. The marks add to a total of 70.

This paper may be held by the Baillieu Library.

Question 1 [12 marks]

This question is about regular and context-free languages.

a. Use the subset construction method to turn the following nondeterministic finite-state automaton (NFA) into an equivalent deterministic automaton (DFA). [6 marks]



b. Give a regular expression for the language recognised by the NFA (or DFA). Make the regular expression as simple as you can. [2 marks]

c. Give a context-free grammar for the language $\{\mathtt{a}^{2n}\mathtt{b}\mathtt{c}^{n} \mid n \geq 0\}$, that is, the language consisting of strings `b`, `aabc`, `aaaabcc`, and so on. [2 marks]

d. Programming languages typically have certain "syntactic" aspects that cannot be captured by a context-free grammar. These aspects are often referred to as "static semantics". Give two examples of such aspects. [2 marks]

Question 2 [14 marks]

This question is about parsing. Consider this augmented grammar:

(0)	S'	$\rightarrow$	S				
(1)	S	$\rightarrow$	a	(3)	E	$\rightarrow$	S
(2)	S	$\rightarrow$	(E)	(4)	E	$\rightarrow$	E , S

a. Give the FIRST and FOLLOW sets for S', S, and E. [2 mark]

b. Is the grammar LL(1)? State why or why not. [2 marks]

c. Compute the sets of LR(0) items for the grammar. [3 marks]

d. Construct the SLR parsing tables for the grammar. [4 marks]

e. Show how the parser works by tracing the parsing stack's development while parsing input '`((a,a),a(`'. [3 marks]

Question 3 [8 marks]

This question is about the runtime stack in a typical implementation of a C-like language. Consider the following function definitions:

```
int square(int n) {
   return n * n;
}

int powerA(int x, int n) {
   if (n==0) return 1;
   return x * powerA(x, n-1);
}

int powerB(int x, int n) {
   int p;
   if (n==0) return 1;
   p = powerB(x, n/2);
   if (n == 2*(n/2)) return p*p;
   return p*p*x;
}

int powerC(int x, int n) {
   if (n==0) return 1;
   if (n == 2*(n/2)) return square(powerC(x, n/2));
   return square(powerC(x, n/2)) * x;
}
```

For each of the following function calls, sketch the resulting stack of activation records at the time the stack has reached its maximal height (for that call):

a. `powerA(2,5)` [2 marks]

b. `powerB(2,5)` [2 marks]

c. `powerC(2,5)` [2 marks]

The sketches should include local variables and formal parameters, and indicate which other kinds of information will be kept in activation records.

d. For the three versions of power discuss (1) relative runtime speed and (2) relative use of runtime stack space. [2 marks]

Question 4 [16 marks]

This question is about type inference using bison. The following grammar defines a rudimentary expression language:

$$
\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{number} \\
& | & \texttt{F} \\
& | & \texttt{T} \\
& | & \texttt{[ ]} \\
& | & \texttt{(}\ \mathit{exp}\ \texttt{+}\ \mathit{exp}\ \texttt{)} \\
& | & \texttt{(}\ \mathit{exp}\ \texttt{<}\ \mathit{exp}\ \texttt{)} \\
& | & \texttt{(}\ \mathit{exp}\ \texttt{:}\ \mathit{exp}\ \texttt{)} \\
& | & \texttt{head}\ \mathit{exp} \\
& | & \texttt{tail}\ \mathit{exp} \\
& | & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp}
\end{array}
$$

'F' and 'T' denote the Boolean values false and true, respectively. '`[]`' denotes the empty list, and '`:`' is the list constructor. So, for example, `(3:(4:[]))` denotes the list whose elements are 3 and 4. The expression '`head` E' denotes the first element of list E, and '`tail` E' denotes the remainder of the list. Here is the essential part of a Lex program for the language:

```
[ \t]    { ; }
[0-9]+   { return NUM; }
"+"      { return PLUS; }
"<"      { return LESS; }
"F"      { return FALSE; }
"T"      { return TRUE; }
"("      { return LPAR; }
")"      { return RPAR; }
"[]"     { return NIL; }
":"      { return CONS; }
"head"   { return HEAD; }
"tail"   { return TAIL; }
"if"     { return IF; }
"then"   { return THEN; }
"else"   { return ELSE; }
"\n"     { return NEW; }
```

The language is typed. There are three base types: `int`, `bool`, and `any`. There is one type constructor, '`list of ...`', so that we have infinitely many types: `list of int`, `list of bool`, `list of any`, `list of list of int`, `list of list of bool`, `list of list of any`, etc.

Expression	Type	Code
`14`	`int`	1
`F`	`bool`	2
`[]`	`list of any`	3
`tail (22:[])`	`list of int`	4
`([]:[])`	`list of list of any`	6
`head tail (42:(45:[]))`	`int`	1
`head ((if T then 44 else 45:[]):[])`	`list of int`	4
`if T then tail (((3<4):[]):[]) else ([]:[])`	`list of list of bool`	5
`(3+F)`	`Type error`	
`if (3+5) then T else T`	`Type error`	
`if T then 44 else []`	`Type error`	
`((3<4):(5:[]))`	`Type error`	
`head []`	`Type error`	

Table 1: Some expressions and their types

No expression has type `any`, but `[]` has type `list of any`. This type is compatible with all types, except `int` and `bool`. Hence `[]` is polymorphic. It can appear wherever an expression of type '`list of ...`' is expected. So, for example,

```
if (3<4) then [] else ((42:[]):[])
```

is well-typed, because we can give the 'then' branch's `[]` the type `list of list of int`, and then the two branches have the same type. Hence the whole expression is well-typed, having type `list of list of int`.

You should study the examples in Table 1 carefully. The "code" refers to a convenient way of encoding the types as numbers. We can use

0	for type error,
1	for `int`,
2	for `bool`,
3	for `list of any`, and
$n+3$	for `list of` T where $n > 0$ is the code for type T.

There are two cases where type compatibility is an issue. One is the conditional (already discussed) where the two branches must have compatible types. The other is list construction, $(e : es)$. If e has type T then the type of es must be compatible with `list of` T.

The following C function takes two type codes and returns 0 if the types are incompatible. Otherwise it returns the more specific of the two. Note that a type is polymorphic if and only if its code is positive and a multiple of 3.

```
int unify(int code1, int code2) {
    if (code1 == code2)
        return code1;
    if (code1 > 0 && (code1 % 3) == 0 && (code1 < code2))
        return code2;
    if (code2 > 0 && (code2 % 3) == 0 && (code2 < code1))
        return code1;
    return 0;
}
```

Using this function, and the tokens given in the Lex program above, write a bison program to infer types of expressions. Given a standard-input file with one expression per line, the resulting type inference tool should write (to standard output) the corresponding types, one per line (or "`Type error`" when appropriate). The rules could start like this:

```
exprs   : /* nothing */
        | exprs exp NEW                 { print_type($2); }
        | exprs NEW
        ;

exp     : NUM                           { $$ = 1; }
        | FALSE                         { $$ = 2; }
        | TRUE                          ...
```

for a suitable definition of `print_type` (also to be given as part of your answer). For example, given input

```
if T then (5+6) else []
tail (T:[])
```

the following should be output:

```
Type error
list of bool
```

Question 5 [10 marks]

This question is about intermediate code generation and "short-circuit" code. Consider these rules for syntax-directed code generation:

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

$S \rightarrow \mathbf{id} := E$
  $S.code := E.code \parallel gen(\mathbf{id}.place\ ':='\ E.place)$

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

$E \rightarrow \mathbf{id}$
  $E.place := \mathbf{id}.place$;
  $E.code := ''$

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

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

$B \rightarrow \mathbf{id}_1 < \mathbf{id}_2$
  $B.code := gen('\texttt{if}'\ \mathbf{id}_1.place\ '<'\ \mathbf{id}_2.place\ '\texttt{goto}'\ B.true) \parallel gen('\texttt{goto}'\ B.false)$

Show the *code* attribute for

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

Assume its *next* attribute is (the label) 42, and the next available label is 45.

Question 6 [10 marks]

This question is about register allocation and code generation. The task is to translate the following three-address code fragment to the assembly language used in the textbook (using `MOV`, `ADD`, `MUL`, and `SUB`):

```
0:
   a := b + e
1:
   e := a * a
2:
   a := d + e
3:
   b := b - c
4:
   a := a * b
5:
```

Assume that registers R0–R7 are available. At the end of the (assembly code corresponding to this) fragment, a should reside in register R0.

a. Assuming that a is the only variable which is live at program point 5, determine, for each program point, which variables are live at that point. [2 marks]

b. Draw the register interference graph for the code fragment. [3 marks]

c. Given the constraint on a, assign each variable one of the registers R0–R7, in such a way that you use as few registers as possible. (Beware that the heuristic for graph colouring suggested in the textbook may not give an optimal solution.) [2 marks]

d. Give assembly code for the fragment, obeying the register assignment determined in the previous question. [3 marks]

[end of exam]