Reminder - (E)BNF

A Notation for describing the grammar of a language.

- The notation consists of:
  - **Terminals** - the actual legal strings, written as is or inside quotes
  - **Nonterminals** - concepts of the language, written `<program>` or `program` or `program` in different variants
  - **Rules** - expanding a non-terminal to a series of nonterminals and terminals.

- One nonterminal is designated as the start of any derivation.

- A sequence of terminals not derivable from start symbol by rules of the grammar is illegal.
EBNF Syntax

- Each rule ends with a semicolon.
- Terminals are enclosed with quotes (single or double).
- Nonterminals written as-is.
- Special symbols (partial list):

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>rule definition</td>
</tr>
<tr>
<td>,</td>
<td>concatenation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[ .. ]</td>
<td>option</td>
</tr>
<tr>
<td>{ .. }</td>
<td>repetition (zero or more)</td>
</tr>
<tr>
<td>( .. )</td>
<td>grouping</td>
</tr>
</tbody>
</table>

Note: this is only one variation of EBNF syntax, there are other variations as well.
EBNF Example - Tiny Pascal

program = program-heading, block, ".";
program-heading = "program", identifier, ";";
block = declaration-part, statement-part;

declaration-part = [ variable-declaration-part ];
variable-declaration-part = "var", variable-declaration, ";", { variable-declaration, ";" };
variable-declaration = identifier, ":" , type;

statement-part = "begin", statement-sequence, "end";
statement-sequence = assignment, { ";", assignment };
assignment = identifier, "=" , (number | boolean-value);

identifier = letter, {letter};
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k"
     | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u"
     | "v" | "w" | "x" | "y" | "z";

type = "integer" | "boolean";
number = digit, {digit};
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
boolean-value = "true" | "false";
EBNF Example - Tiny Pascal

To check whether a program is legal we need to try to derive it from the start symbol (a.k.a create a syntax tree). Let's try to create a syntax tree for the following program:

```pascal
program test;
var x:integer;
begin
  x := 54
end.
```
program = program-heading, block, ".";
program-heading = "program", identifier, ";";
block = declaration-part, statement-part;

declaration-part = [ variable-declaration-part ];
variable-declaration-part = "var", variable-declaration, ";", (variable-declaration, ";")
variable-declaration = identifier, ";", type;

statement-part = "begin", statement-sequence, "end";
statement-sequence = assignment, (";", assignment);
assignment = identifier, "=",(number | boolean-value);

identifier = letter, (letter):
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
| "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";

type = "integer" | "boolean";

number = digit, (digit);
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

boolean-value = "true" | "false";

program test;
var x:integer;
begin
  x := 54
end.

Is the following program legal?

program test;
var x : integer;
begin
  x := true
end.
Ambiguity in grammars

Given the following EBNF:

```plaintext
expr = expr, op, expr | number;
op = “+” | “-” | “*” | “/”;
number = “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” | “0”;
```

What is the syntax tree of the following expression?

```
1+2*3
```
Defining EBNF using EBNF

grammar = { rule } ;
rule = lhs , "=" , rhs , ";" ;

lhs = identifier ;
rhs = identifier |
    terminal |
    "[" , rhs , "]" |
    "{" , rhs , "}" |
    "(" , rhs , ")" |
    rhs , "|" , rhs |
    rhs , "|" , rhs ;

identifier = letter , { letter | digit | "_" } ;
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
    "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

terminal = "'" , character , { character } , "'" | "'" , character , { character } , "'" ;
character = letter | digit | "-" | "_" ;
Regular Expressions

- A sequence of characters that forms a search pattern
- Mainly used in pattern matching with strings
- Some PLs have built-in support for regular expressions and some use a standard library

- Implementations of regular expression functionality is often called a regular expression engine
Regular Expressions with grep

```
$ echo hello | grep -o ell
ell
$ grep -o Whatever myfile
Whatever
Whatever
What
```

- `-o` means "only matching"
- `-E` - extended regexp syntax.
- `-P` - PCRE syntax (exponential complexity!)
- `grep` works line-by-line (usually)
Regular Expressions with sed

$ echo hello, world. | sed -r -e 's/hello/goodbye/'
goodbye, world.

- **sed** is short for stream editor
- Powerful tool, but we care just about the 's/regex/replacement/' command for today.
- **-r** - extended regex syntax
- **-e THING** - execute **THING** as a command
- **s/one/two/g** - replace globally, i.e. more than once.
Regexp Syntax

<table>
<thead>
<tr>
<th></th>
<th>Matches any character (except a newline, usually).</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches 0 or more repetitions of the preceding sub-regexp (greedy).</td>
</tr>
<tr>
<td>+</td>
<td>Matches 1 or more repetitions of the preceding sub-regexp (greedy).</td>
</tr>
<tr>
<td>?</td>
<td>Matches 0 or 1 repetitions of the preceding sub-regexp (greedy).</td>
</tr>
<tr>
<td>{m}</td>
<td>Matches exactly (m) repetitions of the previous sub-regexp</td>
</tr>
<tr>
<td>{m, n}</td>
<td>Matches from (m) to (n) repetitions of the preceding sub-regexp (greedy).</td>
</tr>
</tbody>
</table>
## Regexp Syntax

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Escape character.</td>
</tr>
<tr>
<td>[]</td>
<td>Character class. Used to indicate a set of characters.</td>
</tr>
<tr>
<td></td>
<td>[sc] will match ‘s’ or ‘c’</td>
</tr>
<tr>
<td></td>
<td>[a-z] will match all characters between a and z</td>
</tr>
<tr>
<td></td>
<td>[^sc] will match any character except ‘s’ and ‘c’</td>
</tr>
<tr>
<td></td>
<td>Alternation (or)</td>
</tr>
<tr>
<td>( . . . )</td>
<td>Match group. Allows recalling whatever was matched inside the parentheses later.</td>
</tr>
<tr>
<td>^</td>
<td>Start of line.</td>
</tr>
<tr>
<td>$</td>
<td>End of line.</td>
</tr>
</tbody>
</table>
Extract emails

Given the following myfile.txt, extract things that look sort-of like email addresses:

```html
<html>
... 
<a href="mailto:bruce@gmail.com">send mail to bruce</a>
... 
<a href="mailto:lee@yahoo.com">send mail to lee</a>
... 
</html>
```
Solution: extract email addresses

Extract all email addresses:
bruce@gmail.com
lee@yahoo.com

Extract all domains of the email addresses:
gmail.com
yahoo.com
Example: extract phone numbers

$ cat phone_numbers.txt
Tal: 04-8294342, room 198
Dan: 04 8298888 room 745
Chen: 0523682930, room 002
Eugene: 97243453455, room 789

$ grep -E -o "(972[0-9]|[0-9]{2,3})\([- ]?[0-9]{7}\)" phone_numbers.txt
04-8294342
04 8298888
0523682930
97243453455
Question from exam