

CMSC 430 Practice Problems 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	$R1 \leftarrow x$
store x R1	$x \leftarrow R1$
add R1 R2 R3	$R1 \leftarrow R2 + R3$
sub R1 R2 R3	$R1 \leftarrow R2 - R3$
mult R1 R2 R3	$R1 \leftarrow R2 * R3$
neg R1 R2	$R1 \leftarrow -(R2)$

Java Stack Code	Effect
nop	none
ldc.int c	push constant <i>c</i> onto stack
iload index(x)	push local variable <i>X</i> onto stack
istore index(x)	pop stack, store in local variable <i>X</i>
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 <i>L</i>
ifeq L	pop stack, jump to handle <i>L</i> if zero
if.icmpeq L	pop 2 elems, jump to <i>L</i> if equal
if.icmpgt L	pop 2 elems, jump to <i>L</i> 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 advantages of syntax-directed translation over context-free parsing?
- What are the differences between context-free and context-sensitive parsing?
- Rank the following techniques in terms of the languages (sets of strings) they can recognize: syntax-directed translation, attribute grammars, context-free grammars, context-sensitive grammars.
- In an attribute grammar, what are the differences between inherited and synthesized attributes?
- Write a small attribute grammar with one inherited and one synthesized attribute, labeling each.

2. Type expressions

Given the following C declarations:

```
int num(char *x, int chuk[4]);
typedef struct {
    int a, b;
} CELL;
CELL foo[100];
CELL *bar(int x, CELL y) { ... }
```

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

3. Syntax-directed translation

Consider the following grammar productions. Assume you have an attribute *E.odd* which can be set to either true, false, or unknown, and an attribute *CONST.val* which is the value of the constant.

$$\begin{array}{l}
 E \rightarrow \text{CONST} \quad \{ E.odd = ?? \} \\
 \quad | \text{ID} \quad \{ E.odd = \text{unknown} \} \\
 \quad | E_1 + E_2 \quad \{ E.odd = ?? \} \\
 \quad | E_1 - E_2 \quad \{ E.odd = ?? \} \\
 \quad | E_1 * E_2 \quad \{ E.odd = ?? \} \\
 \quad | (E_1) \quad \{ E.odd = ?? \} \\
 \quad | - E_1 \quad \{ E.odd = ?? \}
 \end{array}$$

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

4. Syntax-directed translation and type checking

Consider the following grammar, which generates expressions for a number of operators. Assume you have the attributes *E.min* and *E.max* which should be set to the minimum and maximum values for *E*, and an attribute *CONST.val* which is the value of the constant.

$$\begin{array}{l}
 E \rightarrow \text{CONST} \quad \{ E.min = ?? ; E.max = ?? \} \\
 \quad | \text{ID} \quad \{ E.min = \text{ID.min} ; E.max = \text{ID.max} \} \\
 \quad | E_1 + E_2 \quad \{ E.min = ?? ; E.max = ?? \} \\
 \quad | E_1 - E_2 \quad \{ E.min = ?? ; E.max = ?? \} \\
 \quad | E_1 * E_2 \quad \{ E.min = ?? ; E.max = ?? \} \\
 \quad | (E_1) \quad \{ E.min = ?? ; E.max = ?? \} \\
 \quad | - E_1 \quad \{ E.min = ?? ; E.max = ?? \}
 \end{array}$$

- Write the type-checking rules which calculates the range for each subexpression.
- Provide the parse tree for the expression $(5 + 2) - (4 * -x)$ and show the calculation for *E.min* and *E.max* at each point, assuming you know *x.min* = -3 and *x.max* = 10.

5. Syntax-directed translation and type checking

Consider the following grammar, which generates expressions formed by applying "+" to integer and floating point constants. When two integers are added, the result is integer, otherwise, it is a float.

$$\begin{array}{l}
 E \rightarrow E + T \mid T \\
 T \rightarrow \text{num} . \text{num} \mid \text{num}
 \end{array}$$

- (a) Give a syntax-directed definition to determine the type of each subexpression. Assign each symbol an attribute "type".
- (b) Provide the parse tree for the express
 $5 + 4 + 3.2 + 1$
 and show the computation E.type and T.type at each point.

6. Symbol table

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

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

- (a) What is nested lexical scoping?
- (b) How can symbol tables handle nested lexical scoping?
- (c) How does the compiler determine where each variable encountered during compilation is actually declared when handling nested lexical scoping?
- (d) Use your answer to construct the logical state of the symbol table(s) for the example when the compiler reaches the point marked HERE.

7. Run-time environment

- (a) What is a frame used for?
- (b) Name six items of of run-time information stored in a frame. For each item, identify whether its value is set before the procedure is called, during procedure execution, or right before procedure return.
- (c) Name one type additional type of information only stored in a frame for Java programs.
- (d) When can storage for a variable be allocated in a frame?
- (e) What advantage is obtained when allocating variables in a frame?
- (f) Name two advantages of managing memory allocation manually in the code.
- (g) Name two advantages of managing memory allocation automatically in the run-time system.
- (h) Name two methods of managing memory allocation automatically in the run-time system.

8. Intermediate representations.

Consider the statements:

- $x := a + (b * a)$
- $x := a - ((b + a) * c)$

- (a) Translate each into an AST

- (b) Translate each into 3-address code
- (c) Translate each into Java stack code
- (d) Which representation is the most compact? Why?
- (e) Which representation is easy to manipulate? Why?
- (f) Which representation is hard to manipulate? Why?
- (g) Which representation is closest to the input program? Why?

9. 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 IF statemen in the following production?
 $stmt \rightarrow IF (exp) stmtList ;$
 $\{ stmt.code = ??; \}$
- (b) What grammar actions needed to generate code for a C-style FOR loop in the following production?
 $stmt \rightarrow FOR (stmt ; exp ; stmt) stmt ;$
 $\{ stmt.code = ??; \}$
- (c) Write grammar actions needed to generate control code for an AND expression in the following production, using numerical value representation of booleans. Use *short circuiting*.
 $exp \rightarrow exp_1 AND exp_2$
 $\{ exp.code = ??; \}$
- (d) Write grammar actions needed to generate control code for an NOR expression in the following production, using numerical value representation of booleans. Use *short circuiting*.
 $exp \rightarrow exp_1 NOR exp_2$
 $\{ exp.code = ??; \}$
- (e) Write grammar actions needed to generate control code for an \geq (GEQ) expression in the following production, using numerical value representation of booleans.
 $exp \rightarrow exp_1 GEQ exp_2$
 $\{ exp.code = ??; \}$

10. Complex code generation.

- (a) Name two issues and solutions to generating code for function calls in C.
- (b) Name two issues and solutions to generating code for array references in C.

- (c) What code must the compiler generate for the code
`int i, a[100] ;`
`...`
`a[i + 5] = 4 ;`
- (d) What code must the compiler generate for the code
`int foo (int x) ;`
`...`
`x = foo(i + 2) ;`
- Assuming `foo()` is call-by-value?
 - Assuming `foo()` is call-by-reference?

11. 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:

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

- 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).
- 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”).
- 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)?
- How many registers are needed by the DLS algorithm to eliminate all stalls (relative to Sethi-Ullman), assuming a 1-instruction load delay?
- Generate code for the expression using the DLS algorithm, using the minimal number of registers required to eliminate all stalls for loads.

12. Compiling object-oriented languages

Object-oriented programming languages such as C++ and Java use classes and inheritance to improve programmer productivity.

- What is inheritance?
- How is inheritance implemented in the run-time environment?
- What code needs to be generated by the compiler to access a field in an object with single-inheritance (with and without prefixing)?

- What is multiple inheritance?
- How does multiple inheritance affect prefixing?
- 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 B   { int d;    f4();      }
class D extends A   { int d, e; f4(); f5(); }
```

```
C x;
x.d = x.f4();
```

13. Compiling functional programming languages

- Name two distinguishing features of functional programming languages.
- Sketch how each feature is implemented using compiler and run-time support.
- Describe why each feature may be desirable.
- Describe why each feature may lead to inefficient program execution.
- Describe one compiler strategy for improving performance for each feature.