Part 1:
Your first project is to build a scanner for a subset of Turing. You are to use lex (or write equivalent code) for the scanner. The productions for the constructs that I will have you work with are on the attached sheet. The tokens are:

- The class of Identifiers: a case-sensitive sequence of letters, digits and underscores that start with a letter (not an underscore).
- The class of Integers (without leading signs)
- The class of Real Constants (both 'Int . Int' and 'Int . Int* Exponent')
- Any of the Reserved Words, all of which are in lower case: var, int, real, boolean, record, end, bind, to, assert, begin, loop, exit, when, if, then, elsif, else, put, or, and, not, not=, div, mod
- Any of the operators ; : := < > = <= >= + - * / . , ( )

You will also need to write a tester program that reads stdin, separates out the tokens, and writes the token number and the string representing the token to stdout.

Part 2:
Your second project is to build a parser for the subset of Turing. You are to use yacc (or write recursive descent code) for the parser. The productions are on the attached page. I want your code to build an AST (Abstract Syntax Tree) for the Turing input, and to print out a representation of the AST at the end.
Description of Mini-Turing

CFG Productions

program → pStateDeclSeq
pStateDeclSeq → ϵ
               → statement ";" pStateDeclSeq
               → "var" id_list ";:" type ";" pStateDeclSeq
idlist → Ident | Ident "," idlist
type → "int" | "real" | "boolean"
       → "record" field_list ";end" "record"
field_list → id_list ";:" type
           → id_list ";:" type ";" field_list
state_decls → ϵ
             → statement ";" bStateDeclSeq
             → declaration ";" bStateDeclSeq
declaration → "var" id_list ";:" type
              → "bind" id ";to" ref
              → "bind" "var" id ";to" ref
statement → ref ";=" expn
         → "assert" bool_expn
         → "begin" state_decls "end"
         → "loop" state_decls "end" "loop"
         → "exit" | "exit" "when" bool_expn
         → "if" bool_expn "then" state_decls end_if
ref → Ident | Ident "." Ident
end_if → "end" "if" | "else" state_decls "end" "if"
         → "elsif" bool_expn "then" state_decls end_if
expr → expr "or" and_expr | and_expr
and_expr → and_expr "and" not_expr | not_expr
not_expr → "not" not_expr | rel_expr
rel_expr → sum | rel_expr "=" sum | rel_expr "not=" sum
         → rel_expr ";" sum | rel_expr ";<" sum
         → rel_expr ";>" sum | rel_expr ";>=" sum
sum → prod | sum "+" prod | sum "-" prod
prod → factor | prod "*" factor | prod "/" factor
     → prod "div" factor | prod "mod" factor
factor → "+" basic | "-" basic | basic
basic → ref | "(" expr ")" | IntConst | RealConst