No: 509.xxx

THE CITY UNIVERSITY

Warning: Model Answers Included

LONDON

B.Eng. Computing
B.Eng. Software Engineering
B. Eng. Computing (DISC)
B.Sc. Business Computing Systems
Professional Pathway 3 - Software Engineering

PART III EXAMINATION

Language Processors

May 9th 2003 9.00-11.00

Answer THREE questions
All questions carry equal marks

1. (a) The following regular expression recognises certain strings consisting of the letters a, b and c:

$$a((ab)|(ac)) * c$$

Indicate which of these five strings are recognised by the above regular expression:

$aacc$, $abac$, ac, $abababacac$, $aabacc$

Also, show three more strings that are recognised by the above expression. Finally, show two more strings consisting of the letters a, b and c that are *not* recognised by the above regular expression. [30]

Answer: Yes, No, Yes, No, Yes. 3 marks each. Five further strings, 3 marks each.

(b) Most programming languages allow the same symbol to denote both the subtraction operator (in $x-y$) and the unary negation operator (in $-x$). Make clear the difficulty this causes for a parser-generator (for example, CUP) and explain how it may be overcome. [20]

Answer: Usually the precedence of the operator needs to be different in the two different contexts, eg highest for unary minus, and equal to plus etc for binary minus. Can be overcome by assigning a specific precedence to the rule for the unary case. So a 'pseudo-terminal' is introduced of appropriate precedence and the appropriate rule is annotated with it:

```
%left + -
%left * /
%left UM

...
exp : exp + exp
    | ...
    | - exp %prec UM
```

(c) Explain what it means for a context-free grammar to be ambiguous. Write down an ambiguous grammar and show why it is ambiguous. [20]

Answer: Two parse trees, two derivations, for same sentence. 5 marks. 5 marks for a grammar. 10 for the two trees/derivations.

(d) Consider the following Tiger function:

```
1 function f(a:string, b:int, c:int)=
2       (print_int(b+c);
3        let var c := "hi"
4            var a := b
5            var b := "hello"
6        in print(b); print_int(a)
7        end;
8        print_int(c); print_int(b);
9       )
```

Given an initial environment $\sigma_0 = \{\text{a} \rightarrow \text{int}, \text{b} \rightarrow \text{string}\}$, derive the type binding environments for the function at each use of an identifier and indicate where type lookups will occur. [30]

Answer:

0 σ_0 is starting environment
1 $\sigma_1 = \sigma_0 + \{a \rightarrow string, b \rightarrow int, c \rightarrow int\}$
2 lookup ids b, c in σ_1
3 $\sigma_2 = \sigma_1 + \{c \rightarrow string\}$ (overrides arg c)
4 lookup id b, then $\sigma_3 = \sigma_2 + \{a \rightarrow int\}$ (overrides arg a)
5 $\sigma_4 = \sigma_3 + \{b \rightarrow string\}$ (overrides arg b
6 lookup id b, then a in σ_4
6 discard σ_4, σ_3, σ_2 revert to σ_1
7 lookup c, b in σ_1
8 discard σ_1 revert to σ_0

2. The reference manual for a Tiger-like programming language contains the following definition for a kind of expression:

   The if-expression

   if exp_1 **then** exp_2 **else** exp_3

   evaluates the expression exp_1. If the result is non-zero the if-expression yields the result of evaluating exp_2; otherwise it yields the result of evaluating exp_3.

   (a) Write down a BNF concrete syntax for the if-expression. [10]

Answer:

Terminals in CAPS (trivial).

```
if-exp -> IF exp THEN exp ELSE exp
```

   (b) Sketch a possible abstract syntax for the if-expression. [10]

Answer:

```
IfExp Exp Exp Exp
(be flexible about how this is expressed - it might be Java)
```

   (c) Show how semantic actions in a grammar for a parser-generator such as CUP can be used to produce abstract syntax trees for the if-expression. [20]

Answer:

```
        |   IF:i exp:e1 THEN exp:e2 ELSE exp:e3
            {: RESULT = new Absyn.IfExp(ileft,e1,e2,e3); :}
```

   (d) Informally describe an appropriate typecheck for the if-expression. [20]

Answer:

Exp e1 must be an integer, and e2 and e3 must be the same type (which is the type of the whole expression), or both must produce no value, and then the whole expression produces no value.

   (e) Suppose a Tiger compiler translates all expressions and subexpressions into intermediate code (eg expression trees). Outline the intermediate code that might be generated in translation of the if-expression:

   `if a < b then c := a else c := b`

   You can assume that the expression tree for any variable `v` is simply `TEMP v`. [40]

Answer:

```
SEQ(SEQ(CJUMP(LT, TEMP a, TEMP b, L0, L1),
        SEQ(SEQ(SEQ(LABEL L0,
                    MOVE(TEMP c, TEMP a)),
                JUMP(NAME L2)),
            SEQ(SEQ(LABEL L1,
                    MOVE(TEMP c, TEMP b)),
                JUMP(NAME L2)))),
     LABEL L2)

(might be a longer translation where a < b is evaluated to
the 0 or 1 and then checked)
```

3. (a) Choose a programming language you know well and describe how run-time storage is organised and managed during program execution. Clearly associate any storage structures you mention with the implementation of particular language features. [25]

Answer:

Language features: procedure/method call and parameter passing, lexical scoping, dynamic storage allocation and deallocation, including object creation. Name stack, heap, why different. More marks for good exposition of example language (eg Java object creation, method call). Operation of stack and heap (with mention of garbage collection).

(b) What is a stack frame? Outline a typical layout for a stack frame and describe each element of a frame. Comment on how local variables, arguments and non-local variables are addressed by the code generated for a procedure or method. [25]

Answer:

A structure to store info local to a proc invocation. Args, static link, locals, return address, regs/temp space. More marks for mentioning code genned for a proc/func doing the pushing. Locals, args: offsets from SP or FP; non-locals: follow static links (more for description of static link).

(c) Explain why registers might be used for parameter passing and suggest situations where passing in registers is particularly appropriate. Outline situations where it is necessary for the code generated for a procedure or method to write registers to the stack. [30]

Answer:

Efficiency; particularly appropriate when leaf procs, interproc reg alloc, dead variables, reg windows (but...). Reg saves: when address is taken, when call-by-ref, when accessed by inner nesting, value too big, an array, convention of save for partic reg prior to call, spilling in exp evaluation, saving a reg window.

(d) Explain the difference between *caller-save* and *callee-save* registers. Why might caller-save registers sometimes not be saved? [20]

Answer:

Caller-save if code for caller of a func saves and restores the reg value around a func call. Callee save if code for a func does it. Might not be saved when analysis shows that a value of some var will not be needed following the call - put in caller-save reg but NOT save it ahead of call.

4. (a) Parser-generators such as CUP provide commands for specifying the *precedence* and *associativity* (or *non-associativity*) of operators in a programming language. Using examples, explain the difference between precedence and associativity and show how both are specified in a parser-generator you know (eg CUP). [40]

Answer:

Precedence specifies the order of evaluation of subexpressions according to the differing operators that combine the subexpressions. Usually we want * to evaluate before + in expressions containing both, so `9+5*2` is interpreted as `(9+(5*2))`. Allows us to omit brackets. Associativity determines the order of evaluation of subexpressions combined by operators which have the same precedence. Left-assocative operators like + group expressions from the left, so `9-5-2` is grouped `((9-5)-2)`. Some operators (eg exponentiation, assignment in C) may be right-associative. Some may be non-associative, meaning that more than two subexpressions may not be combined with them (eg a>b>c is not legal if > is non-associative).

(b) A version of the Tiger language allows logical expressions to include both the short-circuiting Boolean operators that do not evaluate the right-hand operand subexpression if the result is determined by the left-hand one (written & and |), and Boolean operators that always evaluate both of their operand subexpressions (written && and ||).

i. Give a grammar for these logical expressions omitting the semantic actions for the construction of the abstract syntax representation. Include specifications of precedence and associativity. Write your answer in the form of input to a parser-generator (eg CUP), and assume that the rest of the expression productions have been written. [10]

Answer:

```
precedence left OR, OROR;
precedence left AND, ANDAND;

exp ::=       ....
          |   exp AND exp
          |   exp OR exp
          |   exp ANDAND exp
          |   exp OROR exp
          ;
```

ii. The two short-circuiting boolean operators can be regarded as abbreviations for other expressions. They can be removed by a parser and replaced by an equivalent expression. The two operators that do not perform short-circuiting evaluation can also be treated in this way. Explain how, for all four cases. Logical expressions in Tiger produce the integer 1 for true, and 0 for false. [30]

Answer:

```
AND - if exp1 then exp2 else 0
OR  - if exp1 then 1 else e2
ANDAND - if exp1 then exp2 else if exp2 then 1 else 0
OROR   - if exp1 then if exp2 then 1 else 1 else if exp2 then 1 else 0
```

Some may replace the first sub-if with an SeqExp that evals exp2 and throws away the result and returns 1 (eg SeqExp (exp2,IntExp(1)))

iii. Show how your answer for the two short-circuiting operators in part (ii) can be expressed in semantic actions for a parser-generator producing an abstract syntax representation. You may assume that appropriate abstract syntax representation definitions are already written. [20]

Answer:

```
boolop_exp::= exp:a AND exp:b
                {: RESULT = new Absyn.IfExp(aleft, a, b,
                       new Absyn.IntExp(aleft,0)); :}
            |   exp:a OR exp:b
                {: RESULT = new Absyn.IfExp(aleft, a,
                       new Absyn.IntExp(aleft,1), b); :};
```