Com Sci 221 — Programming Languages
Take-Home Midterm Exam, Autumn 2000
Due Monday, 6 November, 1:30 PM in class
5 problems on pages 2–7

- This is a self-timed exam. Please limit yourself to 3 working hours from the time that you open the paper, until you complete the solutions that you hand in. You may take a break up to 30 minutes during the exam, but you should neither work on the problems nor study the class material during the break.
- You may use the textbook, my class materials, any other published material, and notes of your own.
- There are 5 problems. They cover material in Chapters 1–4 of the text, and the class lectures through datatypes. Each numbered problem is weighted equally for grading purposes.
- The problem statements are a bit wordy, since I will not be nearby to answer questions. The wordiness does *not* not indicate subtlety.
- You may use whatever sort of paper you find convenient, but please make your answers legible, and write your name on every sheet.
- Please deliver your finished exam to me on paper at the beginning of class.

Problem 1: abstract syntax, postfix, evaluation

Draw the abstract syntax tree and postfix forms for the expressions

1. $a + (b * ((c + d) * e))$
2. $(((a + b) * c) + d) * e$
3. $a + ((b * (c + d)) * e)$
4. $a + (b * (c + (d * e)))$

The values of $a, \ldots, e$ are in memory, but our machine can only do arithmetic operations on registers. Suppose that we are committed to evaluating all expressions simple-mindedly from left to right. We can construct machine code to evaluate the expression by following the left-to-right sequence in the postfix form. Here is the machine code for expression 1, using 4 registers:

$$\begin{aligned}
R_1 &:= a \\
R_2 &:= b \\
R_3 &:= c \\
R_4 &:= d \\
R_3 &:= R_3 + R_4 \\
R_4 &:= e \\
R_3 &:= R_3 * R_4 \\
R_2 &:= R_2 * R_3 \\
R_1 &:= R_1 + R_2
\end{aligned}$$

Every operation is done in an assignment of the form $R_i := R_i \circ R_{i+1}$ ($\circ$ stands for $+$ or $*$). The value of the whole expression always ends up in R_1. We reuse registers as much as possible while keeping strictly to the left-to-right evaluation dictated by the postfix expression. Depending on the form of the abstract syntax tree, an expression with $n \geq 2$ operands may require anything from 2 to n registers.

Translate the postfix expressions for 2, 3, and 4 into pure left-to-right evaluation code in the same way.

Explain very briefly how to determine the number of registers required for this sort of evaluation from the shape of the abstract syntax tree. Look for a very simple property of the paths from root to leaf in the tree.

Problem 2: context-free grammar

Recall the context-free grammar for arithmetic expressions that gives precedence to $*$ and $/$ over $+$ and $-$:

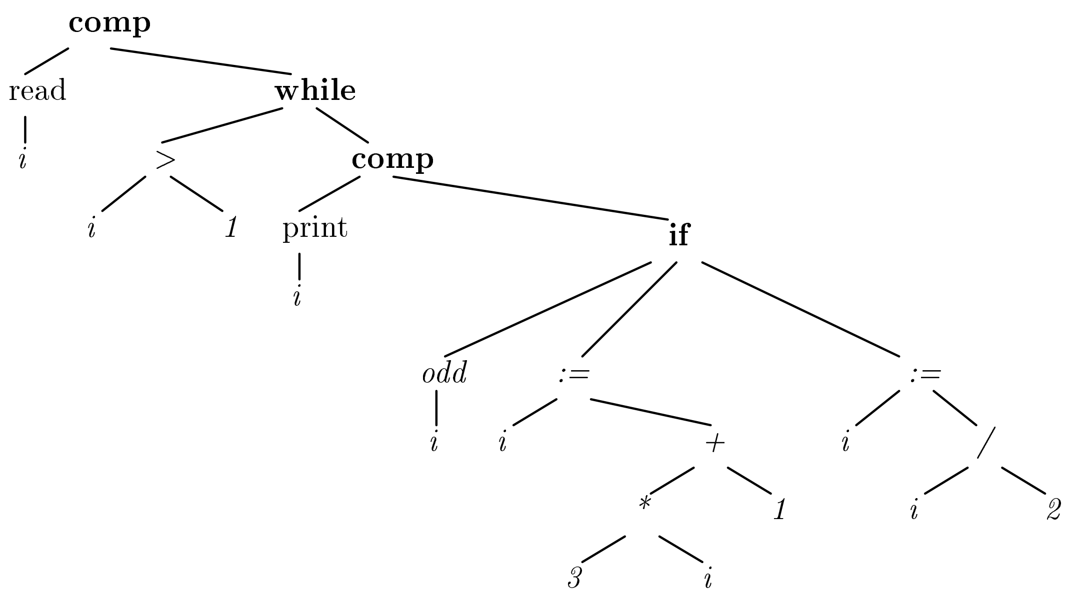
$$
\begin{aligned}
E &\rightarrow E + T \\
E &\rightarrow E - T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow T / F \\
T &\rightarrow F \\
F &\rightarrow C \\
F &\rightarrow V \\
F &\rightarrow (E)
\end{aligned}
$$

Assume that C derives numerical constants and V derives variable symbols. Modify the grammar to allow a single unary minus sign before any subexpression. For example, $x + -y$, $x - -y$, $-w * z$, $-x + y$, $-(x - y)$, $-(-x)$, are allowed, but not $x + - - y$, nor $x - + y$. A unary $-$ has the highest precedence. (It should go without saying that your grammar must be unambiguous, and its parse trees must correspond sensibly to abstract syntax.)

Problem 3: structured programs

Here is a simple program with its abstract syntax tree:

```
read(i);
while i > 1 do begin
        print(i);
        if odd(i) then begin
                i := 3 * i + 1
        end else begin
                i := i/2
        end
end
```



1. Draw the flowchart for the program. Draw a box around the portion of the flowchart that corresponds to each subtree in the abstract syntax tree (that is, to the whole **while** loop, the body of the **while** loop, and the whole **if-then-else** statement—the read, print, and assignment commands are already in boxes). How many flowchart edges go into each box? How many go out of each box? *(problem continued on the next page)*

2. Draw abstract syntax tree and flowchart for the following variant of the program above, with boxes around the portions of flowchart corresponding to the subtrees of the abstract syntax tree.

   read(i);
   while true do begin
       if $i \leq 1$ **then break**;
       print(i);
       if odd(i) **then begin**
           $i := 3 * i + 1$
       end else begin
           $i := i/2$
       end
   end

3. Explain very briefly (2–5 sentences should do it) the reason that pure structured programming does not allow **goto**s nor **break**s, in terms of the structural difference between the two examples above.

Problem 4: reasoning with invariants

Consider the usual iterative program fragment to reverse a list s_0:

$s := s_0;$
$r := empty;$
while $s \neq empty$ **do begin**
    $r := cons(head(s), r);$
    $s := tail(s)$
end

cons adds a new first element to a list, *head* returns the first element of a list, and *tail* returns all but the first element of a list.

Give appropriate precondition, postcondition, and loop invariant that could be used to show that the program fragment is correct (but don't do the proof). Use the mathematical primitives p^R (reversal of p) and $p \,.\, q$ (p concatenated with q—equivalently q appended to the end of p) to express your assertions. The invariant is one very simple equation using the reversal and concatenation operations.

Problem 5: datatypes

In C the following two assignments are precisely equivalent:

$$*C[i] = (*B[j]).first$$

$$*C[i] = B[j]\texttt{->}first$$

I list both of them since some of you may be more familiar with one form rather than the other. Describe the execution of the assignment above briefly but precisely, mentioning every l value and every r value involved, and every way in which an l value is treated as an r value or vice versa. You don't need to give the formula for calculating addresses of array components—merely mention that such a calculation must be done. Here are the first 3 steps to get you started. I'll abbreviate 'l value of ... ' by '$l(\ldots)$' and 'r value of ... ' by '$r(\ldots)$'.

1. dereference $l(j)$ to $r(j)$
2. calculate $l(B[j])$ from $l(B)$ and $r(j)$
3. dereference $l(B[j])$ to $r(B[j])$
4. ...

There are 11 steps total (including the 3 that I wrote above). If you follow the second version of the assignment, keep in mind that the '`->`' operator corresponds to a combination of a '.' and a '$*$'.