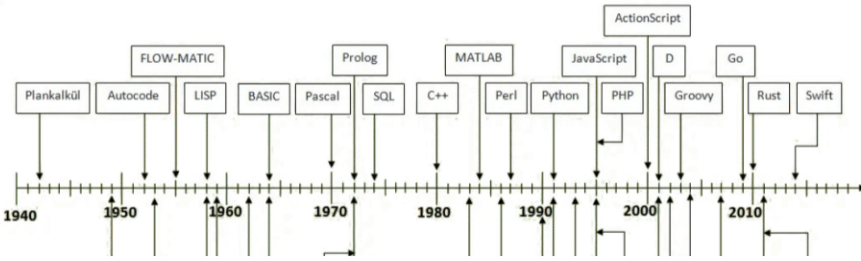
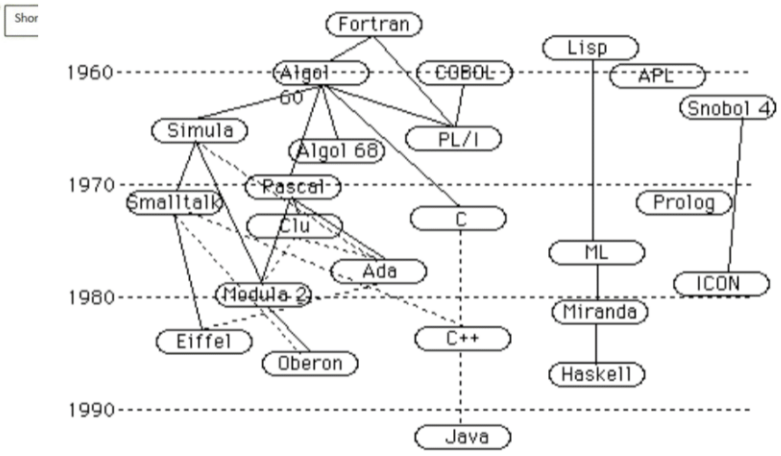
Programming Language History

- Compilers translate from source lang to target lang
- Source language “high-level” language
- Compilers also source-to-source and decompilation (low -> high-level)
- Interpreters similar to compilers but execute commands
- Declarative languages focus on “what” to do (high-level, implementation freedom, some gap betw. alg and low-level implementation)
- Imperative languages focus on “how” (spans several abstraction levels, close enough to detail/implementation -> algorithms)

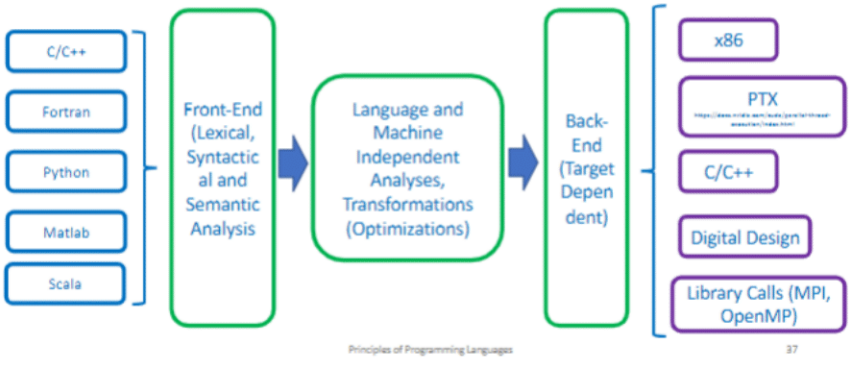


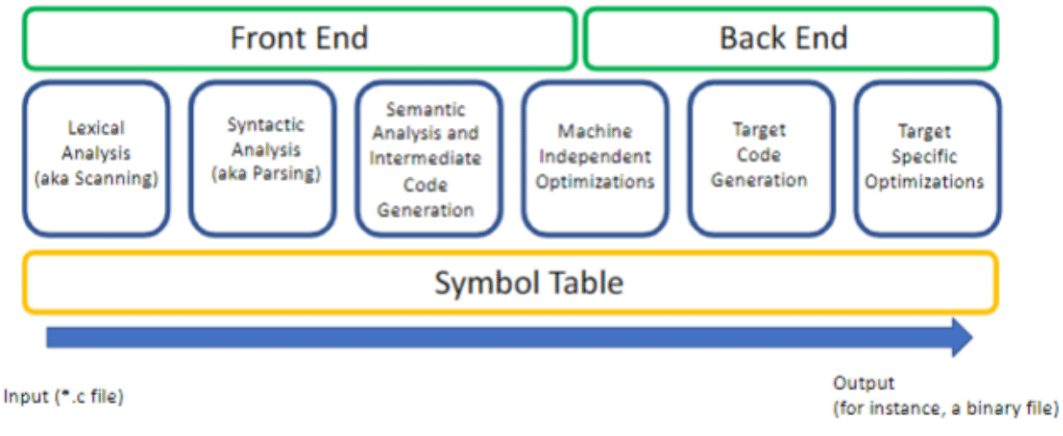


- Late 60s/early 70s: structured programming
  - GoTo-based control-flow (Fortran, Cobol, Basic) vs Explicit Control (loops, if/switch)
- Late 80s: nested block structure, precursor of Object-Oriented Progr.
  - Algol, Pascal, Ada
- 90s: Smalltalk, C++, Eiffel, Java
- Late 90s and 00s: scripting languages (PHP, Python, Ruby)
- Productivity (machine code -> assembly, writing shorter/more compact (array slices), reusability, maintainability, structures, functions, objects)
- New areas/domains (classical scientific computations (doing computer simulations), web, circuit/hardware design, quantum)

Compilation Overview

- Front End
  - In: Program in Source Lang (c file) Out: Abstract Syntax Tree
  - Source -> Token Stream -> AST
- Middle End
  - In: AST, Out: Intermediate Representation (IR)
  - AST -> IR -> IR (optimizations)
- Back End
  - In: IR, Out: Target Assembly (ASM)
  - IR -> ASM -> ASM (optimizations)





NIX Crash Course: Basic UNIX/LINUX command; ~ means home directory

DIRECTORIES

Display path of current working directory
$ pwd

Change directory to <directory>
$ cd <directory>

Navigate to parent directory
$ cd ..

List directory contents
$ ls

List detailed directory contents, including hidden files
$ ls -la

Create new directory named <directory>
$ mkdir <directory>

OUTPUT

Output the contents of <file>
$ cat <file>

Output the contents of <file> using the less command (which supports pagination etc.)
$ less <file>

Output the first 10 lines of <file>
$ head <file>

Direct the output of <cmd> into <file>
$ <cmd> > <file>

Append the output of <cmd> to <file>
$ <cmd> >> <file>

Direct the output of <cmd1> to <cmd2>
$ <cmd1> | <cmd2>

Clear the command line window
$ clear

FILES

Delete <file>
$ rm <file>

Delete <directory>
$ rm -r <directory>

Force-delete <file> (add -r to force-delete a directory)
$ rm -f <file>

Rename <file-old> to <file-new>
$ mv <file-old> <file-new>

Move <file> to <directory> (possibly overwriting an existing file)
$ mv <file> <directory>

Copy <file> to <directory> (possibly overwriting an existing file)
$ cp <file> <directory>

Copy <directory1> and its contents to <directory2> (possibly overwriting files in an existing directory)
$ cp -r <directory1> <directory2>

Update file access & modification time (and create <file> if it doesn't exis
$ touch <file>

PERMISSIONS

Change permissions of <file> to 755
$ chmod 755 <file>

Change permissions of <directory> (and its contents) to 600
$ chmod -R 600 <directory>

Change ownership of <file> to <user> and <group> (add -R to include a directory's contents)
$ chown <user>:<group> <file>

SEARCH

Find all files named <file> inside <dir> (use wildcards [*] to search for parts of filenames, e.g. "file.*")
$ find <dir> -name "<file>"

Output all occurrences of <text> inside <file> (add -i for case-insensitivity)
$ grep "<text>" <file>

Search for all files containing <text> inside <dir>
$ grep -rl "<text>" <dir>

NETWORK

Ping <host> and display status
$ ping <host>

Output whois information for <domain>
$ whois <domain>

Download <file> (via HTTP[S] or FTP)
$ curl -O <url/to/file>

Establish an SSH connection to <host> with user <username>
$ ssh <username>@<host>

Copy <file> to a remote <host>
$ scp <file> <user>@<host>:/remote/path

PROCESSES

Output currently running processes
$ ps ax

Display live information about currently running processes
$ top

Quit process with ID <pid>
$ kill <pid>

Special Characters in Regular Expressions & their meanings

Character	Meaning	Example
*	Match **zero, one or more** of the previous	`Ah*` matches "Ahhhhh" or "A"
?	Match **zero or one** of the previous	`Ah?` matches "Ah" or "Ah"
+	Match **one or more** of the previous	`Ah+` matches "Ah" or "Ahhh" but not "A"
\	Used to **escape** a special character	`Hungry\?` matches "Hungry?"
.	Wildcard character, matches **any** character	`do.*` matches "dog", "door", "dot", etc.
( )	**Group** characters	See example for |
[ ]	Matches a **range** of characters	`[cbf]ar` matches "car", "bar", or "far" `[0-9]+` matches any positive integer `[a-zA-z]` matches ascii letters a-z (uppercase and lower case) `[^0-9]` matches any character not 0-9.
|	Matche previous **OR** next character/group	`(Mon)|(Tues)day` matches "Monday" or "Tuesday"
{ }	Matches a specified **number of occurrences** of the previous	`[0-9]{3}` matches "315" but not "31" `[0-9]{2,4}` matches "12", "123", and "1234" `[0-9]{2,}` matches "1234567..."
^	**Beginning** of a string. Or within a character range `[]` negation.	`^http` matches strings that begin with http, such as a url. `[^0-9]` matches any character not 0-9.
$	**End** of a string.	`ing$` matches "exciting" but not "ingenious"

Lexical Analysis (including regular expression, remember how it is used in Group Project Part 1)
Input: String stream (think whatever you type as code)
Output: Token stream containing different tokens, ex: + -> T_ADD, if -> T_IF

- Ignore white space/tab, assign token (by scanner)
- Stores the actual values of some strings (identifiers, numbers, literal strings)
- Records source location info (file, line number, column, etc) for error reporting
- Regular Expressions
    - Character, empty string, two REs concatenated, two REs separated by |, Re followed by Kleene star * (concatenation of zero or more strings)
    - Scanner recognizes identifiers (i, my_sum, etc), key words (special case identifiers, for while if etc), numbers (integers and floating point), operators, other (,) [,] {,} ;
    - ADD INFO ABOUT DFA AND NFA
    - Lexeme - “unit” of a program, example: {if, else, 3.14}

Rules to recognize numbers:

- number → integer | real
- integer → digit digit *
- real → integer exponent | decimal (exponent | e)
- decimal → digit * (. digit | digit .) digit *
- exponent → (e | E) (+ | - | e) integer
- digit → [0 – 9]

Syntactic Analysis
The next step after lexical analysis, Syntactic Analysis converts a token stream into a tree according to multiple “grammar” rules

- PARSE TREE
- Terminal vs Non-Terminal symbols:
    - Terminal can be in the <u>final</u> output of the parse tree, non-terminal will eventually be changed through the grammar into something else

QUIZ 1

Question 1

0.5 / 0.5

Consider the following code fragment, which attempts to call the square root function:

double result = sqrt(3.14;

Clearly, the above statement will not compile in C/C++. What compilation phase would be responsible of detecting and reporting the problem?

- Lexical Analysis
- Syntactic Analysis
- Semantic Analysis
- Intermediate Code Generation

Question 2

0.5 / 0.5

Programming language XYZ doesn't allow variable names to use the underscore '_' character.

What compiler component would need to be modified to permit the use of the underscore character in identifiers?

- Syntactic Analyzer (aka Parser)
- Lexical Analyzer (aka Scanner)
- Semantic Analyzer
- Intermediate Code Generator

Question 3

1 / 1 pts

Program Y uses the following statement to compute the cube of a:

a_cube = a * a * a;

Program Y is compiled for execution on processors (machines) Z and W.

Processor Z has a native (built in) power instruction that makes it more efficient (i.e. faster) to compute the cube of a, like so:

a_cube = z_processor_power (a,3); // compute a*a*a

Processor W also has a similar built in instruction, but which becomes "profitable" to be used for powers greater than 3 like "a * a * a *a".

The decision to replace (or not) expression "a*a*a* by the call to the native power function would normally be performed in what compilation phase? Choose only one.

- Intermediate code generation
- Semantic Analysis
- Machine Independent Optimization
- Target Specific Optimizations

Question 4

Sort the compilation phases by choosing their order in a standard compilation pipeline:

Target Code Generation	5
Syntactic Analysis	2
Target Specific Optimization	6
Lexical Analysis	1
Machine Independent Optimization	4
Semantic Analysis and Intermediate Code Generation	3

Question 5

0.5 /

Which one of these programming language is considered the first "high level" programming language?

- C
- COBOL
- FORTRAN

Question 6

What is the output of lexical analyzer?

- A set of RE
- Syntax Tree
- Set of Tokens

Question 7

To connect to OU's network from anywhere, we use Global Protect VPN. A VPN ____

- Stands for Virtual Private Network
- Is used to apppear as though your physically on a network your not
- s a bullet-proof security measure
- A and B
- All of the above

Question 8 0.5 / 0.5 pts

If I'm on a Linux system on the account User1, then I can get back to my home directory from anywhere by typing

- cd /..
- cd /home
- goto home
- cd /User1
- ✓ cd /~

Question 9 0.5 / 0.5 pts

Which of the following is a valid command to compile hello.java Java code using CLI.

- java hello.java -out hello
- java compile hello.java
- java hello.java
- ✓ javac hello.java

Question 10 0.5 / 0.5 pts

Which of the following Linux commands can I use to get from directory home/homework/hw5/wannaCry to home/homework/viruses/dataDumper

- cd /homework/viruses/dataDumper
- cd ../../viruses/dataDumper
- cd ~/homework/viruses/dataDumper
- a and b
- ✓ b and c

Question 11 0.5 / 0.5 pts

Which of the following commands creates a new text file in Linux

- new text book
- newFile book.txt
- mkFile book.txt
- ✓ touch book.txt

Question 12 0.5 / 0.5 pts

Which of the following are Lexemes?

- Identifiers
- Constants
- Keywords
- ✓ All of the mentioned

Question 13

When expression sum=3+2 is tokenized then what is the token category of 3

- ✓ Integer Literal

Question 14

The lexical analyzer takes _____ as input and produces a stream of _____ as output.

- ✓ Source program, tokens

Question 15 0.5 / 0.5 pts

What goes over the characters of the lexeme to produce a value?

- ✓ Scanner

Exercise 2.17 from the book "Programming Language Pragmatics 4th ed" Chapter 2

1. *program* ⟶ *stmt_list* $$
2. *stmt_list* ⟶ *stmt_list stmt*
3. *stmt_list* ⟶ *stmt*
4. *stmt* ⟶ id := *expr*
5. *stmt* ⟶ read id
6. *stmt* ⟶ write *expr*
7. *expr* ⟶ *term*
8. *expr* ⟶ *expr add_op term*
9. *term* ⟶ *factor*
10. *term* ⟶ *term mult_op factor*
11. *factor* ⟶ (*expr*)
12. *factor* ⟶ id
13. *factor* ⟶ number
14. *add_op* ⟶ +
15. *add_op* ⟶ -
16. *mult_op* ⟶ *
17. *mult_op* ⟶ /

Figure 2.25 LR(1) grammar for the calculator language.

Extend the grammar of Figure 2.25 to include `if` statements and `while` loops, along the lines suggested by the following examples:

```
abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum
```

Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should also allow an arbitrary number of statements in the body of an `if` or `while` statement.

QUIZ 2

Consider the following grammar:

1) V → V [expr]
2) V → ID
3) expr → NUM
4) expr → V

From the following three derivation sequences, select the one that corresponds to a top-down, right-most derivation of the string "ID [NUM] [ID]" :

Derivation A:

V
V [expr]
V [expr] [expr]
ID [expr] [expr]
ID [NUM] [expr]
ID [NUM] [ID]

Derivation B:

V
V [expr]
V [ID]
V [expr] [ID]
V [NUM] [ID]
ID [NUM] [ID]

Derivation C:

V
V [expr]
V [expr] [expr]
V [NUM] [expr]
V [NUM] [ID]
ID [NUM] [ID]

- Derivation C
- **Derivation B**
- Derivation A

1) V → V [expr]
2) V → ID
3) expr → NUM
4) expr → V

From the following three derivation sequences, select the one that corresponds to a top-down, left-most derivation of the string "ID [ID [NUM]]" :

Derivation A:

V
V [expr]
V [V]
V [V [expr]]
V [V [NUM]]
V [ID [NUM]]
ID [ID [NUM]]

Derivation B:

V
V [expr]
V [V]
V [V [expr]]
V [ID [expr]]
ID [ID [expr]]
ID [ID [NUM]]

Derivation C:

V
V [expr]
ID [expr]
ID [V]
ID [V [expr]]
ID [ID [expr]]
ID [ID [NUM]]

- Derivation A
- Derivation B
- **Derivation C**

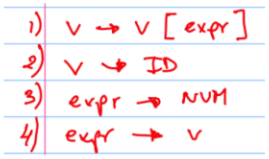
Possible solution for if statement

- 18: stmt -> if_stmt
- 19: if_stmt -> if l_expr then stmt fi
- 20: l_expr -> expr l_op expr
- 21: l_op -> < | > | == | != | <= | >=

Question 3

1 / 1 pts

Consider the following grammar:



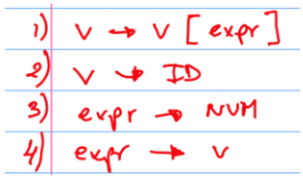
Match the concepts of left columns with their corresponding elements of the right column (NOTE: curly brackets and commas are not symbols of the grammar):

Terminal symbols	{ [,] , NUM, ID }
Non-terminal symbols	{ v, expr }
Recursion style for symbol v	Left-most recursion
Recurion style for symbol expr	Indirect recursion

Question 4

1 / 1 pts

Consider the following grammar:



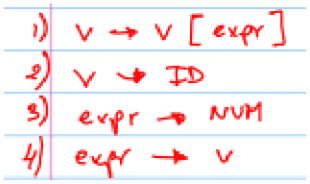
Possibly ignoring the convention that the non-terminal of the first production is the starting symbol of the grammar, choose the non-terminal that should be the starting symbol if the string "NUM" were to be accepted as a legal string:

- ○ v
- ○ Either (v or expr)
- ◉ expr

Question 6

1 / 1 pts

Consider the following grammar:



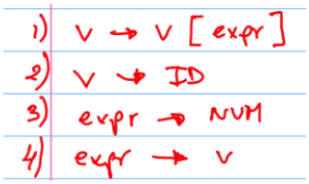
Can the string consisting of only "ID" be derived by the above grammar, assuming that expr is the starting symbol?

- ◉ Yes
- ○ No

Question 5

1 / 1 pts

Consider the following grammar:



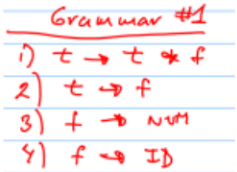
Can the string consisting of only "NUM" be derived from the above grammar, assuming that v is the starting symbol?

- ○ Yes
- ◉ No

Question 7

1 / 1 pts

Consider the following grammar:



Assume standard conventions for the above grammar, such as assuming which is its starting symbol.

Match the terms of the left column with the corresponding definitions:

Grammar starting symbol	t
Tokens	{ NUM, ID, * }
Non-terminal symbols	{ t, f }
Recursive style for production #1	Left recursion

Question 8

1 / 1 pts

Consider the following grammar:

R1) e -> e - t
R2) e -> t
R3) t -> NUM
R4) t -> - t

Assume that e is the starting symbol of the grammar.

The options below represent different derivation trees, producing potentially different sentential forms. Select the one corresponding to a top-down, right-most derivation for the string: NUM - - - NUM

- ○ R1, R4, R3, R4, R2, R3
- ○ R1, R4, R4, R3, R3, R2
- ◉ R1, R4, R4, R3, R2, R3
- ○ R1, R4, R3, R3, R2, R4

Question 9

1 / 1 pts

Match the following:

(P) Syntactic Analysis	(i) Leftmost derivation
(Q) LR	(ii) Regular expressions
(R) Top-down Parsing	(iii) Context Free Grammar
(S) Lexical Analysis	(iv) Rightmost derivation in reverse

- P - iv, Q - i, R - iii, S - ii
- P - iii, Q - ii, R - iv, S - i
- **P - iii, Q - iv, R - i, S - ii**
- P - iv, Q - iii, R - ii, S - i

Question 10

1 / 1 pts

In Lexical analysis, the input is -------, and the output is -------. While in syntactic analysis, the input is -------- and the output is -------.

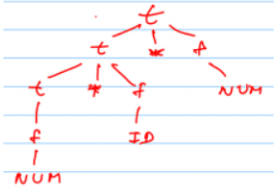
- syntax tree, character stream, token stream and token stream
- code generator, token stream, syntax tree and token stream
- **character stream, token stream, token stream and syntax tree**
- syntax tree, token stream, syntax tree and token stream

Practice quiz #2

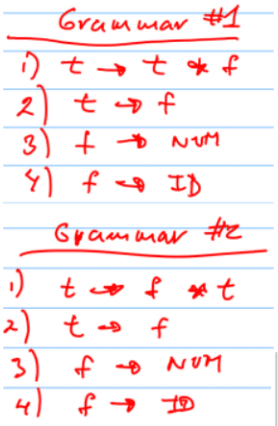
Question 1

1 / 1 pts

Consider the following parsing tree:



From the two grammars below, choose the one that would produce the above parsing tree given the string "NUM * ID * NUM"



- **Grammar #1**
- Grammar #2

Consider the following simple grammar for arithmetic expressions:

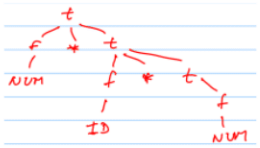
R1) e -> e + t
R2) e -> t
R3) t -> t * f
R4) t -> f
R5) f -> NUM
R6) f -> ID

Match the sentential forms provided in the left column to match the corresponding derivation order to parse the string: NUM * ID + ID

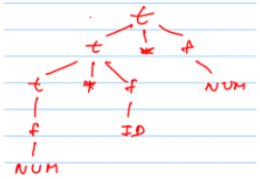
Assume that the string is produced by a left-most derivation and that the starting symbol of the grammar is e.

NUM * f + t	6
NUM * ID + t	7
t + t	3
t * f + t	4
NUM * ID + ID	9
e	1
e + t	2
NUM * ID + f	8
f * f + t	5

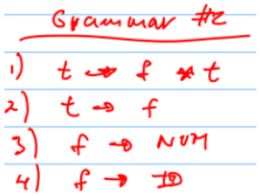
Parsing tree #1:



Parsing tree #2:



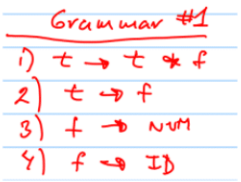
Which of the above trees would be produced for the string: NUM * ID * NUM given the grammar:



- Both
- Parsing Tree #2
- Neither
- **Parsing Tree #1**

Question 3

Consider the following grammar:



Match the terms of the left column with the corresponding definitions:

Grammar starting symbol	t
Tokens	{ NUM, ID, * }
Non-terminals	{ t, f }
Recursive style for production #1	Left recursion

Other Incorrect Match Options:
- Right recursion

Question 5

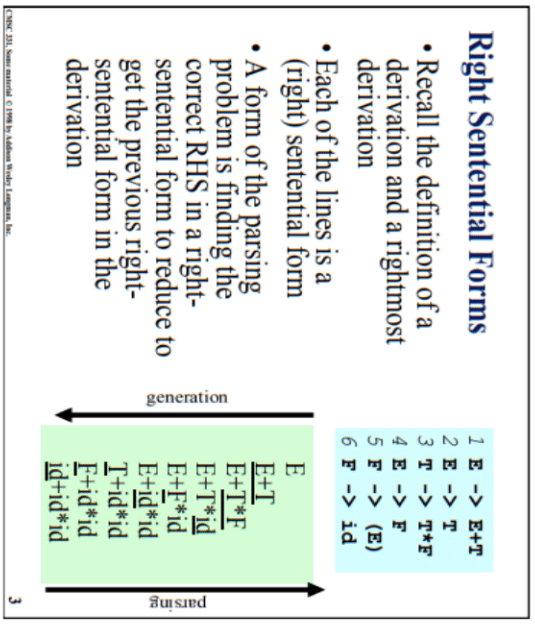
1.5 / 1.5 pts

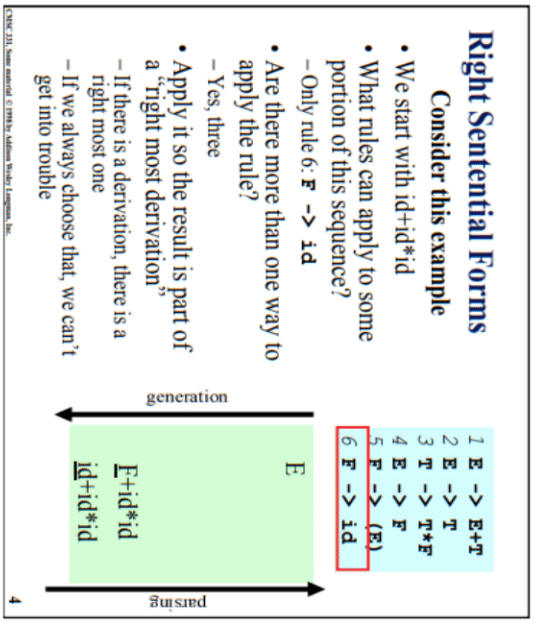
Consider the following grammar:

R1) e -> e - t
R2) e -> t
R3) t -> NUM
R4) t -> - t

The options below represent different derivation orders, producing potentially different sentential forms. Select the one corresponding to a right-most derivation for the string:
NUM - - - NUM

- (●) R1, R4, R4, R3, R2, R3
- ( ) R1, R4, R3, R3, R2, R4
- ( ) R1, R4, R4, R3, R3, R2
- ( ) R1, R4, R3, R4, R2, R3





Regex Examples

Try it at https://regex101.com

Input string	Regex	Match
aaabbbccc	ab	ab
aaabbbccc	a*b*	aaabbb
aaabbbccc	a*b*c*	aaabbbccc
aaabbb	a*b*c+	
aaabbbcc	a*b*c+	aaabbbcc
CS 3323 is cool	[0-9]	3 3 2 3
CS 3323 is cool	[0-9]{4}	3323
CS 3323 is cool	[0-9]{3}	332 323
CS 3323 is cool	[0-9]*	3323
CS 3323 is cool	[a-z]+	is cool
CS3323iscool	[a-zA-Z0-9_]+	CS3323iscool
CS3323iscool	\w+	CS3323iscool

Regex	Description	Possible matched strings
ab	match the exact sequence 'ab'	ab
a*b*	match strings that consist of zero or more 'a's followed by zero or more 'b's	a, b, ab, abb, aab, aaaaaabb, ...
a*b*c*	match strings starting with zero or more 'a's, followed by zero or more 'b's and end with zero or more 'c's	a, b, c, ab, ac, bc, abc, aaabbb, bcccc, ...
a*b*c+	match strings starting with zero or more 'a', followed by zero or more 'b' and end with zero or more 'c'	ac, bc, abc, aaaaabbbbc, abccccccc, ...
[0-9]	match any digit from '0' to '9'.	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
[0-9]{4}	match four occurrences of digits from '0' to '9'	1234, 5678, 1456, 7910, ...
[0-9]{2}	match two occurrences of digits from '0' to '9'	23, 12, 45, 89, ...
[0-9]*	match zero or more occurrences of digits from '0' to '9'	, 1, 5, 6, 89, 12345567
[0-9]+	match one or more occurrences of digits from '0' to '9'	1, 2, 3, 12, 123, 5678, ...
[a-z]+	match zero or more occurrences of lowercase letters from 'a' to 'z'.	a, aaaaa, abcded, fgh, z
[a-zA-Z0-9_]+	match strings that contain one or more alphanumeric characters (letters or digits) or underscores	C, A, B, ABC, CS3323iscool, _, A_B, ..
\w	match any single character that is a letter (either lowercase or uppercase), a digit, or an underscore	A, b, C, _, ...