Compilers

Fall 2009

Intermediate Code Generation

Sample Exercises and Solutions

Prof. Pedro Diniz

Departamento de Engenharia Informática (DEI)
Instituto Superior Técnico / Tagus Park
Av. Prof. Dr. Cavaco Silva
2780-990 Porto Salvo
Portugal

Problem 1: Intermediate Code Generation for Three-Address Instructions – Expressions

For the assignment x = (a+c) * (a+b) perform the following two translations:

a. Using the SDT scheme described in class and without any effort to minimize the number of registers
b. Using the SDT scheme described in class but now using the slightly modified `newtemp` functions that attempts to reuse temporary variable as much as possible.
c. Is the number of minimal temporaries for this statement to number found by the approach in b) above? Justify.

Answer:
Assuming a post-order traversal of the parse tree we get the following three-address code.

a.

```
t1 = a + b
t2 = a + c
t3 = t1 * t2
x = t3
```

b.

```
t1 = a + b
t2 = a + c
t1 = t1 * t2
x = t1
```

c. Yes, this is the best one can do as you need to hold the values of (a+c) and (a+b) in two registers at least before executing the multiplication. You cannot apply any of the distributive and associative arithmetic properties and thus need to have a second register at least.

Problem 2: Intermediate Code Generation for 3-Address Instructions – Boolean Expressions
For the boolean expression **a < b AND c > d** perform the following two translations:

a. Using the SDT scheme described in class
b. Using the short-circuit evaluation scheme described in class.

State your assumptions as to the starting addresses are.

Answer:

```
a
.
    00: if a < b goto 03
    01: t1 = 0
    02: goto 04
    03: t1 = 1
    04: if c > d goto 07
    05: t2 = 0
    06: goto 08
    07: t2 = 1
    08: t3 = t1 and t2

b.
    00: if a < b goto 02
    01: goto Lfalse
    02: if c < d goto Ltrue
    03: goto Lfalse
```

Problem 3: Intermediate Code Generation with Frame-Pointer

In class we saw the actions for a Syntax-Directed Translation scheme to generated code using semantic actions for a simple arithmetic expression. In this exercise you are asked to extend the semantic actions to include the explicit loading of a given scalar variable from a predefined offset of a *frame-pointer* or *FP*. The frame pointer is a specific register in the hardware processor that holds the virtual address of a section of the virtual address space where variables local to a given procedure are to be stored. As such reading the value of a scalar variable, say `a` amount to generating the sequence of three-address instructions:

```
t1 = FP + offset_a;
t2 = *t1;
```

where `offset_a` is a compiler predefined constant, say 4, and the second instruction performs an indirect loading from memory based on the address calculated in the temporary variable t1.

Assuming the frame-pointer register is correctly set up when you initiate the generation of intermediate code show the translation of the sequence of statements below indicating the following:

```
x = a * b - a * 2;
if(x < 10) then
  y = 0;
```

a) The augmented semantic actions for loading the value of a given scalar variable.
b) Using the approach for the minimization of temporary variables indicate how many temporaries would you require for executing these statements assuming the value of x is not needed after the evaluation of the conditional statement. Explain.

Answer:

a) The augmented semantic action will have instead of direclt using the symbol of a scalar variable v as its address, it would insert two instructions as given by the semantic action below for a read operations

```
tn = newtemp();
emit(tn = FP + offsset_v);
tm = newtemp();
emit(tm = *tn);
```

For a write operation the semantic action would have to be modified as

```
tn = newtemp();
emit(tn = FP + offsset_v);
emit(*tn = tx);
```

where tx would be the temporary holding the value to be written to the position of the scalar variable.

Given these modifications the generated code it is depiected below where we have made no effort of reusing temporary variables and have thus assumed an infinite number of such temporaries.

```
        t1 = FP + offset_a        ; computing the address of the scalar variable a
        t2 = *t1                  ; reading the value of the scalar variable a
        t3 = t2 * 2               ; computing subexpression (a*2)
        t4 = FP + offset_b        ; computing the address of the scalar variable b
        t5 = *t4                  ; reading the value of the scalar variable b
        t6 = t2 * t5              ; computing (a * b) as t2 still has the value of a
        t7 = t6 – t3              ; assigning (a*b) – (a*2) to temporary variable
        t8 = FP + offset_x        ; computing the address of the scalar variable x
        *t8 = t7                  ; writing value of expression back to location of scalar variable x
        if(t7 < 10) goto L1       ; condition (x < 10) evaluation
        goto L2                   ;
L1:     t9 = FP + offset_y        ; computing the address of scalar variable y
        *t9 = 0                   ; writing value of y as zero
        goto L2                   ;
L2:
```

b) Using the counter-based assignment of temporaries we arrive at the code below, which uses only three temporary variables. Note in this case a write via an address is like a read operation of the address and thus will lead to a decrease in the value of the counter used during the code generation.

```
        t1 = FP + offset_a        ; computing address of scalar variable a
        t1 = *t1                  ;
        t2 = t1 * 2               ; t2 has the value of expression (a*2)
        t3 = FP + offset_b        ; computing address of scalar variable b
        t3 = *t3                  ; t3 has the value of scalar variable b
        t3 = t1 * t3              ; t3 has the expression (a*b)
        t2 = t3 – t2              ;
        t3 = FP + offset_x        ;
        *t3 = t2                  ; writing value of expression inx
        if(t2 < 10) goto L1       ;
        goto L2                   ;
L1:     t1 = FP + offset_y        ; computing address of scalar variable x
        *t1 = 0                   ;
        goto L2                   ;
L2:
```

Problem 4: Back-patching of Loop Constructs

In class we saw the actions for a Syntax-Directed Translation scheme to generated code using the back-patching technique for a *while* loop construct. In this exercise you will develop a similar scheme for the *repeat-until* construct using the production:

S → **repeat** S **until** E;

Do not forget to show the augmented production with the marker non-terminal symbols, M and possibly N along with the corresponding rules for the additional symbols. Argue for the correctness of your solution without necessarily having to show an example.

Answer:

```
S → repeat M1 S1 until M2 E          {
                                          backpatch(S1.nextlist, M2.quad);
                                        backpatch(E.falselist,M1.quad);
                                         S.nextlist := E.truelist;
                                     }
```

The first back-patching command fills in the places where the control in S_1 is transferred to the next iteration, that is, to the evaluation of the conditional E that is given by the M_2.quad value. The second back-patching command links the places where the evaluation of E is false to M_1.quad, that is to the top of the loop. Finally the overall construct next list simply consists of the location where the control for which the conditional E corresponds to the evaluation E to true. Notice that there is no need to generate an extract goto statement as we had in the while loop since the control can fall through the evaluation of the conditional statement.

Problem 5: Intermediate Code Generation

Consider the sample source code depicted below:

```
i = 0;
while (i < 10) {
  b[i] = a;
  i = i + 1;
}
```

For this code the register allocation has determined that variables `i` are allocated the physical register `r0` and the scalar variable `a` will use register `r1`. All other variables are not allocated registers but rather stay in the Activation Record throughout the computation.

a) Present a three-address instruction code for this segment of code under the assumption that all variables are live at the end of the block except variable `i`. Use the address calculation with the Frame-Pointer register (FP) where the offsets have been computed already and have symbolic names such as offset_a and offset_i.

b) Assuming variable `a` is volatile, i.e., its value can changed as a result of an external event, could you use it in a register? Why or why not?

Answers:

a) One possible translation of the code above is presented below with appropriate comments.

```
1:       r0 = 0
2:       r1 = 0
3: L1:   if r0 < 10 goto L2
4:       goto L3
5: L2:   t2 = r0 + 1
6:       t3 = a[t2]
7:       a[r0] = t3
8:       t4 = a[r0]
9:       r1 = r1 + t4
10:      t5 = r0 + 1
11:      r0 = t5
12:      goto L1
13: L3: t2 = FP + offset_i
14:      *t2 = r0 // update value of i in AR
15:      t2 = FP + offset_sum
16:      *t2 = r1 // update value of sum in AR
17:      return
```

b) Lines 8 through 9 can be replaced by the sequence below

```
8:       t4 = a[r0]
9:       putparam t4   // a[i] is second argument, goes in first
10:      putparam r1   // sum is register r1, first argument, goes last
11:       r1 = call adder, 2 // function with two arguments on the stack
```

Problem 6: Attributive Grammar and Syntax-Directed Translation

In class we described a SDT translation scheme for array expressions using row-major organization. In this exercise you are asked to redo the SDT scheme for a column-major organization of the array data. Review the lectures note to understand the layout of a 2-dimensional array in column-major format. Specifically perform the following:

a. Indicate the attributes of your SDT scheme along with the auxiliary functions you are using.
b. Define a grammar and its the semantic actions for this translation.
c. Show the annotated parse tree and generated code for the assignment `x = A[y,z]` under the assumption that A is 10 x 20 array with $low_1 = low_2 = 1$ and sizeof(baseType(A)) = sizeof(int) = 4 bytes

Hint: You might have to modify the grammar so that instead of being left recursive is right recursive. Below is the grammar used for the row-major organization.

$L \rightarrow$ Elist]
Elist $\rightarrow$ Elist_1 , E
Elist $\rightarrow$ id [E
$E \rightarrow L$
$S \rightarrow L = E$
$L \rightarrow$ id

Answers:

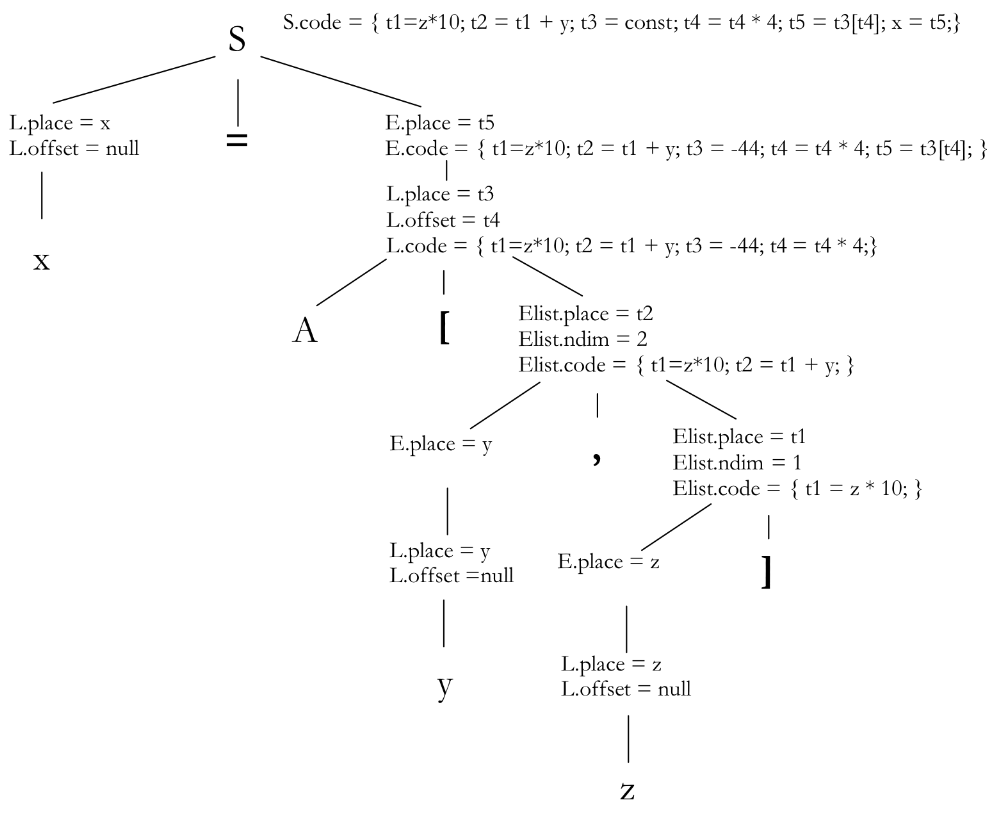
a.),b.) The basic idea is the same as in the row-major organization but rather than counting the dimensions from the left to the right we would like to count and accumulate the dimension sizes from the right to the left. This way the right-most index of the array indeces is the last dimension of the rows and should be multiplied by the first dimension, in this case the size of each row. As such the revised grammar that is right-recursive would work bettter. Below is such a grammar to which we are associating the same **synthesized** attributed as in the row-major formulation, namely, *place*, *offset* and *ndim* as described in class.

$S \rightarrow L = E$
$E \rightarrow L$
$L \rightarrow$ id
$L \rightarrow$ id [Elist
Elist $\rightarrow$ E]
Elist $\rightarrow$ E, Elist_1

We also use the same auxiliary function such as `numElemDim(A, dim)` and `constTerm(Array)` as described in class. Given this grammar we define the semantic actions as described below:

```
Elist → E ]          {
                       Elist.place = E.place
                       Elist.code = E.code;
                       Elist.ndim = 1;
                     }

Elist → E , Elist1   {
                       t = newtemp();
                       m = Elist1.ndim + 1;
                       Elist.ndim = m;
                       code1 = gen(t = Elist1.place * numElemDim(m));
                       code2 = gen(t = t + E.place);
                       Elist.place = t;
                       Elist.ndim = m;
                       Elist.code = append(Elist1.code,E.code,code1,code2);
                     }

L → id [ Elist       {
                       L.place = newtemp();
                       L.offset = newtemp();
                       code1 = gen(L.place '=' constTerm(id));
                       code2 = gen(L.offset '=' Elist.place * sizeof(baseType(id)));
                       L.code = append(Elist.code,code1,code2);
                     }

E → L                {
                       if (L.offset = NULL) then {
                         E.place = L.place;
                       } else {
                         E.place = newtemp;
                         E.code = gen(E.place = L.place[L.offset]);
                      }
                     }

S → L = E            {
                        if L.offset = NULL then
                            E.code = gen(L.place = E.place);
                        else
                           S.code = append(E.code,gen(L.place[L.offset] = E.place);
                     }

L → id               {
                        L.place = id.place;
                        L.offset = null;
                     }
```

c) For the assignment example x = A[y,z] we have the annotated parse tree shown below under the assumption that the base address for the array A is 0.


S
S.code = { t1=z*10; t2 = t1 + y; t3 = const; t4 = t4 * 4; t5 = t3[t4]; x = t5;}
L.place = x
L.offset = null
x
=
E.place = t5
E.code = { t1=z*10; t2 = t1 + y; t3 = -44; t4 = t4 * 4; t5 = t3[t4]; }
L.place = t3
L.offset = t4
L.code = { t1=z*10; t2 = t1 + y; t3 = -44; t4 = t4 * 4;}
A
[
Elist.place = t2
Elist.ndim = 2
Elist.code = { t1=z*10; t2 = t1 + y; }
E.place = y
,
Elist.place = t1
Elist.ndim = 1
Elist.code = { t1 = z * 10; }
L.place = y
L.offset =null
E.place = z
]
y
L.place = z
L.offset = null
z

Problem 7: Intermediate Code Generation

For the assignment instruction below perform the following:

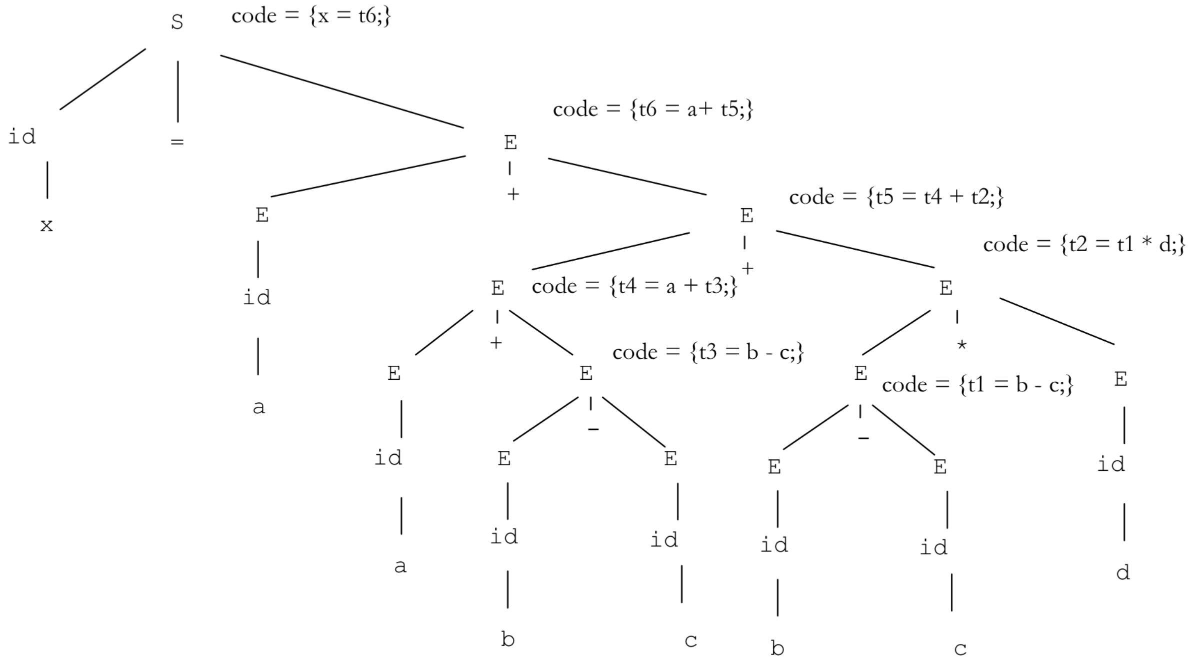
```
x = a +  (a + (b-c)) + ((b-c) * d);
```

a. Generate three-address instructions using the SDT scheme described in class and without any minimization of temporary variables.
b. Redo the code generation above but now reusing temporaries using the method described in class.
c. Argue that the solution found in b. is optimal.

Answer:

a. Below is the annotated (simplified) parse tree where we show the code that is added to the output for each node resulting the in the final three-address code sequence as follows:

```
t1 = b - c;
t2 = t1 * d;
t3 = b - c;
t4 = a + t3;
t5 = t4 + t2;
t6 = a + t5;
x  = t6;
```



b. Using the `newtemp` allocation function described in class that attempt to reuse a temporary as soon as its value is used we would get the code sequence shown below.

```
t1 = b - c;
t1 = t1 * d;  // as soon as t1 is used on the RHS it becomes available.
t2 = b - c;   // here we must use a second temp as t1 is still live.
t2 = a + t2;  // reuse of t2
t1 = t1 + t2; // reuse of t1 and t2 (both became free)
t1 = a + t1;  // reuse of t1
x  = t1;
```

c. Given that the solution above uses 2 temporaries, the best possible scenario would be for it to use a single temporary variable. In order to have a single temporary we would have to have the parse tree totally skewed towards the RHS. Even using the commutative, associative and distributive laws we cannot get this expression as a summation of basic terms. We will always need a second temporary for the computation of the product term. Not even the fact that we have a common sub-expression, the `(b-c)` factor helps out in this case. As such the next best number of temporary variables is 2, making this code optimal with respect to the number of temporary variables.

Note that you could improve the code sequence above by recognizing the common sub-expression `(b-c)` and save the result of this subtraction in a register to be used in the addition and product with `a` and `d` respectively. The resulting code uses the same number of registers but performs one less arithmetic operation.

```
t1 = b - c;
t2 = t1 * d;
t2 = b - c;   // not needed
t1 = a + t1;
t1 = t1 + t2;
t1 = a + t1;
x  = t1;
```

Problem 8: Control-Flow Constructs

In this exercise you are going to define the SDT scheme for function calls and procedures calls. The general idea is that you need to evaluate the various arguments of the functions and procedure, generate the code that puts their values into temporary variables, push the variables onto a stack using the three-address instructions and then invoke the procedure or function with the corresponding assignment to the variable, in case of a function invocation.

To keep this exercise simple you should assume that you already have a grammar for expressions that include function calls and procedure calls. You need to define the simple productions for these cases using the *arg_list* recursive grammar symbol as shown below. State your assumption for the attributes these additional grammar symbol need to have to support your code generation scheme.

```
Expr     →  ID ( ArgList );

ArgList →       ε
        |       Arg
        |       Arg ',' ArgList

Arg      →      Expr

Assign   →      Expr '=' Expr
```

The example below illustrates the generated code for a simple function invocation. Please recall that the arguments of the function may also be function call, so you need to be careful in the way you create temporary variable for holding the values of the various argument.

```
                                   t1= a
                                   t2 = b + 1
                                   putparam t2
x = soma(a,b+1);                   putparam t1
                                   t = call soma, 2
                                   x = t
```

Answer:

Really the only tricky part here it to save the temporary in which each parameter is to be computed into a stack so that we can pops these temporaries during the parsing and output the various `putparam` instructions with the correct identifier of the temporary. While popping you should also count the number of arguments that you need to have to issue the call instruction.

We thus define a synthesized attribute *cnt* to count the number of argument (although this would not be strictly necessary as one could do it by popping the stack until it would be empty and count at the same time the number of elements popped) and a *place* to save the temporary where a given value of an expression is saved. Also note here we are emitting the code right away as the reductions take place. For this reasons we are decoupling the assignment on a function call by having the function use a new temporary and then making the assignment to the *Expr* on the LHS of an assignment statement.

```
Expr     →   ID ( ArgList );   {     n = ArgList.cnt;
                                     t = newtemp();
                                     for(i = 0; i < n; i++){
                                       emit("putparam ",argsStack.pop());
                                     }
                                     emit("t = call ID, n");
                                     Expr.place = t;
                                    }
```

```
ArgList →   ε                   { ArgList.cnt = 0; }
         |  Arg                 { argsStack.push(Arg.place); ArgList.cnt ++; }
         |  Arg ‘,’ ArgList     { argsStack.push(Arg.place); ArgList.cnt ++; }

Arg      →  Expr                { Arg.place = Expr.place; }

Assign   →  Expr1 ‘=’ Expr2     { Emit(“Expr1.place = Expr2.place”); }
```