Exam 2 - 628 Cover Sheet

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device. I also affirm that I have completed the exam according to the restrictions listed in the exam document.

Signed: ______________________________

Print name: ______________________________

Date: ______________________________

Please submit a signed and dated copy of this cover sheet as the first page of your exam submission.

You will not receive credit for the exam unless your submission includes this (signed and dated) cover sheet.

601.628 Compilers and Interpreters Fall 2020, Exam 2 - 628

Due: Wednesday, October 28 by 3:00pm EDT

The permitted resources for this exam are:

- The textbook(s)
- Materials posted directly on the course website (e.g., slides)

Do *not* use any resources other than the ones explicitly noted above.

You may *not* write program(s) or use automated calculation devices or programs. You will need to do required calculations by hand. In other words, this is a "pencil and paper" exam, but you can (and should) type your answers. (Neatly handwritten answers are also fine.)

Do *not* discuss the exam with anyone else: your answers must be your own answers.

You will submit your answers to Gradescope as "Exam 2 - 628" in PDF format. When you upload to Gradescope, you will need to select the page of your submitted document corresponding to your answer to each question. You may use software (word processor, LaTeX, etc.) to prepare your answers.

Important: Make sure the first page of your submitted document is the signed and dated cover sheet (which is the first page of this exam document.) You will not receive credit for the exam if your submission does not include the (signed and dated) cover sheet.

Important: Show your work, and justify your answers. "Bare" answers (without supporting work or justification) may not receive full credit.

[Questions begin on next page.]

For Questions 1–5, consider the following infix expression grammar (nonterminal symbols are in *italics*, terminal symbols are $\boxed{\texttt{+ * ( )}\ \textbf{identifier int-literal}}$

$$
\begin{aligned}
&\textit{expr} \rightarrow \textit{expr} + \textit{term}\\
&\textit{expr} \rightarrow \textit{term}\\
&\textit{term} \rightarrow \textit{term} * \textit{factor}\\
&\textit{term} \rightarrow \textit{factor}\\
&\textit{factor} \rightarrow \textbf{identifier}\\
&\textit{factor} \rightarrow \textbf{int-literal}\\
&\textit{factor} \rightarrow (\ \textit{expr}\ )
\end{aligned}
$$

Assume that occurrences of **identifer** are variable references, and that + and * are addition and multiplication.

For the questions asking you to write an attribute grammar (using the grammar productions shown above), you will need to distinguish multiple occurrences of the same symbol within a production; for example

$$\textit{expr} \rightarrow \textit{expr} + \textit{term}$$

could become

$$\textit{expr}_0 \rightarrow \textit{expr}_1 + \textit{term}$$

Question 1. [8 points]

Show a derivation and parse tree for the input

```
a + 3 * (b + (9 + 6))
```

Assume that `a` and `b` are occurrences of **identifier** and that `3`, `9`, and `6` are occurrences of **int-literal**.

Question 2. [16 points]

(a) Write an attribute grammar (with the productions of the infix expression grammar above) which defines an "`isconst`" attribute, which should be true or false depending on whether the expression computes a constant value.

(b) Show the parse tree you constructed in Question 1 annotated with "`isconst`" attribute values. Briefly explain, in words, the order in which attribute values were computed.

[Questions continue on next page.]

Question 3. [16 points]

(a) In a *postfix expression*, the operator is written *after* the operands. For example, the infix expression `a + 3` would be written as `a 3 +`.

Write an attribute grammar (with the productions of the infix expression grammar above) which defines a "`postfix`" attribute. The value of this attribute should be the translation of the infix expression into an equivalent postfix expression.

For example, for the input string in Question 1 (`a + 3 * (b + (9 + 6))`), the generated postfix expression should be

```
a 3 b 9 6 + + * +
```

(b) Show the parse tree you constructed in Question 1 annotated with "`postfix`" attribute values. Briefly explain, in words, the order in which attribute values were computed.

Question 4. [16 points]

(a) Write an attribute grammar (with the productions of the infix expression grammar above) which defines an "`exprid`" attribute. Each node in the parse tree should have an `exprid` value. Nodes in the parse tree which are the roots of equivalent subtrees must have identical `exprid` values, while nodes which are roots of non-equivalent subtrees must have different `exprid` values. You may define additional attributes as needed to support the computation of `exprid` values.

Subtrees are "equivalent" if they are structurally the same, ignoring parentheses. For example, the expressions `a + b * 3` and `a + (b * 3)` produce parse trees that should be considered equivalent. (Equivalence in this sense doesn't mean that `a + b` and `b + a` should be considered equivalent just because addition is commutative.)

You may choose any reasonable data type as the value type for the `exprid` attribute.

(b) Show the parse tree for the input string `(a + b) + 3 * (10 + a + b)` annotated with `exprid` attribute values (and other supporting attribute values, if any.) Briefly explain, in words, the order in which attribute values were computed.

Question 5. [22 points]

Show a bottom-up parse of the input string `a + 3 * (b + (9 + 6))`. At each step, show the current stack, input string symbols, and the action that is carried out (shift or reduce.) For reduce actions, indicate which production is being reduced. Use "$" to indicate the bottom of the stack and also the end of the input string.

Recall that the reductions in a completed bottom-up parse, when read backwards, are a rightmost derivation of the input string.

The following incomplete table shows how you should start.

Stack	Input string	Action
$	`a + 3 * (b + (9 + 6)) $`	

[Exam continues on next page.]

Question 6. [22 points]

Consider the following program, which is written using a language similar to the one in Assignments 3–6, but which

- has true block scoping, meaning that variables can be declared within nested blocks, and
- has string constants

```
PROGRAM scopes;

BEGIN
  VAR x, fac, count : INTEGER;
  VAR factors : ARRAY 20 of INTEGER;

  READ x;
  fac := x DIV 2;
  count := 0;

  WHILE fac > 1 DO
    VAR rem : INTEGER;

    rem := x MOD fac;

    IF rem = 0 THEN
      VAR nextcount : INTEGER;
      factors[count] := fac;
      nextcount := count + 1;
      count := nextcount;
    ELSE
      VAR y : INTEGER;
      y := fac;
      WRITE y;
      WRITE " is not a divisor!\n";
    END;

    fac := fac - 1;
  END;
END.
```

Assume that the `INTEGER` type requires 4 bytes of storage.

Create symbol tables for this program. Each symbol table entry should indicate

- the lexical depth (0 for the outermost scope)
- the kind of symbol (variable, constant, or type)
- name
- data type

- offset in the storage area allocated for the program variables

For each variable reference in the program, draw an arrow to the corresponding symbol table entry.

Finally, indicate the total amount of storage required for all variables. Storage allocations should use the minimum number of bytes required to ensure that no two live variables or array elements have overlapping storage allocations when the program runs.