UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Prof. R. Fateman

Fall, 2001

SUGGESTED SOLUTIONS CS 164 Final Examination: December 18, 2001, 8-11AM

Some of the problems had alternative solutions. We suggest one or occasionally alternative solutions. figures

1. [12 points] Consider the following language L1. L1 consists of sequences of three different symbols `< w >`. A sentence in L1 is either the single symbol w, or it begins with `<` followed by zero or more sentences and ends with `>`. For example, the following are sentences:

```
w
< >
< w w >
< w < > >
< < < > w > >
< < w < w > < > > w < w > w >
```

a. [4] Write a simple grammar for L1. Make sure it is NOT left recursive.
SOLUTION

```
S -> w | < L >
L -> S L | epsilon
```

b. [4] On the reverse of this page, write a recursive descent parser that accepts exactly the language L1.
SOLUTION

```
(defun s()(case (peek)(w (eat 'w)) (< (and (eat '<)(l)(eat '>))))))
(defun l()(case (peek)((w <)(and (s)(l)))))
;; many other versions possible.
```

c. [2] This grammar could describe (possibly nested) comments in a conventional programming language where `<` = "begin comment", `>` = "end comment", and `w` = "anything else" You cannot write a finite state machine that accepts exactly L1. Please explain why.
SOLUTION

```
Any FSM would fail to recognize sufficiently deeply nested
matching < .. < ...> ...> because  it has only finite memory.
```

d. [2] Consider another language, L2. L2 is the same as L1 except that comments are not nested, and sequences of comments allowed. For example `< w w > < w > < >` is in L2. The first 3 examples given for L1 are also in L2. Draw, in the space below, a finite state machine that accepts L2.

SOLUTION

```
you need about 4 states to recognize  w | (<w*>)* .
```

2. [12 points]

Here's an assembly language program printed exactly as it came out of our code generator. Write it in Tiger. That is, "unassemble" it.

```
code generation:
   0: args    0
   4: pushenv 1                        //      (f)
   8: fn
            0: args    1
            4: lvar    3       0       //      x
            8: pushi   1
           12: <
           16: jumpz   28
           20: pushi   1
           24: jump    80
           28: save    52
           32: lvar    3       0       //      x
           36: pushi   1
           40: -
           44: lvar    2       0       //      f
           48: callj   1
           52: save    76
           56: lvar    3       0       //      x
           60: pushi   2
           64: -
           68: lvar    2       0       //      f
           72: callj   1
           76: +
           80: return
  12: lset    2       0       //      f
  16: pop
  20: save    36
  24: pushi   5
  28: lvar    2       0       //      f
```

```
  32: callj   1
  36: popenv
  40: exit    0
```

The Tiger version:

SOLUTION

```
let function f(x:int):int = if (x<1) then 1 else f(x-1)+f(x-2) in f(5) end
```

3. [3 points] What does the op-code `fn` in the virtual machine do?

SOLUTION

```
1. creates function object; 2. puts current environment in the
environment of that object, 3. pushes that object [its address]
on the runtime stack.
```

4. [6 points] In the Tiger interpreter `tig-interp` there is a big case statement that looks at the operator of the expression being interpreted. If the expression is (AssignExp x y), the appropriate branch is

```
(AssignExp (set-var (elt x 1)(elt x 2) env))
```

What would be wrong with just implementing `set-var` in place, by

```
(AssignExp (setf (tig-interp (elt x 1))(tig-interp (elt x 2)) env))
```

Hints: use the words lvalue, rvalue, environment.

SOLUTION

```
There are lots of things wrong. tig-interp takes another
argument, env.  setf does not.  But that's just a start.
set-var must compute the l-value of (elt x 1), not the r-value.
then compute the r-value of (elt x 2) in environment env.
(This is almost done by the suggested substitute). 3. store
rvalue in env.  Lisp's setf doesn't do anything with the
tiger env.
```

5. [4 points] In this Scheme expression we use the `call/cc` construction:

```
(let ((x 200))
   (+ (call/cc (lambda (cc)
                  (let ((x 300))
                      (set! foo cc)
                      (+ 20 x))))
      x))
;; doesn't use the cc but saves it
```

a. [2] What does `(foo 500)` return and why?

SOLUTION

```
700 because 200 is the value for x at the call/cc and (+500 200) is 700
```

b. [2] What several pieces of information must be encoded in `foo`? (Consider your knowledge of the Tiger virtual machine and/or the interpreter).

SOLUTION

```
environment at the point of call/cc, continuation of the
computations at the call/cc
```

6. [6 points] Consider the following two programs.

```
/* 1 */ let type zz={a:int} var q:zz:=zz{a=3}in q.a end
/* 2 */ let type zz={a:int} var q:zz:=nil in q.a end
```

a. [2] The first typechecks correctly, and when run, it leaves 3 on the stack when it exits. The second program also passes the typechecker, but dies when you run it. Why?

SOLUTION

```
Bad access to q.a in 2nd program makes it fail. q is nil and has
no parts to access.  It passes the type checker because such accesses
were not checked.  In general such accesses cannot be checked until
runtime.
```

b. [4] A sequence of about 5 instructions could have been inserted into the code by a more careful compiler, to make the program exit, say with exit `code 1`, under this condition. Fix the listing below of program 2 to show what the compiler might generate. You may find that useful opcodes include `lvar, dup, jumpz, exit, =,` but not necessarily in that order.

Here is the code for /* 2 */

```
 0: args    0
 4: pushenv 1              //      (q)
 8: lvar    0       2      //      nil
12: lset    2       0      //      q
16: pop
20: lvar    2       0      //      q
24: iconst  0
28: mem
32: popenv
36: exit    0
```

SOLUTION

```
insert after 20:
dup
lvar 0 2 //nil
=
```

```
jumpz L2
exit 1 //illegal mem ref
L2:

// also renumber locations 20.
```

7. [12 points] Classify each of the following problems as lexical L , syntactic S, type checking T, run-time R or none of the above N. If the question needs to refer to a specific programming language to make sense, use Tiger. If you are unsure of your answer, you may write one of these letters and write a footnote explanation on the reverse. We will not read your explanation if your answer is right.

SOLUTION

```
T,S, STN, T,L,R,R,R,R,T,N,T
```

_____Record field reference to non-existent subfield
_____Unbalanced parentheses
_____Comma inside number (e.g. 1,000 in Tiger)
_____Wrong type of actual parameter
_____Missing end-of-comment delimiter
_____Division by zero
_____Array reference out of bounds
_____Initialization of array z:=intarray[-4] of 10
_____Reference through a nil pointer
_____Duplicate function name in scope
_____Type and Function with same name
_____Undefined function

8. [9 points] Consider the addition of an inheritance hierarchy to Tiger. In a type definition, right after the name of the type, an optional pair of parenthesis will enclose the parent type of that type. (If it is omitted, the class forms a root class and has no parent.) All fields of the parent are included as fields of this type.

For example, this now becomes legal Tiger.

```
let
   type t1 = {a: int}
   type t2(t1) = {b: int}
   var x := t2{a = 1, b = 2}
in   x.a + x.b end
```

(a) How will this affect the lexical analyzer?

SOLUTION

```
not at all
```

(b) How will this change the parser?
SOLUTION

```
the grammar must elaborate on type definitions to allow superclasses
```

(c) How does this change the type-checker?
SOLUTION

```
type t2(t1) must check that t1 is a record type. The record fields
for t2 must be set up to include the fields of t1, references to the
fields of var x of type t2 must be checked.  There should be no
duplicate names in t1 and t2.
```

(d) How does this change the run-time system?
SOLUTION

```
If you have done the work earlier, no changes necessary.
```

(e) What about the code-generator?
SOLUTION

```
If you have done the work earlier, no changes necessary.
```

9. [6 points] Reduce each of the following lambda expressions to normal form if possible or state that the expression has no normal form. It may help to explicitly show the reductions you make.

We use `(lambda(x) e)` for λx.e in math notation, and (lambda (x y) e) is λx.λy.e . That is, we allow multiple arguments.

a. `(y (lambda (y) (y (lambda(x) (x y)))))`
SOLUTION

```
no change
```

b. `((lambda (x y z)(x y z))(lambda(x) x)y)`
SOLUTION

```
(lambda(z)(y z))
```

c. `((lambda (x y) y) ((lambda(x) ( x x x))(lambda(z) ((z z) z))) z)`
SOLUTION

```
z
```

d. if **true** is denoted `(lambda(x y) x)`and **false** is `(lambda(x y) y)` then how would you express an "if expression" given like "if p, then t else e"?
SOLUTION

```
p t e
```

10. [8 points] The professor has changed the lexical analysis assignment and now requires that you extend your Tiger scanner (written in Lisp) to accept both decimal and trinary integers. Trinary numbers are expressed in base 3, using the digits 0, 1, and 2.

You must distinguish between the two number bases by insisting that decimal numbers must NOT begin with a digit "0" and that trinary numbers always begin with a zero. Thus counting in trinary goes like `01, 02, 010, 011, 012, 020`. The number `0121` (base 3) = 1+2*3+1*9 or `16` (base 10).

The scanner produces only one type, "iconst" for an integer, and the "value" associated with the token is a Lisp integer.

a. [4] On the reverse side of this page, draw a finite state machine that shows how such a scanner works for numbers. Include transitions to error states.

SOLUTION

```
4 states suffice including one error state.
```

b. [2,2] Louis Reasoner is confused and asks the professor: "How can we type the number zero in decimal, since it begins with a 0 and would thus be in trinary?

SOLUTION

```
0 trinary is the same value as 0 decimal so it doesn't matter.
```

And what is the base for the Lisp integer that is passed to the parser?"

SOLUTION

```
irrelevant. We have no way of knowing. We can PRINT the number in
any base that is convenient to us, but the data structure inside
the computer could be something else.
```

Show you are not as confused as Louis by answering these questions in complete sentences in the spaces above.

11. [9 points] Consider the following intermediate code:

```
    a := a + b
    c := a + b
    d := c
    d := a
    f := 3
    jump L1
    e := c
    d := a
L1: b := d
    jumpz x L3
L2: b := f
L3: d := d - 1
```

Put a box around each basic block. Draw all control flow edges. Draw a line through dead code, assuming that only b and d are live after this portion of code has completed execution.

SOLUTION

```
5 blocks, 5 edges,  dead code lines:
  c:=a+b,
  d:=c
  e:=c
  d:=a
```

12. [10 points] Mark the following statements as true (T) or false (F).

SOLUTION

```
F T F F T T T F - F
```

WARNING: points will be subtracted for incorrect answers, so don't guess.

_____ You can create a finite automaton to accept the language described by an arbitrary context free grammar.

_____ You can create a context free grammar to describe the language accepted by an arbitrary finite automaton.

_____ A regular expression is sufficient to describe the language of an arbitrary context-free grammar.

_____ Every LL(1) language can be described by a regular expression.

_____ Every regular expression can be translated into a finite automaton and vice versa.

_____ The table used to determine actions in an LALR(1) parser describes a finite-state machine.

_____ There are LALR(1) languages which require an arbitrary size stack to parse.

_____ Lambda calculas always uses call by name.

_____ All terminating evaluations of a lambda calculas expression terminate with the same value.

_____This sentence is false.

_____Tiger semantics can be completely described by a context-free grammar.