CSCI465: Guidelines For Writing one Pass Mini-Pascal Compiler

1. Study the Pascal grammar
   (a) The Pascal grammar essentially splits into declarations, subprogram-declarations and compound-statement Examine each and its form/role carefully: declarations are where things are defined and statements are where things are used/executed
2. Test scanner/parser on input programs (good and bad pascal source code)
3. Create syntax trees for rules involving *statements* and *expressions*
   (a) A syntax tree record should (at least) encode:
      - Expressions: numbers, identifiers, operators, function calls, array access
      - Statements: assignment, **if**, **while**, procedure calls
   (b) Need extra attributes (mimic the yacc union structure) for:
      - Identifiers: pointer to symbol table record
      - Numbers: numeric values (integer or real)
      - Operators: operator types
      - Others (add as needed): access depth (run-time information), etc.
   (c) Not a bad idea to generate syntax tree for the entire program but not necessary
   (d) Create additional structures (trees or otherwise) for declarations (see relevant grammar rules)
   (e) Create *type* structure (separate from syntax tree) *type* rules (including both basic and array types; can be expanded for future)
   (f) Implementation tips
      - Write a visual tree print (helps debugging)
      - Test your trees by calling visual print from yacc (at minimum check compound-statement)

5. Build a symbol table that handles *stack*-based scoping rule

   (a) A symbol table record for an identifier should have the following information:

   - Name (string)
   - Class (variable: array or not; subroutine: function or procedure)
   - Type (pointer to type structure): NULL for procedure
   - List of argument types (for function or procedure only)
   - Location: local or parameter (run-time information for variable only)
   - Index: position in declaration list (run-time information for variable only)
   - Others (as the need arises)

   (b) Implementation tips

   - A traditional choice is a stackable hash table (bucket chaining for collision resolution)
   - Key functions required include: *push-new-scope*, *pop-top-scope*, *local-lookup*, *global-lookup*, *local-insert*, pick, etc
   - Test functionalities of symbol table separately (scaffolding test)

6. Write necessary semantic and type checking functions as a separate module

7. Build your code generator as a separate module:

   (a) Start with expressions (as done in class and lab demos) (ignore **real**-valued entities for now)

   (b) Move on to statements:

   - Handle input/output (setup calls to scanf and printf)
   - Handle assignment statement (requires locating identifiers at run-time (ignore arrays for now))
   - Handle **if** statement (requires label management)
   - Handle **while** statement (requires label management)
   - Handle procedure statement (requires calling sequence management)

   (c) Solve the activation record size value computation

   (d) Solve the nonlocal access problem

Check List

Module	Status	Limitations
Scanner		
Parser		
Syntax Tree		
Symbol Table		
Semantic Analyzer		
Code Generator		