Final practice and solution

Exam Facts

Thursday March 22, 7-10pm **Terman Auditorium**

There is no alternate exam; everyone is expected to take the exam at the scheduled time. Local SITN students will come to campus for the exam, since that ensures you will be able to ask questions and receive any clarifications that might come up during the exam. SITN folks outside the Bay Area will take the exam at their local site. There is a link to a campus map up on the web page and some information about parking options for those of you coming from off-campus.

Format

The final will be an in-class 3-hour **open-note/closed-book** written exam. We won't ask you to wedge everything you've learned this quarter on one piece of paper, so you can bring class handouts, your own notes from lecture, and printouts of your programming projects to refer to during the exam. You may not use any books or a computer during the exam.

Given it is open-note, the exam will feature fewer short-answer questions whose answers can easily be looked up and more on application of the material, extension beyond the basics, synthesis of concepts, thought questions, and the like.

Material

The exam will cover the entire quarter, but will focus more on the second half (semantic analysis, code generation, and implementation issues). Material similar to the homework will figure prominently, although you are responsible for all topics presented in lecture and in the handouts. The sorts of things you might expect to see on the exam:

- (see midterm practice handout for list of topics covered prior to midterm)
- Type systems — named/structural type-equivalency, strong/weak typing, static/dynamic typing
- Scoping —variable lifetime and visibility, static/dynamic scoping
- Object-oriented issues— type compatibility, polymorphism, inheritance, static/dynamic binding, runtime implementation of objects and classes
- Intermediate representations —abstract syntax trees, TAC, intermediate code generation
- Target machine—processor architectures, memory hierarchies, data representation, instruction sets, executable format, assemblers, linkers, loaders
- Runtime issues— address space layout, function call protocols, parameter-passing, runtime stack, dynamic memory management, register use
- Optimization—basic blocks, control flow analysis, constant folding and propagation, algebraic simplifications, operator strength reduction, common sub-expression elimination, code motion, register allocation, instruction scheduling, peephole optimization
- Language design issues— tradeoffs between efficiency, safety, and expressiveness, CT versus RT decision-making

The rest of this handout is the final I gave last quarter. It gives you a good idea of expected problem formats, but remember all the material listed above is fair game, whether present on last quarter's exam or not. Solutions to all problems are given at the end of this handout, but we encourage you to not look at them until you have worked through the problems yourself.

1) The following input file was used to generate a lex scanner:

```
%%
a+b*a     { printf("1 %s\n", yytext); }
(ab)+c?   { printf("2 %s\n", yytext); }
aa        { printf("3 %s\n", yytext); }
(a|b)*c   { printf("4 %s\n", yytext); }
```

Show the output printed from the scanner when reading the input: `ababcbacaabaababaa`

2) Answer the following questions about LL parsing using the grammar below:

S –> Sx | zN
M –> (M) | Nz
N –> PMy | ∂
P –> ∂ | yPz

a) Compute the First and Follow sets for the non-terminal N:

	First	Follow
N		

b) Fill in the row for the non-terminal N in the LL(1) parse table. If there are conflicts, just write both entries into the table.

	x	y	z	(	)	$
N						

3) Answer the following questions about LR parsing, given the grammar below:

S' –> S
S –> AaB
A –> ABC | B
B –> Bb | bc
C –> c | ∂

a) Show the initial configurating set I_0 of the LR(1) parser for the above grammar.

b) Show the configurating set which is successor(I_0, B).

c) Based on what you have done so far, is the grammar LR(1)? If not, specify the lookahead symbol and type of conflict for each conflict(s).

4) The following input file was used to generate a yacc parser. Its associated scanner just returns each character read as an individual token.

```
%{
   int num = 0;
%}

%%

A  :  B { printf("A %d num %d\n", $1, num); }
   ;

B  : '(' { $$ = ++num; } B C ')' { num = $2; $$ = 2 * $3;}
   | C '+'
   ;

C  : /* empty */ { printf("C %d\n", num); $$ = 10;}
   ;
```

Show the output printed from the parser when reading the input `((+))`

5) Assuming you are dealing with a compiler for a Decaf-like language. For each of the following errors, circle one letter to indicate which phase would normally issue the message: (L) lexer, (P) parser, (SA) semantic analyzer, (CL) code generator/linker, (R) runtime.

(a) Else without matching if	Circle one:	L	P	SA	CL	R
(b) Array index out of bounds		L	P	SA	CL	R
(c) Unterminated comment		L	P	SA	CL	R
(d) Unknown method called on object		L	P	SA	CL	R
(e) No size argument in call to `NewArray`		L	P	SA	CL	R
(f) Method called on NULL object		L	P	SA	CL	R
(g) Function declared and used but not defined		L	P	SA	CL	R
(h) Wrong number of arguments in function call		L	P	SA	CL	R
(i) Divide by zero		L	P	SA	CL	R

6) You are given the definition of the following Decaf function:

```
void Binky(int[][] nums)
{
   int winky;

   nums[1][3] = 8;          // marked line
}
```

a) Draw a picture of the state of runtime memory right before execution of the marked line, assuming the call to Binky was made correctly. Distinguish between the stack and heap. We do not need to see register values, and you do not have to worry about temps.

b) Give the sequence of TAC instructions that would emitted from the compiler doing a straightforward translation of the marked line. Assume the compiler is not doing any optimization. Be sure to use only legal TAC instructions.

7) As part of semantic analysis in pp4, you verified that any return statement within a function definition was compatible with the declared return type. However, there is another error with function return that we did not consider which is a non-void function that fails to return a value at all. You are to consider two different approaches for adding this error detection.

a) Give a scheme that detects at compile-time if a non-void function will fail to return a value during some execution. Which phase of the compiler will be responsible for doing this check? What strategy will it use to recognize the error? Describe the impact this additional check has on compile and runtime performance. Does your strategy fail to work in some cases? If so, indicate under what situations it will report a false error on correct code or fail to report an error in faulty code.

b) Give a scheme that detects at run-time when a non-void function fails to return a value. Outline how you will generate TAC code to do the check at runtime and halt on error. Describe the impact this additional check has on compile and runtime performance. Does your strategy fail to work in some cases? If so, indicate under what situations it will report a false error on correct code or fail to report an error in faulty code.

8) We would like to add an exponentiation operator to Decaf that raises a base to a power. We will adopt Pascal's `**` syntax for this operator:

```
int result;

result = 2 ** 3;           // result = 8
result = result ** 2 + 2;  // result = 66
```

Exponentiation will only be defined for integer base and integer exponent. Some rules for exponentiation:

```
x ** 0  == 1
x ** 1  == x
x ** y  == x * x * x * ... * x     (y times)
x ** -y == 0   (this isn't quite true but it's close enough for ints)
```

Exponentiation binds tighter than ordinary binary arithmetic operators but lower than the unary operators. It associates right.

a) What changes will you need to make to the scanner? How will you distinguish the use of `*` for multiplication from the use for exponentiation?

b) What changes will you need to make to the parser? Be specific in terms of token types, precedence rules, and productions that are added, removed, or changed from the original Decaf parser.

c) What changes will you need to make to the semantic analyzer? Be specific in terms of what validity checks and error messages must be added, removed, or changed. It is preferred that you show this by indicating the actions associated with your productions from part b).

d) Show the steps you will use to generate the TAC instructions necessary to perform exponentiation. You will need to simulate the computation with repeated multiplication since there is no TAC exponentiation instruction. It is preferred that you show this by indicating the actions associated with your productions from part b). You can do this by either showing the sequence of emitted TAC instructions or by showing a list of calls to methods of the CodeGenerator class.

9) Provide brief answers to the following two short questions about object implementation.

a) C++ has a `dynamic_cast` operator that allows the programmer to safely downcast an object. The C++ statement `cow = dynamic_cast<Cow *>(animal)` will set the variable `cow` to `animal` if the animal is a Cow object or NULL otherwise. How could you implement such a verification of the runtime type of an object in Decaf?

b) Java has a notion of a *final* class, which is a class that cannot be subclassed. When a variable is declared to be of a class type that is final, the compiler is guaranteed the object will always be exactly of that type. Because there are no subclasses, no methods can be overridden. What advantage does this information provide the compiler in terms of optimization?

10) Optimize the basic block of TAC code below in a series of passes using the local optimization techniques described in class: (CP) constant propagation (CF) constant folding, (AS) algebraic simplification, (OSR) operator strength reduction, (CSE) common sub-expression elimination, (CPY) copy propagation, (DCE) dead code elimination. Do one optimization in each pass (CP, CSE, etc.), and apply the optimization to the entire block in that pass. Take as many passes as you need to get the code fully optimal, label each pass with the optimization being performed. You do not need to recopy the entire block on each pass, just the sections that change.

```
t0 = 0;
i = t0;
t2 = 4;
t3 = t2 * i;
t1 = a + t3;
t4 = 1;
*t1 = t4;
t5 = 1;
t6 = i + t5;
i = t6;
t8 = 4;
t9 = t8 * i;
t7 = a + t9;
t10 = 2;
*t7 = t10;
t11 = 1;
t13 = 4;
t14 = t13 * t11;
t12 = a + t14;
t16 = 4;
t17 = t16 * i;
t15 = a + t17;
t18 = *t15;
*(t12 + 4) = t18;
```

Solutions

1)

```
2 ababc
4 bac
1 aaba
2 abab
1 aa
```

2 a)

First(N) = { ∂ y z (}
Follow(N) = { x z $ }

b)

	x	y	z	(	)	$
N	N –> ∂	N –> PMy	N –> ∂, N –> PMy	N –> PMy		N –> ∂

3 a)

S' –> •S, $
S –> •AaB, $
A –> •ABC, a/b
A –> •B, a/b
B –> •Bb, a/b
B –> •bc, a/b

b)

A –> B•, a/b
B –> B•b, a/b

c) No, there is shift/reduce conflict on next symbol 'b'.

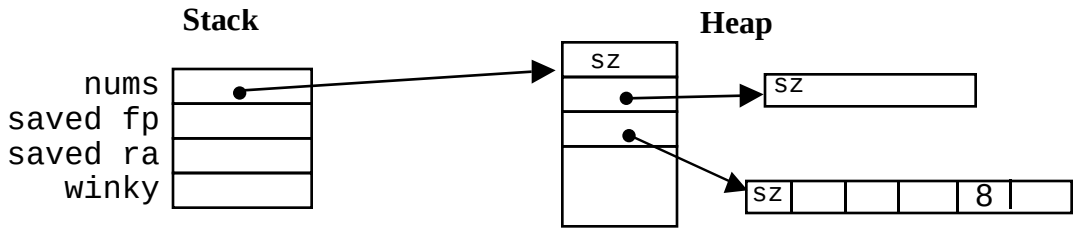
4)

```
C 2
C 2
C 2
A 40 num 1
```

5)

(a) P
(b) R
(c) L
(d) SA
(e) P
(f) R
(g) CL
(h) SA
(i) R

6 a)



b)

```
_t0 = 4;
_t1 = 1;
_t1 = _t1 + 1;
_t2 = _t0 * _t1;
_t3 = nums + _t2;
_t4 = *t3;
_t5 = 4;
_t6 = 3;
_t6 = _t6 + 1;
_t7 = _t5 * _t6;
_t8 = _t4 + _t7;
_t9 = 8;
*_t8 = _t9;
```

7 a) One strategy is to incorporate techniques from optimization: divide the function into basic blocks, build a control-flow graph, and trace backwards from the function exits to ensure that there is a return on each path. Another strategy is to use synthesized attributes on each statement to track a boolean value about whether this statement returns. A return statement does return, simple statements don't, while loops don't (because the loop may not be entered at all), an if/else returns only if both if and else return, compound returns if any statements within return, and so on. This type of detection would take place as part of semantic analysis and would slow down that phase a bit with the extra checking but would have no impact on runtime performance. Neither of these two strategies will fail to report an existing error, but both can be fooled into reporting spurious errors where the control flow or logical structure makes certain paths irrelevant, but the compiler can't figure that out, such as:

```
if (y * y >= 0) return 10;
while (true) return false;
```

b) Emit a TAC Halt as the last instruction before the EndFunction marker for all non-void functions. Thus any invocation that ends up falling off the end (i.e. didn't hit a return earlier), will execute the halt instruction at runtime. This will have minimal effect on compiler speed and basically no effect on runtime performance (tiny increase in code size, no additional runtime checks). It is also foolproof— if the function returns a value it won't get to the halt and any invocation that doesn't return will. However, a RT check has the disadvantage that it only examines those paths that are actually executed and doesn't consider what the other alternatives might have been (i.e. in unusual cases the function may not return a value, but only when that case is actually executed will that be detected).

8 a) Add a pattern to match the lexeme for our new token type:

```
<N>"**" { return T_Power; }
```

Because lex always tries to match the longest pattern, it will always match two stars in a row if there is more than one. Only a single star by itself will it be interpreted as multiplication.

b) Add the new token:

```
%token T_Power
```

Establish precedence and associativity for it (add line after the binary arithmetic operators and before the unary):

```
%right T_Power
```

Add the new expression:

```
Expr:   ...
          Expr T_Power Expr
```

c) In the action for `Expr T_Power Expr`, verify that both base and exponent expressions are of integer type, just as we did for the binary `%` operator. If either or both is non-integer, print an appropriate error message.

d) Here is sample TAC code (assuming base is currently in _t0 and exponent in _t1). This isn't exactly how what our compiler would generate (there would be some extra temporaries and more copying along the way), but we just needed to see the essential steps:

```
        zero = 0;
        one = 1;
        result = zero;
        _t3 = _t1 < zero;
        IfNZ _t3 Goto L2;       // if exp negative, jump to end
        result = one;            // init result to 1
L1:     IfZ _t1 Goto L2;         // loop til expt count is zero
        result = result * _t0;   // result *= base
        _t1 = _t1 – one;         // count--
        Goto L1;
L2:           // result holds _t0 ** _t1
```

9 a) Use the fact that each object already has a pointer to its vtable, which in effect already identifies the object's runtime type. Vtables are shared and accessible in the global space by their named label. In each vtable, include a pointer to parent vtable. Verifying an object is a "Cow" at runtime is a matter of examining its vtable pointer and potentially following the chain up the hierarchy looking for the Cow table.

b) If no methods can be overridden, all methods can be statically bound, saving the cost of indirection and offset into the vtable at RT. The vtable for the class may possibly be eliminated also (small space saving).

10) Instructions removed due to DCE are crossed out. Instructions that didn't change in the pass are grayed out.

	CP (+ DCE)	CF	CP (again)	AS
t0 = 0	~~t0 = 0~~			
i = t0	~~i = 0~~			
t2 = 4	~~t2 = 4~~			
t3 = t2 * i	t3 = 4 * 0	t3 = 0	~~t3 = 0~~	
t1 = a + t3	t1 = a + t3	t1 = a + t3	t1 = a + 0	t1 = a
t4 = 1	~~t4 = 1~~			
*t1 = t4	*t1 = 1	*t1 = 1	*t1 = 1	*t1 = 1
t5 = 1	~~t5 = 1~~			
t6 = i + t5	t6 = 0 + 1	t6 = 1	~~t6 = 1~~	
i = t6	i = t6	i = t6	~~i = 1~~	
t8 = 4	~~t8 = 4~~			
t9 = t8 * i	t9 = 4 * i	t9 = 4 * i	t9 = 4 * 1	t9 = 4
t7 = a + t9	t7 = a + t9	t7 = a + t9	t7 = a + t9	t7 = a + t9
t10 = 2	~~t10 = 2~~			
*t7 = t10	*t7 = 2	*t7 = 2	*t7 = 2	*t7 = 2
t11 = 1	~~t11 = 1~~			
t13 = 4	~~t13 = 4~~			
t14 = t13*t11	t14 = 4 * 1	t14 = 4	~~t14 = 4~~	
t12 = a + t14	t12 = a + t14	t12 = a + t14	t12 = a + 4	t12 = a + 4
t16 = 4	~~t16 = 4~~			
t17 = t16 * i	t17 = 4 * i	t17 = 4 * i	t17 = 4 * 1	t17 = 4
t15 = a + t17	t15 = a + t17	t15 = a + t17	t15 = a + t17	t15 = a + t17
t18 = *t15	t18 = *t15	t18 = *t15	t18 = *t15	t18 = *t15
*(t12+4)= t18	*(t12+4)= t18	*(t12+4)= t18	*(t12+4)= t18	*(t12+4)= t18

CP (again)	CSE	CPP	Final
t1 = a	t1 = a	~~t1 = a~~	
*t1 = 1	*t1 = 1	*a = 1	*a = 1
~~t9 = 4~~			
t7 = a + 4	t7 = a + 4	t7 = a + 4	t7 = a + 4
*t7 = 2	*t7 = 2	*t7 = 2	*t7 = 2
t12 = a + 4	~~t12 = a + 4~~		
~~t17 = 4~~			
t15 = a + 4	~~t15 = a + 4~~		
t18 = *t15	t18 = *t7	t18 = *t7	t18 = *t7
*(t12+4)= t18	*(t7+4)= t18	*(t7+4) = t18	*(t7+4) = t18