# Final Examination

- Please read all instructions (including these) carefully.
- Please print your name at the bottom of each page on the exam.
- There are nine questions on the exam, some in multiple parts. You have 3 hours to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. There are no "tricky" problems on the exam—each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary.

NAME: Sam P. L. Solution

SID or SS#: ______________________

| Problem | Max points | Points |
|---|---|---|
| 1 | 10 | |
| 2 | 15 | |
| 3 | 50 | |
| 4 | 20 | |
| 5 | 20 | |
| 6 | 15 | |
| 7 | 35 | |
| 8 | 15 | |
| 9 | 20 | |
| TOTAL | 200 | |

1. **Scope** (10 points)

   Give a simple program that produces different results if executed using lexical scoping than if executed using dynamic scoping. Show what your program produces in both cases. Use any reasonable and clear programming notation.

```
main()
    var x;

    proc p1
        var x;
        x := 1;
        p2();
    end;

    proc p2
        print(x);
    end;

x := 0;
p1();
end;

This program prints 0 with lexical scope and 1 with dynamic scope.
```

2. **Parsing** (15 points)

   For each of the following questions, we are looking for clarity and brevity as well as the right idea.

   (a) (5 points) Give a definition of the term *derivation*. What is a left-most derivation?

   ```
   Begin with the start symbol S. At each step, choose a non-terminal X and
   replace it by Y1...Yn, where X -> Y1...Yn is a production of the grammar.
   Any sequence of such replacements is a derivation.
   A left-most derivation always chooses the leftmost non-terminal for
   replacement.
   You could also say that a derivation ends in a string of terminals, but
   this is not required for full credit.
   ```

   (b) (5 points) Give one advantage of SLR(1) grammars over LR(1) grammars; give one advantage of LR(1) grammars over SLR(1) grammars.

   ```
   An SLR(1) parser generator produces tables of practical size, whereas an
   LR(1) parser generator often produces tables too large to be useful for
   realistic languages.  On the other hand, the class of LR(1) grammars is
   larger than the class of SLR(1) grammars.
   ```

   (c) (5 points) Consider a bottom-up parser for a grammar with no $\epsilon$-productions and no single productions (i.e., productions with a single symbol on the right-hand side). For an input string with $n$ tokens, what is the maximum number of reduce moves the parser can make? Why?

   All productions in the grammar have at least two symbols on the right-hand side and one symbol on the left-hand side. Thus, every reduction reduces the number of symbols by at least 1. Therefore, there can be at most $n - 1$ reductions.

3. (50 points)

   A few years from now, fanatical graduates of CS164 have succeeded in convincing the world that Cool should replace C++ as the object-oriented programming language of choice. Naturally, a Cool standards committee is formed. The committee cannot leave a superior design alone and decides that Cool needs some changes.

   (a) **Regular Expressions** (10 points)
   One of the committee members has an obsession with comments. She thinks it is ugly that one cannot write "*)" inside a "(*" comment; after all, what if that is what you want to write?

   The proposal is to replace all Cool comments with the following mechanism: A comment begins with one of the special characters $,!, or @ and ends with the same special character. Between the first and last character, the comment may contain any character except the first (and last) character. Comments may not be nested. For example, !cool@cs.berkeley.edu! and @Cool code rules!@ are valid comments, but !This is great!! and $Help! are not valid comments.

   Write a regular expression for comments. Use any reasonable and clear notation.

```
The notation [^c] means "all characters except c."

(![^!]*!) | (@[^@]*@) | ($[^$]*$)
```

Someone else on the committee feels that Cool is not expressive enough. He thinks that Cool simply *must* have backtracking. The proposed backtracking mechanism has two commands:

$$\begin{array}{l}\textbf{track}\ (\mathrm{e_1}, \mathrm{e_2}, \ldots, \mathrm{e_n}) \\ \textbf{back}\end{array}$$

Informally, a **track** command is evaluated as follows. First, $e_1$ is evaluated. If $e_1$ terminates normally, then the result of the expression is the result of evaluating $e_1$. If $e_1$ *fails* (see below), then $e_2$ is evaluated. If $e_2$ terminates normally, the result is the value of $e_2$; if $e_2$ fails, then $e_3$ is evaluated. This process continues until one of the expressions does not fail. A **back** command causes a tracked computation to *fail.* When a **back** command is executed, control is transferred to the nearest lexically enclosing **track** command and the next expression is evaluated. The following example evaluates to "1":

```
track (begin 3; back; end,
       back,
       1)
```

(b) **Parsing** (10 points)
Consider the following very simplified grammar for Cool expressions with **track** and **back**:

$$\begin{array}{rcl}
\mathrm{E} & \rightarrow & \textbf{if}\ \mathrm{E}\ \textbf{then}\ \mathrm{E}\ \textbf{else}\ \mathrm{E}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathrm{E}\ \textbf{loop}\ \mathrm{E}\ \textbf{pool} \\
 & | & \textbf{track}\ (\ \mathrm{ELIST} \\
 & | & \textbf{back} \\
\mathrm{ELIST} & \rightarrow & \mathrm{E}\ ) \\
 & | & \mathrm{E,\ ELIST}
\end{array}$$

Boldface denotes a keyword, capitalized words are non-terminals, and lowercase words are terminals. A member of the committee observes that the semantics of **track**($\mathrm{e_1}, \ldots, \mathrm{e_n}$) is undefined if all of the expressions fail. A simple way to guarantee that at least one expression succeeds is to forbid **back** statements inside the last expression $e_n$, except for **back** statements in (all but the last expression of) nested **track** statements. Rewrite the grammar so that $e_1, \ldots, e_{n-1}$ may contain **back** statements, but $e_n$ cannot, except for nested **track** statements, where the same rule applies recursively. Your grammar should allow any other proper nesting of expressions. Don't use ellipses (...) in your grammar.

$$\begin{array}{rcl}
\mathrm{E} & \rightarrow & \textbf{if}\ \mathrm{E}\ \textbf{then}\ \mathrm{E}\ \textbf{else}\ \mathrm{E}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathrm{E}\ \textbf{loop}\ \mathrm{E}\ \textbf{pool} \\
 & | & \textbf{track}\ (\ \mathrm{ELIST} \\
 & | & \textbf{back} \\
\mathrm{ELIST} & \rightarrow & \mathrm{F}\ ) \\
 & | & \mathrm{E,\ ELIST} \\
\mathrm{F} & \rightarrow & \textbf{if}\ \mathrm{F}\ \textbf{then}\ \mathrm{F}\ \textbf{else}\ \mathrm{F}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathrm{F}\ \textbf{loop}\ \mathrm{F}\ \textbf{pool} \\
 & | & \textbf{track}\ (\ \mathrm{ELIST}
\end{array}$$

(c) **Type Checking** (10 points)
Write a sound type checking rule for a **track** expression. Your rule should be reasonably precise—it isn't OK simply to say it has type Object. (Note: Don't worry about the type of **back**; you don't need it to answer this question.)

```
A typechecking rule is easy to write using the least-upper bound operation
on Cool types:

          A |- ei : Ti  1 <= i <= n
    --------------------------------------
    A |- track(e1,...,en) : lub(T1,...,Tn)


Another correct, but less general, answer is:

       A |- ei : T  1 <= i <= n
     ---------------------------
       A |- track(e1,...,en) : T
```

(d) **Code Generation and Semantic Actions** (20 points)
Write semantic actions to generate code for the original grammar (*not* the one you wrote) in part (b). For this problem, we are interested only that you generate the correct control structure.

Your semantic actions should use attributes. Assign to attribute *E.code* the code for expression *E*. You may use any other attributes (inherited or synthesized) you wish. Show the attribute definitions for each production.

Use only the following pseudo-assembly instructions in your solution:

| | |
|---|---|
| JUMP L | unconditional jump to label L |
| JUMPF L | jump to label L if the previous instruction evaluates to false |
| LABEL L | gives the label L to the next executable instruction |

You may use a function *newlabel()* that returns a unique label. The *code* attribute is a string; $s1||s2$ denotes concatenation of two strings $s1$ and $s2$. To help you get started, a partial solution for **while** loops is given below (the solution is partial because you may need to add attributes):

production: $E_1 \rightarrow$ **while** $E_2$ **loop** $E_3$ **pool**

> **while**.top = newlabel()
> **while**.exit = newlabel()
> $E_1$.code = LABEL **while**.top || $E_2$.code || JUMPF **while**.exit ||
> $E_3$.code || JUMP **while**.top || LABEL **while**.exit

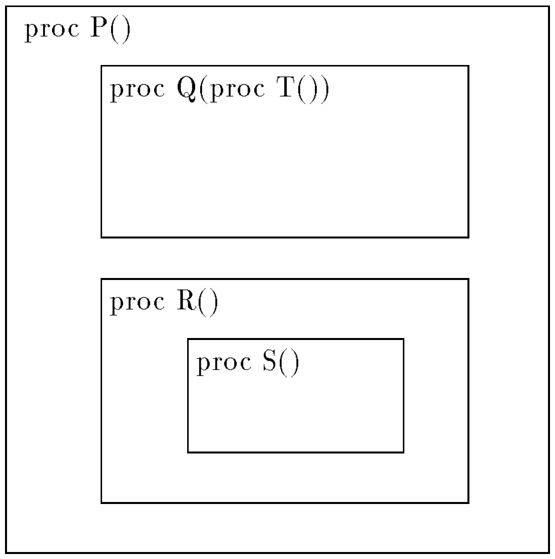
If all expressions in a **track** command fail, the code should jump to the label ERROR. Don't worry about adjusting the stack or modifying registers. Do not change the grammar.

(This page intentionally left almost blank.)

```
E1 -> if E2 then E3 else E4
      if.flabel = newlabel()
      if.exit   = newlabel()
      E1.code = E2.code || JUMPF if.flabel || E3.code || JUMP if.exit ||
                LABEL if.flabel || E4.code || LABEL if.exit
      E2.label = E3.label = E4.label = E1.label

E1 -> while E2 loop E3 pool
      rules given above, plus:
      E2.label = E3.label = E1.label

E ->  track ( ELIST
      ELIST.exit = newlabel()
      E.code = ELIST.code || LABEL ELIST.exit

E ->  back
      E.code = JUMP E.label

ELIST -> E )
      E.label = ERROR
      ELIST.code = E.code


ELIST1 -> E, ELIST2
      ELIST2.exit  = ELIST1.exit
      E.label = newlabel()
      ELIST1.code = E.code || JUMP ELIST1.exit || LABEL E.label || ELIST2.code
```

4. **Activation Records** (20 points)

Consider a program with the following lexical structure. The program is written in a lexically scoped language with nested procedures (like Pascal):

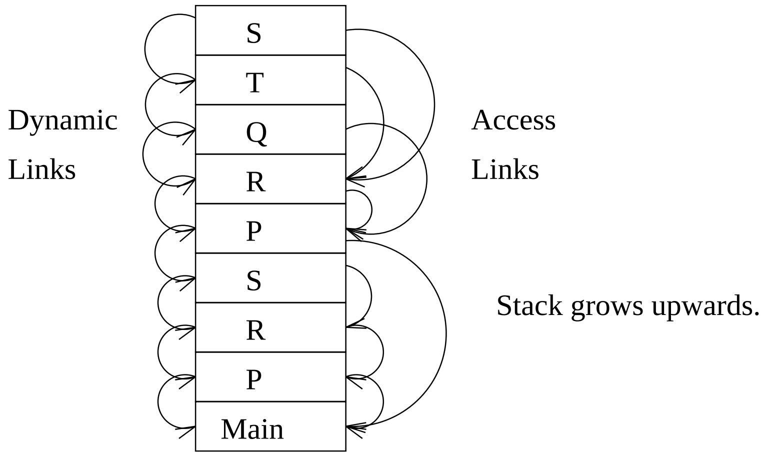


*P*, *R*, and *S* are parameterless procedures; *Q* takes a parameterless procedure *T* as a parameter. Suppose that at run-time the following sequence of calls is made:

P is called from some lexically-enclosing main program
P calls R
R calls S
S calls P
P calls R
R calls Q with S as a parameter
Q calls T
T calls S

Draw the stack of activation records present after this sequence of calls. You don't need to show the entire contents of the activation record—for each indicate only the name of the procedure being activated, the control (dynamic) link for that activation record, and the access (static) link for that activation record.

(This page intentionally left almost blank.)



5. **Cool and Type Checking** (20 points)

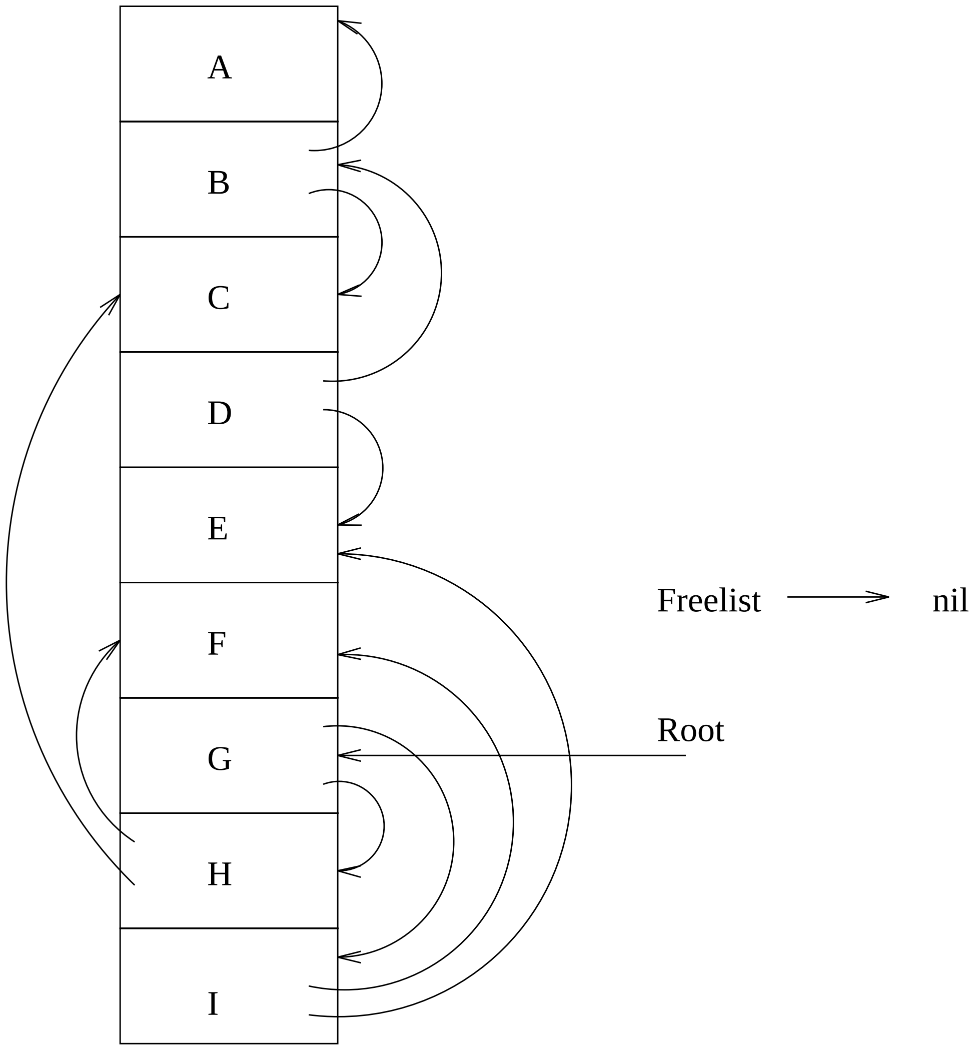
Consider the following Cool program. In the blanks provided, you should fill in both missing type declarations and the types inferred by the compiler for each expression. Fill in the most specific (most accurate) type possible. The final program should type check correctly using the declarations you fill in.

Each blank is the type of the expression immediately to the left; parentheses have been added where necessary to make clear which expression is meant.

```
Class A is
    a : Int ;
    init(x : Int ) : SELF_TYPE is
        begin (a Int ← x Int ) Int ; self SELF_TYPE end SELF_TYPE
    end;
end;

Class B inherits A is
    b : Int ← 1 Int ;
    getb() : Int is b Int end;
end;

Class C inherits A is
    c : Int ← 2 Int ;
    getc() : Int is c Int end;
end;

class Main inherits IO is
    main() : A is
        let y : Bool in
            case (if (y Bool ← ((in_int() Int = 0 Int ) Bool )) Bool then
                    (new B) B
                else
                    (new C) C ) A
            of
                x : B ⇒ (x B .init((x B .getb()) Int )) B ;
                y : C ⇒ (y C .init((y C .getc()) Int )) C ;
            esac A
        end A
    end
end;
```

6. **(Garbage Collection)** (15 points)

Garbage collect the following heap using Mark & Sweep garbage collection. Clearly indicate which cells will be marked, and construct the free list resulting from the collection.



```
Marked: C,E,F,G,H,I

Free-list -> A -> B -> D
```

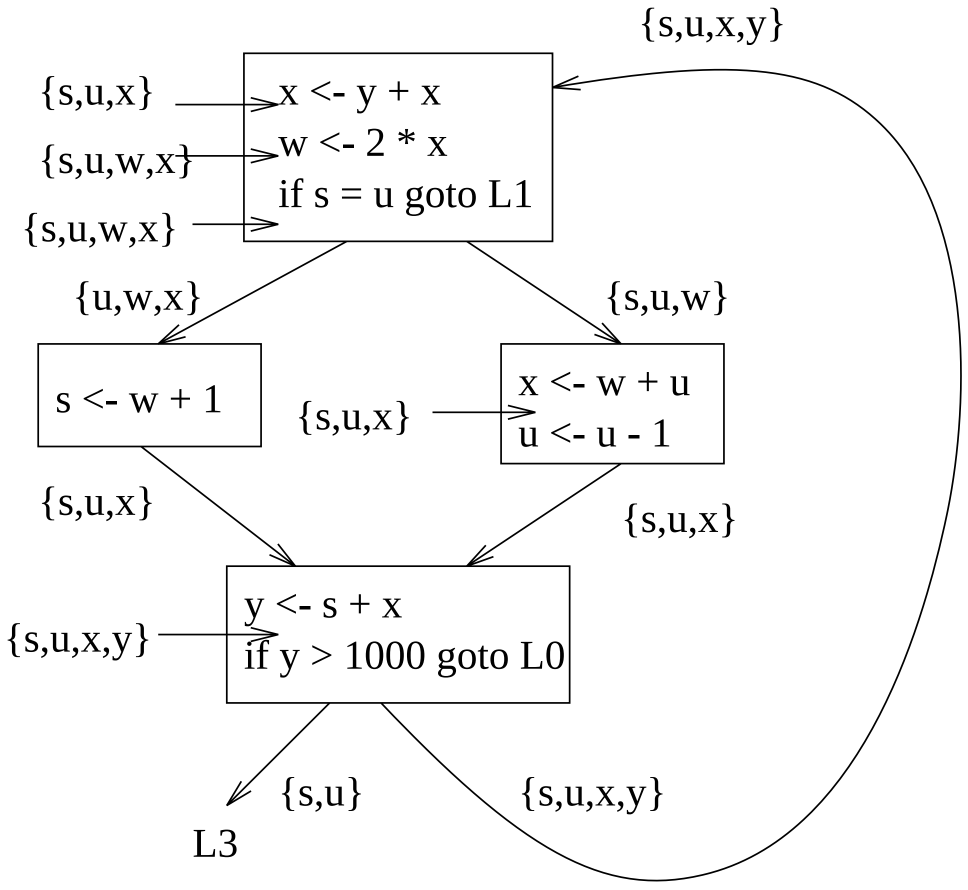
7. **(Dataflow Analysis and Register Allocation)** (35 points)

   Consider the following fragment of intermediate code:

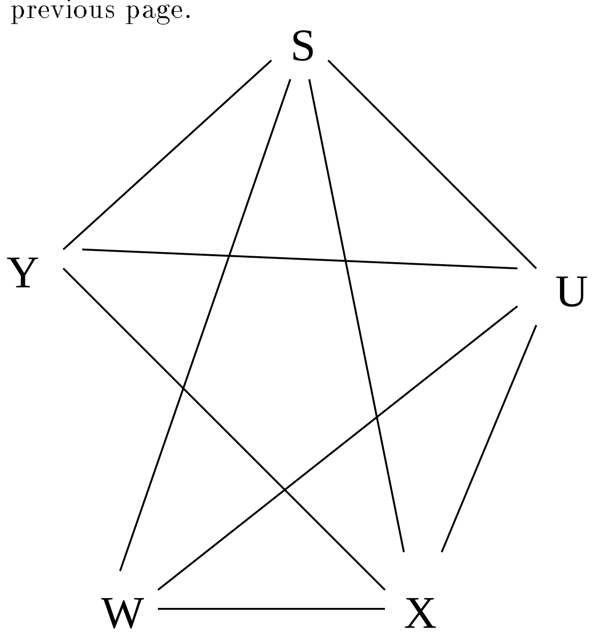
```
L0:
x <- y + x
w <- 2 * x
if s = u goto L1
x <- w + u
u <- u - 1
goto L2
L1:
s <- w + 1
L2:
y <- s + x
if y > 1000 goto L0
L3:
```

   (a) (5 points) Draw a control-flow graph for this piece of code. Place each basic block in a single node; be sure to include the conditionals in the basic blocks.

   (b) (15 points) Annotate your control-flow graph with the set of variables live before and after every statement (not just before and after every block!), assuming that *s* and *u* are live at label L3. Make sure it is clear where your annotations are placed.



(c) (15 points) Draw the register interference graph for the intermediate code given on the previous page.



8. **Type Checking** (15 points)

Consider the following C-like expression language:

$$
\begin{array}{lcl}
\mathrm{e} & \rightarrow & \mathrm{e}_1[\mathrm{e}_2] \\
 & | & \&\mathrm{e} \\
 & | & *\mathrm{e} \\
 & | & \mathrm{i}
\end{array}
$$

In this grammar, $i$ represents an integer. Now consider the following type language and type rules:

$$
\begin{array}{lcl}
\mathrm{T} & \rightarrow & \mathrm{int} \\
 & | & \mathrm{pointer(T)} \\
 & | & \mathrm{array(T)}
\end{array}
$$

$$\frac{}{A \vdash i : \mathrm{int}} \quad [INT]$$

$$\frac{A \vdash e_1 : \mathrm{array(T)} \quad A \vdash e_2 : \mathrm{int}}{A \vdash e_1[e_2] : \mathrm{T}} \quad [ARRAY]$$

$$\frac{A \vdash e : \mathrm{T}}{A \vdash \&e : \mathrm{pointer(T)}} \quad [POINTER]$$

$$\frac{A \vdash e : \mathrm{pointer(T)}}{A \vdash *e : \mathrm{T}} \quad [DEREF]$$

Now consider the expression &((*B)[1]). (The parentheses are included only to clarify precedence and are not part of the expression.) Given the type assumptions $A = \{B : \mathrm{pointer(array(int))}\}$, show the type derivation for the expression. Indicate the rule you use at each step.

```
      A |- B : pointer(array(int))
     ----------------------------- [DEREF]  ------------ [INT]
        A |- *B : array(int)                 A |- 1 : int
       ------------------------------------------- [ARRAY]
           A |- (*B)[1] : int
   ------------------------------------------------------- [POINTER]
   { B:pointer(array(int)) } |- &((*B)[1]) : pointer(int)
```

9. **Optimization** (20 points)

   Consider the following fragment of intermediate code:

```
y := w
z := 4
v := y * y
u := z + 2
r := w ** 2   (* exponentiation *)
t := r + v
s := u * t
```

   Assume that only the variable *s* is live on exit from this fragment. Show the result of applying as much constant propagation, algebraic simplification, common sub-expression elimination, constant folding, and dead code elimination as possible to this code. Show the optimizations you perform and the order in which they are applied as part of your answer. You need not show the entire code sequence after every optimization, but you should explain clearly what changes at each step.

```
w ** 2  => w * w              algebraic simplification/strength reduction
replace y by w                copy propagation
replace r := w * w by r := v  common subexpression elimination
replace r by v                copy propagation
replace z by 4                constant propagation
replace 4+2 by 6              constant folding
replace u by 6                constant propagation
remove assignments to y,z,u,r dead code elimination

Result:
v := w * w
t := v + v
s := 6 * t

If you wanted to get really fancy---not required for full credit---you
could continue:

t := 2 * v                    algebraic "optimization" (see below)
replace s := 6 * t by 12 * v  a kind of constant propagation
eliminate assignment to t     dead code

Result:
v := w * w
s := 12 * v

Most compilers will not find this last sequence of optimizations, because
the step v + v => 2 * v is not an improvement on most machines (in
other words, * is usually slower than +).
```