

CMSC 430 (Spring 2009)

Practice Problems 4 Solutions

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 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. Run-time environment

- (a) What is a frame used for?
A frame (activation record) stores run-time information needed to maintain the environment of a function during program execution
- (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.
- *parameters - set before invocation*
 - *return value - set right before procedure return*
 - *return address - set before invocation*
 - *pointer to frame of enclosing function (access link) - set before invocation*
 - *pointer to frame of calling function - set before invocation*
 - *local variables - set during procedure execution*
- (c) Name one type additional type of information only stored in a frame for Java programs.
Local stack (for stack operations)
- (d) When can storage for a variable be allocated in a frame?
When the variable has fixed size and is no longer accessible after procedure return

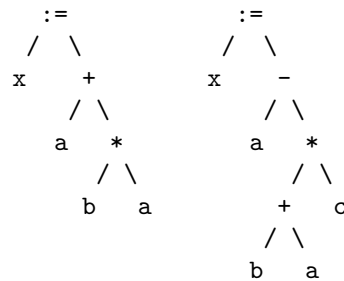
- (e) What advantage is obtained when allocating variables in a frame?
Entire frame may be freed at together at once when returning from procedure
- (f) Name two advantages of managing memory allocation manually in the code.
Most efficient (least wasted memory, memory freed at earliest opportunity), little run-time overhead
- (g) Name two advantages of managing memory allocation automatically in the run-time system.
Less programmer effort, possibility of premature freeing of memory greatly reduced
- (h) Name two methods of managing memory allocation automatically in the run-time system.
Reference counting keeps track of number of possible references, freeing when count reaches zero. Mark-and-scan looks for accessible memory during program execution and frees inaccessible memory.

2. 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

load R1 a	load R1 a
load R2 b	load R2 b
load R3 a	load R3 c
mult R4 R2 R3	add R4 R2 R1
add R5 R1 R4	mult R5 R4 R3
store x R5	sub R6 R1 R5
	store x R6

- (c) Translate each into Java stack code

iload index(a)	iload index(a)
iload index(b)	iload index(b)
iload index(a)	iload index(a)
imult	iadd
iadd	iload index(c)
istore index(x)	imult
	isub
	istore index(x)

- (d) Which representation is the most compact? Why?
Java byte code is the most compact, because it can take advantage of the implicit stack to avoid naming its operands for many instructions.
- (e) Which representation is easy to manipulate? Why?
ASTs are most easy to manipulate for high-level code, 3-address code is easy to manipulate for low-level code
- (f) Which representation is hard to manipulate? Why?
Java byte code is hard to manipulate because stack operations are more difficult to reorder
- (g) Which representation is closest to the input program? Why?
ASTs are closest to the input program because it maintains the original grammatical structure

3. 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 statement in the following production?

```
stmt → IF ( exp ) stmtList ;
      { stmt.code = ??; }
```

Answer:

```
stmt := IF ( exp ) stmtList ;
{
  Handle h1 = genInst( NOP );
  stmt.code = append (
    exp.code,
    genInst ( IFEQ( h1 ) ),
    stmtList.code,
    h1
  )
}
```

- (b) What grammar actions needed to generate code for a C-style FOR loop in the following production?

```
stmt → FOR ( stmt ; exp ; stmt ) stmt ;
      { stmt.code = ??; }
```

Answer:

```
stmt := FOR ( stmt1 ; exp ; stmt2 ) stmt3 ;
{
  Handle h1 = genInst( NOP );
  Handle h2 = genInst( NOP );
  stmt.code = append (
    stmt1.code,
    h1,
```

```
exp.code,
genInst ( IFEQ( h2 ) ),
stmt3.code,
stmt2.code,
genInst ( GOTO( h1 ) ),
h2
  );
}
```

- (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 → exp1 AND exp2
     { exp.code = ??; }
```

Answer:

```
exp := exp1 AND exp2
{
  Handle h1 = genInst( NOP );
  exp.code = append (
    exp1.code,
    dup,
    genInst ( IFEQ( h1 ) ),
    pop,
    exp2.code,
    h1
  );
}
```

- (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 → exp1 NOR exp2
     { exp.code = ??; }
```

Answer:

```
exp := exp1 NOR exp2
{
  Handle h1 = genInst( NOP );
  Handle h2 = genInst( NOP );
  Handle h3 = genInst( NOP );
  Handle h4 = genInst( NOP );
  exp.code = append (
    exp1.code,
    genInst ( IFEQ( h1 ) ),
    genInst ( GOTO( h2 ) ),
    h1,
    exp2.code,
    genInst ( IFEQ( h3 ) ),
    h2,
    genInst ( LDC_INT( 0 ) ),
    genInst ( GOTO( h4 ) ),
    h3,
    genInst ( LDC_INT( 1 ) ),
    h4
  );
}
```

- (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 → exp1 GEQ exp2
      { exp.code = ??; }
```

Answer:

```
exp := exp1 >= exp2
  {
    Handle h1 = genInst( NOP );
    Handle h2 = genInst( NOP );
    exp.code = append (
      exp1.code,
      exp2.code,
      genInst ( IF_ICMPEQ( h1 ) ),
      exp1.code,
      exp2.code,
      genInst ( IF_ICMPGT( h1 ) ),
      genInst ( LDC_INT( 0 ) ),
      genInst ( GOTO( h2 ) ),
      h1,
      genInst ( LDC_INT( 1 ) ),
      h2
    );
  }
```

- (d) What code must the compiler generate for the code
- ```
int foo (int x);
...
x = foo(i + 2);
```

- i. Assuming foo() is call-by-value?  
*If foo() is call-by-value, calculate i+2 and use value as argument*
  - ii. Assuming foo() is call-by-reference?  
*If foo() is call-by-reference, calculate i+2, store in temporary variable, then pass address of temp var as argument*
- 

#### 4. Complex code generation.

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

*Function call could occur in an expression (e.g., 4 + foo()), need to calculate value of function and use return value in expression*

*Function argument could be an expression (e.g., foo(4+x)), need to evaluate expression first and pass as argument*

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

*Could have multidimensional arrays, need to convert indices into memory location by flattening the array in either row or column major order*

*Could have arrays or array elements as arguments (e.g., foo(a[]), foo(a[4])), need to decide whether to pass value or address depending on whether call-by-reference or call-by-value*

- (c) What code must the compiler generate for the code
- ```
int i, a[ 100 ] ;
...
a[ i + 5 ] = 4 ;
```

...

For a[i+5] = 4, compiler must generate code to calculate i+5, (adjusting by first array index, usually 0 or 1) multiply result by size of elements of array, then add to base address of a to find address of array element