Assignment 2

CSE 3341

Due Wednesday, September 19, by 2:45 pm

Consider the Core language defined in the lecture notes. Suppose we want to add to Core a statement with the following structure (somewhat similar to the `CASE` statement in the Modula-2 programming language):

```
case  x  of
          l1   :   e1
   |      l2   :   e2
   |      l3   :   e3
    ...
   |      ln   :   en
          else     en+1
end
```

Here

- x is some program variable of integer type
- l_i is a non-empty list of integer constants, separated by commas (could be a single constant)
- e_i ($1 \leq n \leq n+1$) is an expression that evaluates to an integer value (equivalent to $\langle expr \rangle$ from the grammar of Core)
- `case` and `of` are new keywords; `else` and `end` are keywords that are already in Core
- `:` and `|` are special symbols used in the syntax of the statement (each one is a separate single-character token, e.g., `COLON` and `BAR`)

The number $n \geq 1$ could be arbitrarily large (but $n = 0$ is not allowed). This statement has the following semantics: if the current value of variable x is equal to one of the integer constants in list l_1, then expression e_1 is evaluated, the resulting integer value is assigned to x, and the execution of the `case` statement completes. Otherwise, if the value of x is equal to one of the constants in l_2, then e_2 is evaluated, the resulting integer value is assigned to x, and the execution of the `case` statement completes. Otherwise, if the value of x matches a constant in l_3, etc. If the value of x does not trigger any of the first n cases, expression e_{n+1} is evaluated, the resulting integer value is assigned to x, and the execution of the statement completes. Note that the order in which the n cases appear in the statement affects the semantics of the run-time execution. Also note that the `else` clause must be a part of the statement, and it must appear as the last alternative.

1. (5 pts) Extend the grammar of Core to allow such statements. Show *all* changes to the grammar. Reuse as much as possible from the existing non-terminals and productions in Core. Make sure that the new grammar is appropriate for recursive descent parsing. In addition, show a simple example of a `case` statement and the corresponding parse tree based on your grammar.

2. (5 pts) Describe how the parse tree representation (table `PT` from the lecture notes) can be used to represent the *new* kinds of parse tree nodes. Keep in mind that for some terminals (e.g., `COMMA`, `BAR`, etc.) we do not need to store info in the table. Describe in detail the structure of the table: the number of columns, the data stored in each column (and the data type—e.g. `int`, etc.), the meaning of this column data, etc. In addition, show `PT` for the `case` statement example you defined in the previous question—but show only the rows for the `case` statement and all of its sub-components (the entire parse subtree, all the way down to the leaves), not for other non-terminals such as $\langle prog \rangle$.

   (Please turn over.)

3. (10 pts) Assuming the table representation from (2), how should we change the recursive descent parser for Core? Show the pseudo-code for all parser procedures, using the style of notation we have used in class (e.g., see slide 12). You need to show

   - *all* changes that must be made to the procedures for the original grammar discussed in class (if nothing changes in a procedure, do not show it); this includes all calls to the scanner and all manipulations of `PT`
   - the *complete* parsing procedures corresponding to all new non-terminals that you introduced in (1); this includes all calls to the scanner and all manipulations of `PT`
   - do *not* show any of the code inside the scanner, but *do* show the calls from the parser to the scanner
   - do *not* write any error-handling code (i.e., assume the input program is always valid)

   The scanner API has two procedures: `getCurrent` which returns the current token but does *not* advance to the next token, and `nextToken` which moves to the next token without returning anything.

4. (5 pts) How should we change the recursive descent printer for Core? Show all changes to the original printer described in class: (a) the complete bodies of any changed existing procedures, and (b) the complete bodies of any new procedures. Show explicitly all accesses to `PT` elements.

5. (10 pts) How should we change the recursive descent executor for Core? Show all changes to the original executor described in class: (a) the complete bodies of any changed existing procedures, and (b) the complete bodies of any new procedures. The implementation should use your `PT` representation from (2), as created by your parser from (3). Assume that there are two helper procedures `setValue` and `getValue` that can be used to change and look up the values of program variables, respectively. You do not need to write the code for these procedures—assume that they are already defined by someone else. Procedure `getValue` takes as input a string value representing the name of a program variable, and returns an integer value representing the current value of that variable. Procedure `setValue` takes two parameters: a string value representing the name of a program variable, and an integer value representing the new value of that variable. Do not worry about uninitialized variables — assume that the input program never tries to read the value of an uninitialized variable. Make sure you explicitly show all necessary calls to these helper procedures from within your executor procedures.

- Assignments should be done independently. General high-level discussion of assignments with others in the class is allowed, but **all of the actual work** should be your own. Assignments that show excessive similarities will be taken as evidence of cheating and dealt with accordingly.
- Assignments should be turned in **by the end of class** on the due day. Late assignments turned in by the end of the next class will be graded with 30% reduction. Assignments turned in later than that will not be accepted.
- Make the assignments readable and understandable. They should be handed in on regular paper, legibly written or typed. If you have more than one sheet, **staple the sheets together**. If the grader has trouble reading or understanding what you have done, points will be deducted even if it can finally be determined that you have the correct answer.
- Your solutions have to be **precise and detailed**: you have to work out **all** details that are necessary to solve the problem using the approaches discussed in class. You also have to write your solutions in a way that convinces the grader that you understand all these details. Be careful, precise, and thorough.