

CMSC 430 Practice Midterm 2 (Fall'02)

Use the following 3-address code and Java stack code instructions for answering code generation questions.

3-addr Instruction	Effect
load R1 x	R1 ← x
store x R1	x ← R1
add R1 R2 R3	R1 ← R2 + R3
sub R1 R2 R3	R1 ← R2 - R3
mult R1 R2 R3	R1 ← R2 * R3
neg R1 R2	R1 ← -(R2)

Java Stack Code	Effect
nop	none
ldc.int c	push constant c onto stack
iload index(x)	push local variable X onto stack
istore index(x)	pop stack, store in local variable X
iadd	pop 2 elems off stack, add, push
isub	pop 2 elems off stack, subtract, push
imult	pop 2 elems off stack, multiply, push
ineg	pop stack, negate, push
goto L	jump to handle L
ifeq L	pop stack, jump to handle L if zero
if.icmpeq L	pop 2 elems, jump to L if equal
if.icmpgt L	pop 2 elems, jump to L if 1st greater
dup	duplicate top of stack
pop	pop top of stack
swap	swap top two positions of stack

1. Attribute grammars and syntax-directed translation

- What are the main differences between attribute grammars and ad-hoc, syntax-directed translation?
- Name one advantage and disadvantage of each.
- Write a small syntax-directed translation that is not an attribute grammar.

2. Type expressions

Given the following C declarations:

```
char num(int * chunk);
void foo(char bar[4]);
```

- write the type expression for "num"
- write the type expression for "foo"

3. Syntax-directed translation and type checking

Consider the following grammar productions. Assume you have an attribute E.type which can be set to either INT, BOOL. Assume that the type of an expression is set to INT if an error is detected.

Assume you have a routine msg() similar to printf() that can be used to print error messages.

```
E → CONST      { E.type = ?? }
   | ID         { E.type = getType(ID.name); }
   | E1 + E2   { E.type = ?? }
   | E1 < E2   { E.type = ?? }
   | E1 == E2  { E.type = ?? }
   | ( E1 )     { E.type = ?? }
```

- Add rules to the attribute grammar to calculate E.type for each grammar production.
- Provide the parse tree for the expression $(5 < 2) == (4 + x)$ and show the calculation for E.type at each point.

4. Symbol table

Consider the following program in a lexically-scoped language such as C.

```
int x;
int main() { int y; ... }
int foo() { int z; { int x; } { int y; } }
int bar() { int z; { int y; } // HERE }
```

- What is an advantage of nested lexical scoping? Name a language that does not use nested lexical scoping, and potential problems encountered in that language.
- How can compilers organize symbol table information at compile time to handle different levels of scoping?
- Use your answer to construct the logical state of the symbol table(s) for the example when the compiler reaches the point marked **HERE**.

5. Run-time environment

- What is a frame (activation record)
- Where does the compiler allocate storage (memory) for variables at run time?
- What is garbage collection?
- What is a virtual machine?

6. Intermediate representations.

Consider the statement:

• $x := (a - b) * (c + d)$

- Translate each into an AST
- Translate each into 3-address code
- Translate each into Java stack code
- Which representation is closest to the input program? Why?

- (e) Which representation is closest to the hardware? Why?
- (f) Why might a compiler use multiple intermediate representations?

7. Code generation.

You are generating code for a Java stack machine. You are given the following grammar attributes and helper functions:

Attribute	Holds
astNode.code	list of instructions
Function	Effect
genInst(X)	create new instruction X returns handle to instruction
append(...)	concatenates lists of instructions

- (a) What grammar actions needed to generate code for a C-style DO ... WHILE loop in the following production? Remember that the DO WHILE loop executes at least one iteration, and repeats if the expression is true.

```
stmt → DO stmtList WHILE ( exp ) ;
{ : stmt.code = ??; ; }
```

- (b) What grammar actions needed to generate code for the (exclusive-or) expression in the following production? Remember that XOR is true if exactly *one* of the operands is true, and false otherwise. Your code should leave a 1 or 0 on the stack, depending on whether the expression is true or false. Note you can only use the Java instructions provided.

```
exp → exp1 XOR exp2
{ : exp.code = ??; ; }
```

8. Complex code generation.

What code must the compiler generate for the following:

```
int foo ( int x ) ;
int a[ 100 ] ;
...
x = foo( a[ 5 ] ) ;
```

- (a) Assuming foo() is call-by-value?
- (b) Assuming foo() is call-by-reference?

9. Improved code generation.

Frequently code generation and optimization are combined. Two examples are the Sethi-Ullman and DLS code generation algorithms for expression trees. Assume you are generating 3-address instructions. Consider the following expression:

- (a * (b + c)) - d

- (a) Build the abstract syntax tree for the expression and use the Sethi-Ullman labeling algorithm to mark the number of registers needed for each node of the tree (recall you are generating code for a load-store architecture, so each variable must be put into a register).
- (b) Generate code for the expression using the Sethi-Ullman algorithm. In case of ties, generate code for the left child first. When generating code, start with register R1 and increment (i.e., R1, R2, ...). When deciding which register to use, always use the lowest numbered register available. (I.e., always choose “add R1 R1 R2” instead of “add R2 R1 R2”).
- (c) How many stalls are present in the Sethi-Ullman code, assuming a 1-cycle latency for loads (e.g, they take 2 cycles to complete)?
- (d) How many registers are needed by the DLS algorithm to eliminate all stalls (relative to Sethi-Ullman), assuming a 1-instruction load delay?
- (e) Generate code for the expression using the DLS algorithm, using the minimal number of registers required to eliminate all stalls for loads.

10. Compiling object-oriented languages

Object-oriented programming languages such as C++ and Java use classes and inheritance to improve programmer productivity. Remember that inheritance allows objects of one class to be used in place of objects in a parent class.

- (a) What is prefixing (as applied to object-oriented programming languages)?
- (b) What code needs to be generated by the compiler to invoke a class method in an object with single-inheritance (with and without prefixing)?
- (c) Sketch the run-time state of the following object-oriented program:

```
class A extends Obj { int a;    f1(); f2(); }
class B extends A   { int b, c; f1(); f3(); }
class C extends A   { int c;    f4();      }
class D extends C   { int d, e; f4(); f5(); }

D x;
x.d = x.f1();
```

11. Compiling functional programming languages

- (a) What is the difference between a functional programming language and an imperative language such as C or Java?
- (b) How does this difference affect how programs are written in a functional programming language?
- (c) What is a closure?
- (d) How are closures implemented in the run-time environment?