QUESTION 1:

`program analysis`

CheckStyle is a static analysis tool for enforcing Java coding standards. Focus is on the more cosmetic aspects of coding,

e.g.:Non-empty Javadoc comments for all classes and members;
No * imports; Lines, function bodies; files not too long; { } used around even single-statement loop bodies and if-then-else bodies; Whitespace used uniformly.

Gprof is a profiling Tool for C/C++, part of the GNU compiler suite refer back to earlier lesson on statement and branch coverage. gprof is, essentially, the generalization of gcov. Profilers provide information on where a program is spending most of its execution time. May express measurements in Elapsed time and Number of executions. Measurement may be via Probes inserted into code or Statistical sampling of CPU program counter register.

`program synthesis`

Modelling tools are basically ‘model-based testing tools’ which actually generates test inputs or test cases from stored information about a particular model. It helps to validate models of the system or software. For users, tool can check consistency of data objects in a database and can find inconsistencies and defects when users used.

Antlr is a mature and widely-used parser generator for Java, and other languages as well. Parser generator is a program synthesis tool that user should make part of user toolbox. A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar.

`program translation`

Editpad is a code editor for Java. It can show users a detailed class tree of the Java source file they are editing. The File Navigator makes it easy to find relevant sections of code. By folding blocks users can hide lines to get a better overview of the file's structure, making it easier to rearrange blocks of code or navigate between them.

Java compiler is a compiler for the programming language Java. Compiler is a program that translates a source program written in Java language into machine code for some computer architecture, which users could use Java language to operate computer easily. The generated machine code can be later executed many times against different data each time.

QUESTION 2:

Note: a = VAR “A”, b = VAR “B” for a, b, c. I used Prop circuit program.

a) source input

`^AND a b`

based on correct AND a b, “^” is wrong. The reason is “^” is not valid symbols in Prop circuit.

`&OR a b`

based on correct OR a b, “&” is wrong. The reason is “&” is not valid symbols in Prop circuit.

Justify:

```
ERROR "PropScratch.lhs":28 - Syntax error in expression (unexpected symbol "^")
Prop> :r
ERROR "PropScratch.lhs":32 - Syntax error in expression (unexpected symbol "&")
Prop>
```

b) lexical analysis

`NTS a b`

based on correct AND a b, “NTS” is wrong. The reason is “NTS” is not defined in Prop circuit, which can’t use.

`OROR a b`

based on correct OR a b, “OROR” is wrong. The reason is “OROR” is not defined in Prop circuit, which can’t use.

Justify:

```
ERROR "PropScratch.lhs":28 - Undefined data constructor "NTS"
Prop> :r
ERROR "PropScratch.lhs":32 - Undefined data constructor "OROR"
Prop>
```

c) parsing

`AND a b AND`

based on correct AND a b, the second “AND” is wrong. The reason is there is same grammar defined in Prop circuit.

`OR a b OR`

based on correct OR a b, the second “OR” is wrong. The reason is there is same grammar defined in Prop circuit.

Justify:

```
Prop> :r
ERROR "PropScratch.lhs":28 - Type error in application
*** Expression     : AND a b AND
*** Term           : AND
*** Type           : Prop -> Prop -> Prop
*** Does not match : a -> b -> c -> d

Prop> :r
ERROR "PropScratch.lhs":32 - Type error in application
*** Expression     : OR a b OR
*** Term           : OR
*** Type           : Prop -> Prop -> Prop
*** Does not match : a -> b -> c -> d
```

d) static analysis

defined a = VAR “B”

“B” is wrong. The reason is ‘a’ = “B” is ambiguous by static analysis.

defined c = VAR “A B”

“D” is wrong. The reason is ‘a’ = “A B” is ambiguous by static analysis.

Justify:

```
[PropScratch> m1
 AND (VAR "B") (VAR "B")
[PropScratch> :r
[PropScratch> :r
[PropScratch> m1
 AND (VAR "A B") (VAR "B")
 PropScratch>
```

QUESTION 3:

generator

The address in the intermediate code at which the error occurred. Most errors of this type cause the code generator to display the line number and filename in error
The error will show when run out of memory during optimization.

optimizer

The error will show when connection timeout occurs
The error will show when the Optimizer service is down.

Most error handling occurs in the first three phases of the the front and middle (analysis stage). The scanner keeps an eye out for stray tokens, the syntax analysis phase reports invalid combinations of tokens, and the semantic analysis phase reports type errors and the like. Sometimes these are fatal errors that stop the entire process, while others are less serious and can be circumvented, so the compiler can continue.

Lexical errors are detected during the lexical analysis phase. Which include Exceeding length of identifier or numeric constants; Appearance of illegal characters; Unmatched string.

Syntactic phase errors are detected during syntax analysis phase. Which include Errors in structure; Missing operator; Misspelled keywords; Unbalanced parenthesis.

Semantic errors are detected during semantic analysis phase. Which include Incompatible type of operands; Undeclared variables; Not matching of actual arguments with formal one

QUESTION 4:

In Lex.lhs, I added TSEM in data Token for the symbol ‘;’ . I added TEQL in data Token for the symbol ‘=‘. Then, I implemented lexProp(‘;’ : cs) = TSEM : lexProp cs and lexProp(‘=’ : cs) = TEQL : lexProp in body of Lex.lhs.

Here Is my parseDefns code:

```
> parseDefns :: [Token] -> ([(String,Prop)],[Token])

> parseDefns ((TVAR s) : ts) = ((VAR p, VAR s), ts)
> parseDefns (TSEM : ts)
>   = case parseDefns ts of
>       (p, ts1) -> (p, ts1)

> parseDefns (TEQL : ts)
>   = case parseDefns ts of
>       (p, ts1) -> (p, ts1)

> parseProp (TSEM : ts)
>   = case parseProp ts of
>       (p, ts1) -> (p, ts1)
>       (p, ts2) -> error "missing semicolon!"

> parseDefns (TOPEN : ts)
>   = case parseDefns ts of
>       (p, TCLOSE : ts1) -> (p, ts1)
>       (p, ts2)    -> error " miss close"

> parseDefns (TNOT : ts)
>   = case parseDefns ts of
>       (p, ts1) -> (NOT p, ts1)


> parseDefns (TAND : ts)
>   = case parseDefns ts of
>       (p, ts1) -> case parseDefns ts1 of
>                     (q, ts2) -> (AND p q, ts2)
```

For symbol ‘;’ should be implement like TSEM Token -> Token
For symbol ‘=‘ should be implement like String TEQL Prop -> String “=“ Prop

For thoughtful testing, the correct result should like this:

Parse> parseDefns (lexProp "A = NOT; C = NOT D;")
Out:([(“A",NOT (VAR "C")),("C",NOT (VAR “D"))],[ ])

Parse> parseDefns (lexProp "A = NOT C; C = NOT D")
Out:([(“A",NOT (VAR "C"))
Program error: missing semicolon!
Parse> parseDefns (lexProp "B= A; B=B; NOT VALID")
([("B",VAR "A"),("B",VAR "B")],[TNOT,TVAR "VALID"])