Your next task is to construct a parser using a parser generator (e.g. Bison, ML-YACC, etc.) that recognizes strings in the language whose grammar is provided below and uses the lexer that you have already written. The parser should create an abstract syntax tree (AST) whose leaves are the necessary terminals (represented at this point by lexeme-token pairs) and whose internal nodes are labeled with the kind of sentence represented by the subtree.

You need to define, at this point, the structure of the abstract syntax tree based on the grammar given at the bottom of this document. For example, the C++ declaration might look like

```cpp
eenum node_c { PROGRAM, BLOCK, TYPE_DEF, VAR_DEF, ... }
class Node {
public:
    ... 
    node_c get_node_class();
    ...
protected:
    node_c node_class;
    ...
}
class Program: public Node {
public:
    Program(...) { node_class = PROGRAM; ... }
    ...
protected:
    token_pair name;
    node *type_defs[];
    node *var_defs[];
    node *subprog_defs[];
    node *body;
    ...
}
```

As you add phases to the compiler, you will need to go back and add new fields to the various node types in order to hold other information (type, size, etc.) that is gathered by subsequent phases. For now, though, it is sufficient to define only those fields required to represent the AST.

Be sure to note that there doesn't have to be an interior node in the AST for every non-terminal in the grammar below. For example, the syntax of arithmetic expressions has a large number of non-terminals (such as simple_expression, term, and factor) that are solely introduced for the purpose of specifying operator precedence. There is no need to label interior nodes as being simple_expressions, terms, or factors. For example, `x+y*2`
can simply be represented as:

```
(leaf, "x") -> binary operation -> (leaf, "y")
     |                  |                  |
     |                  |                  |
(leaf, "\+")     (leaf, "\*")   (leaf, "2")
```

where `leaf`, `\+`, etc. are the tokens returned by your lexer. Also, not every terminal in the language needs to appear as a leaf in the AST. For example, there’s no need to have leaves labeled with keyword tokens, such as `begin`, `end`, `if`, `then`, etc., in the AST.

**What you should turn in**

You should turn in:

1. The parser-generator specification along with an updated (if necessary) lexer, and
2. A driver program that reads a program (such as the test programs on the web page) and prints out the AST.

**The Grammar**

The grammar is given in Extended Backus-Naur Form (EBNF). The meaning of the meta-symbols are as follows

<table>
<thead>
<tr>
<th>MetaSymbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>is defined to be</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[X]</td>
<td>0 or 1 instance of X</td>
</tr>
<tr>
<td>{X}</td>
<td>0 or more instances of X</td>
</tr>
<tr>
<td>(X</td>
<td>Y)</td>
</tr>
<tr>
<td>xyz</td>
<td>The terminal symbol <code>xyz</code></td>
</tr>
<tr>
<td>MetaIdentifier</td>
<td>The non-terminal symbol <code>MetaIdentifier</code></td>
</tr>
</tbody>
</table>

The grammar below is for a subset of Pascal, taken (and revised) from the Pascal User Manual and Report, 3rd edition, by K. Jensen & N. Wirth (Springer, 1985). The language has been substantially simplified by not allowing nested procedures.

The terminal symbols `id` and `Int` refer to the identifier and integer literal tokens returned by the lexer.
Program ::= Program id ; [ TypeDefinitions ] [ VariableDeclarations ]
    [ SubprogramDeclarations ] CompoundStatement .
TypeDefinitions ::= type TypeDefinition ; { TypeDefinition ; }  
VariableDeclarations ::= var VariableDeclaration ; { VariableDeclaration ; }  
SubprogramDeclarations ::= { ( ProcedureDeclaration | FunctionDeclaration ) ; }  
TypeDefinition ::= id = Type  
VariableDeclaration ::= IdentifierList : Type  
ProcedureDeclaration ::= Procedure id ( FormalParameterList ) ; ( Block | forward )  
FunctionDeclaration ::= function id ( FormalParameterList ) : ResultType ; ( Block | forward )  

FormalParameterList ::= [ IdentifierList : Type ; IdentifierList : Type ]  
Block ::= [ VariableDeclarations ] CompoundStatement  
CompoundStatement ::= begin StatementSequence end  
StatementSequence ::= Statement ; Statement  
Statement ::= SimpleStatement | StructuredStatement  
SimpleStatement ::= [ ( AssignmentStatement | ProcedureStatement ) ]  
AssignmentStatement ::= Variable := Expression  
ProcedureStatement ::= id ( ActualParameterList )  
StructuredStatement ::= CompoundStatement  
    | if Expression then Statement [ else Statement ]  
    | while Expression do Statement  
    | for id := Expression to Expression do Statement  
Type ::= id | array [ constant .. constant ] of Type | record FieldList end  
ResultType ::= id  
Fieldlist ::= [ IdentifierList : Type ; IdentifierList : Type ]  
Constant ::= [ sign ] int  
Expression ::= SimpleExpression [ RelationalOp SimpleExpression ]  
RelationalOp ::= < | <= | > | >= | <> | =  
SimpleExpression ::= [ sign ] Term { AddOp Term }  
AddOp ::= + | - | or  
Term ::= Factor { MulOp Factor }  
MulOp ::= * | div | mod | and  
Factor ::= int | string | Variable | FunctionReference | not Factor | ( Expression )  
FunctionReference ::= id ( ActualParameterList )  
Variable ::= id ComponentSelection
ComponentSelection ::= [ (. id ComponentSelection | [ expression ] ComponentSelection ) ]

ActualParameterList ::= [ Expression { , Expression } ]

IdentifierList ::= id { , id }

Sign ::= + | -

In a conditional, an **else** is associated with the nearest **if**... **then**.