Regular expressions

ComS 252 — Iowa State University

Andrew Miner

Overview

Remember `grep`?

- "Global, regular expression, print"
- Prints lines that contain a string that matches a pattern

Overview

Remember grep?

- "Global, regular expression, print"
- Prints lines that contain a string that matches a pattern

How do we specify a "pattern"?

Overview

Remember `grep`?

- "Global, regular expression, print"
- Prints lines that contain a string that matches a pattern

How do we specify a "pattern"?

- Write a regular expression

Overview 2

- Regular expressions specify patterns to match strings
- The idea is similar to shell globbing
- Unfortunately, special characters have different meanings
  - What does globbing pattern "`hello.*`" mean?

Overview 2

- Regular expressions specify patterns to match strings
- The idea is similar to shell globbing
- Unfortunately, special characters have different meanings
  - What does globbing pattern “`hello.*`” mean?
  - Regular expression “`hello.*`” is not the same thing

Overview 2

- Regular expressions specify patterns to match strings
- The idea is similar to shell globbing
- Unfortunately, special characters have different meanings
  - What does globbing pattern "`hello.*`" mean?
  - Regular expression "`hello.*`" is not the same thing
    - Worse — it is almost the same

Overview 2

- Regular expressions specify patterns to match strings
- The idea is similar to shell globbing
- Unfortunately, special characters have different meanings
  - What does globbing pattern "`hello.*`" mean?
  - Regular expression "`hello.*`" is not the same thing
    - Worse — it is almost the same
- Regular expressions ("regexes") are used in many places
  - E.g., `vim`, `grep` and friends, `sed`, `awk`, `perl`
- The syntax has been standardized by POSIX
- There are two types of regular expressions:
  1. Basic regular expressions (used by `grep`)
  2. Extended regular expressions (used by `egrep`)

Some theory

- The name comes from "regular languages"
  - Yes, the same ones from automata theory
  - See, ComS 331 is useful after all!

Some theory

- The name comes from “regular languages”
    - Yes, the same ones from automata theory
    - See, ComS 331 is useful after all!
- We will not cover automata theory here
- Take ComS 230 / 331 for this, and learn
    - Algorithms to decide if a string matches a regex pattern
    - Why regexes include some features but not others

Some theory

- The name comes from “regular languages”
  - Yes, the same ones from automata theory
  - See, ComS 331 is useful after all!
- We will not cover automata theory here
- Take ComS 230 / 331 for this, and learn
  - Algorithms to decide if a string matches a regex pattern
  - Why regexes include some features but not others
- We will discuss POSIX regex syntax

Characters in regular expressions

There are two types of characters in a regular expression

meta characters : have special meanings

- Things like: `.` `*` `[` `]` \
- We will cover these, gradually

ordinary characters : everything else

Some preliminary rules

Ordinary characters match themselves

- E.g., string “`A`” matches regex “`A`”
- E.g., string “`B`” does not match regex “`C`”

Some preliminary rules

Ordinary characters match themselves

- E.g., string “`A`” matches regex “`A`”
- E.g., string “`B`” does not match regex “`C`”

Regular expressions match as much as they can, but no more

- E.g., string “`Ab`” does not match regex “`A`”
- The “as much as they can” part will make more sense later

Concatenation rule

Concatenating regexes means concatenate strings that match

- More formally:
  If `stringA` *matches* `regexA`,
  and `stringB` *matches* `regexB`,
  then `stringAstringB` *matches* `regexAregexB`
- This is what you would expect
- E.g., string “`foobar`” matches regex “`foobar`”
- E.g., string “`foo`” does not match regex “`foobar`”
- E.g., string “`bar`” does not match regex “`foobar`”

Concatenation rule caveat

- I stated the concatenation rule to make it look easy
- But there is a subtle catch to this rule
- Suppose I concatenate two regular expressions
  - `regexA`
  - `regexB`
- Now, when does "`string`" match `regexAregexB`?

Concatenation rule caveat

- I stated the concatenation rule to make it look easy
- But there is a subtle catch to this rule
- Suppose I concatenate two regular expressions
  - `regexA`
  - `regexB`
- Now, when does "`string`" match `regexAregexB`?
- When there exists a way to split "`string`" such that
  1. The first "half" matches `regexA`
  2. The second "half" matches `regexB`

Concatenation rule caveat

- I stated the concatenation rule to make it look easy
- But there is a subtle catch to this rule
- Suppose I concatenate two regular expressions
  - `regexA`
  - `regexB`
- Now, when does "`string`" match `regexAregexB`?
- When there exists a way to split "`string`" such that
  1. The first "half" matches `regexA`
  2. The second "half" matches `regexB`
- There are lots of ways to split a string
  - Especially when we concatenate several regexes
- You might end up matching things you do not intend
- Take care with complex regular expressions

Example

Carefully checking a match

String "`foobar`" matches regex "`foobar`" because there exists a way to split string `foobar`:

- String "`f`" matches regex "`f`"
- String "`o`" matches regex "`o`"
- String "`o`" matches regex "`o`"
- String "`b`" matches regex "`b`"
- String "`a`" matches regex "`a`"
- String "`r`" matches regex "`r`"

Example

Carefully checking a match

String "`foobar`" matches regex "`foobar`" because there exists a way to split string `foobar`:

- String "`f`" matches regex "`f`"
- String "`o`" matches regex "`o`"
- String "`o`" matches regex "`o`"
- String "`b`" matches regex "`b`"
- String "`a`" matches regex "`a`"
- String "`r`" matches regex "`r`"

This is overkill for a simple example. But it is useful when we get to more complex rules.

Character classes

To match a single character from a list, use `[list]`

- Like shell globbing
- But regexes let you do more. . .
- E.g., string “`a`” matches regex “`[aeiou]`”
- E.g., string “`f`” does not match regex “`[EFL]`”

Character classes

To match a single character from a list, use `[list]`

- Like shell globbing
- But regexes let you do more. . .
- E.g., string “a” matches regex “`[aeiou]`”
- E.g., string “`f`” does not match regex “`[EFL]`”

Example

- Want to replace “folder” with “directory”
- Need to search for “folder” and “Folder”
- Can use regex:

Character classes

To match a single character from a list, use `[list]`

- Like shell globbing
- But regexes let you do more. . .
- E.g., string “a” matches regex “`[aeiou]`”
- E.g., string “`f`” does not match regex “`[EFL]`”

Example

- Want to replace “folder” with “directory”
- Need to search for “folder” and “Folder”
- Can use regex:

  ```
  [Ff]older
  ```

Ranges in character classes

Use “–” inside [] to specify a character range

- E.g., use regex “`[01]`” to match a binary digit
- E.g., use regex “`[0-7]`” to match an octal digit
- E.g., use regex “`[0-9]`” to match a decimal digit
- E.g., use regex “`[0-9a-f]`” to match a hexadecimal digit

Ranges in character classes

Use “–” inside [] to specify a character range

- E.g., use regex “[01]” to match a binary digit
- E.g., use regex “[0-7]” to match an octal digit
- E.g., use regex “[0-9]” to match a decimal digit
- E.g., use regex “[0-9a-f]” to match a hexadecimal digit

Use “–” first or last inside [] to specify “–”

- E.g., to match arithmetic operators, use “[-+*/]”

Example

How can we match *all* text of the form:

1. Single–character variable name
2. Space
3. Arithmetic operator
4. Space
5. Single–character variable name

Things like: $x + y$, $I - J$, ...

Example

How can we match *all* text of the form:

1. Single–character variable name
2. Space
3. Arithmetic operator
4. Space
5. Single–character variable name

Things like: $x + y$, $I - J$, . . .
Use regex: “`[a-zA-Z] [-+*/%] [a-zA-Z]`”

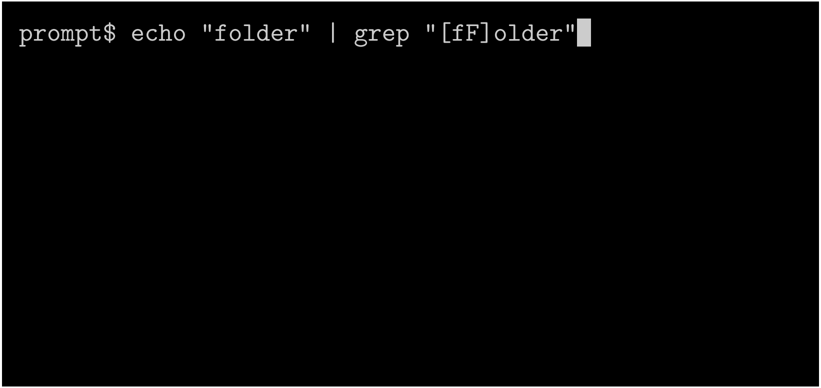
Testing these things

The easiest way is to send a string through `grep`:

```
prompt$ █
```

Testing these things

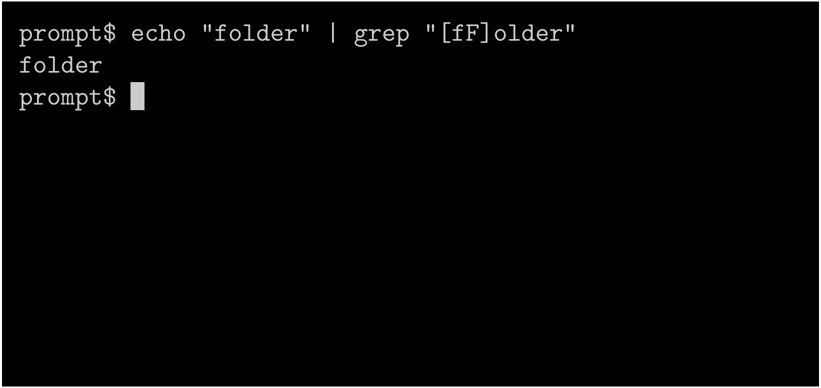
The easiest way is to send a string through `grep`:



```
prompt$ echo "folder" | grep "[fF]older"
```

Testing these things

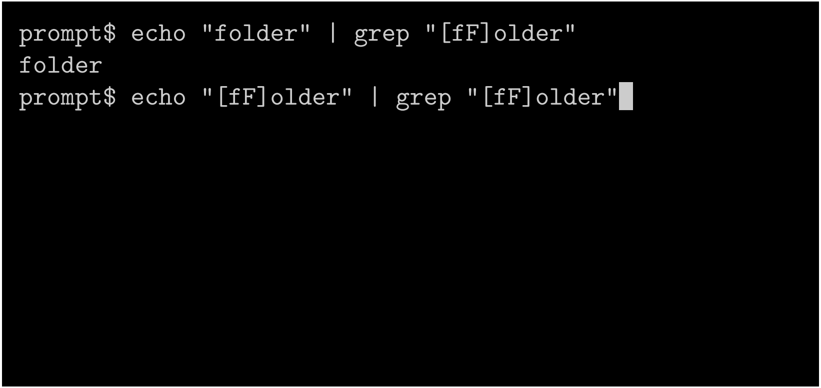
The easiest way is to send a string through `grep`:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ █
```

Testing these things

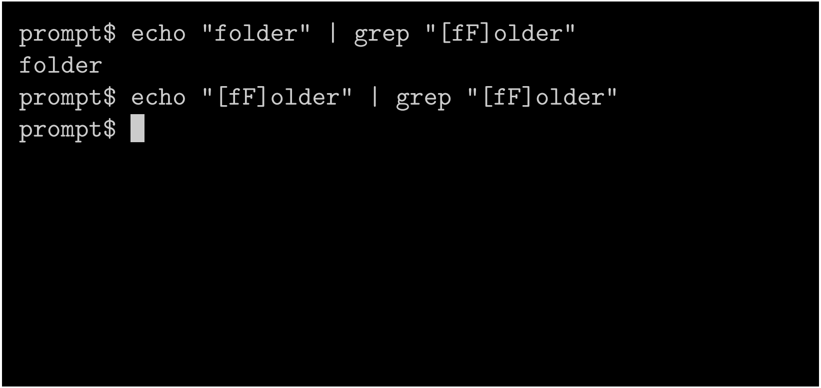
The easiest way is to send a string through grep:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
```

Testing these things

The easiest way is to send a string through `grep`:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ █
```

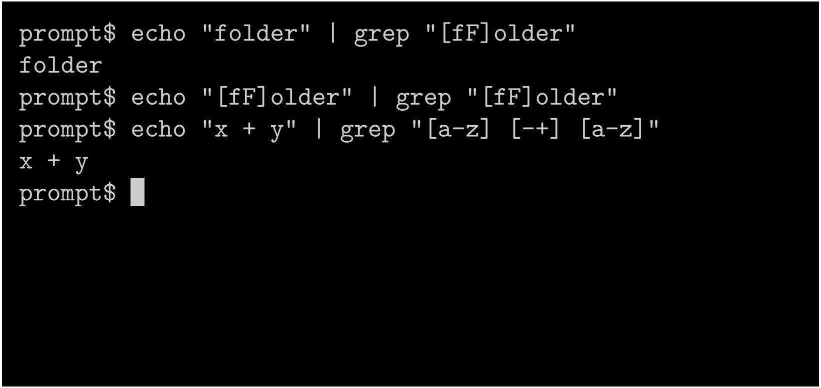
Testing these things

The easiest way is to send a string through grep:

```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
```

Testing these things

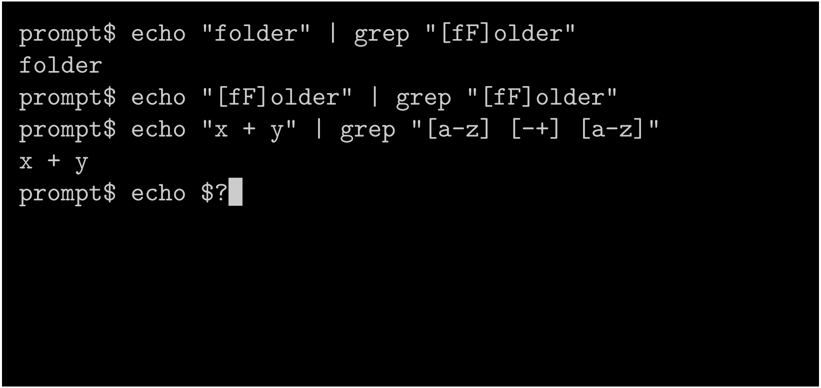
The easiest way is to send a string through grep:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
x + y
prompt$ █
```

Testing these things

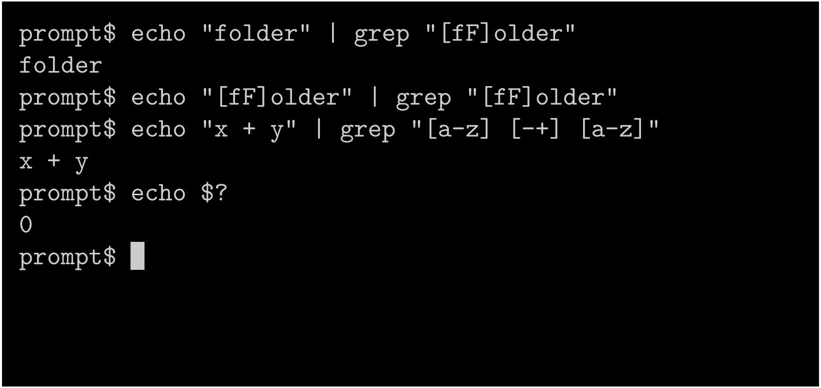
The easiest way is to send a string through `grep`:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
x + y
prompt$ echo $?
```

Testing these things

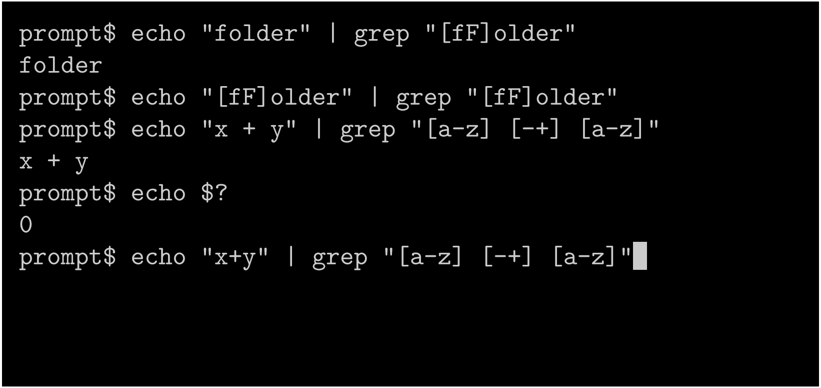
The easiest way is to send a string through grep:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
x + y
prompt$ echo $?
0
prompt$ █
```

Testing these things

The easiest way is to send a string through grep:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
x + y
prompt$ echo $?
0
prompt$ echo "x+y" | grep "[a-z] [-+] [a-z]"
```

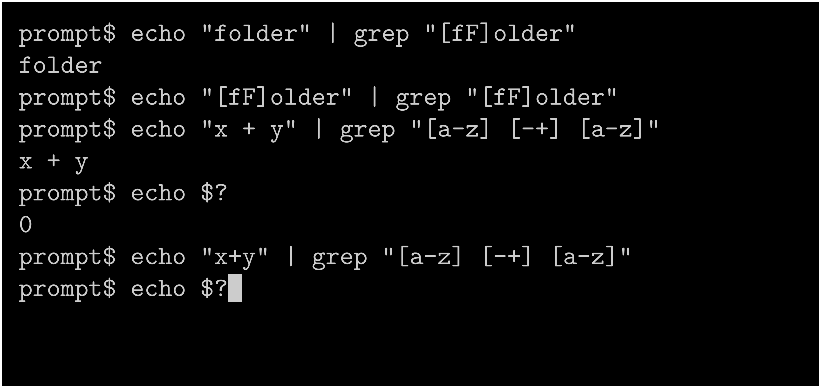
Testing these things

The easiest way is to send a string through grep:

```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
x + y
prompt$ echo $?
0
prompt$ echo "x+y" | grep "[a-z] [-+] [a-z]"
prompt$ █
```

Testing these things

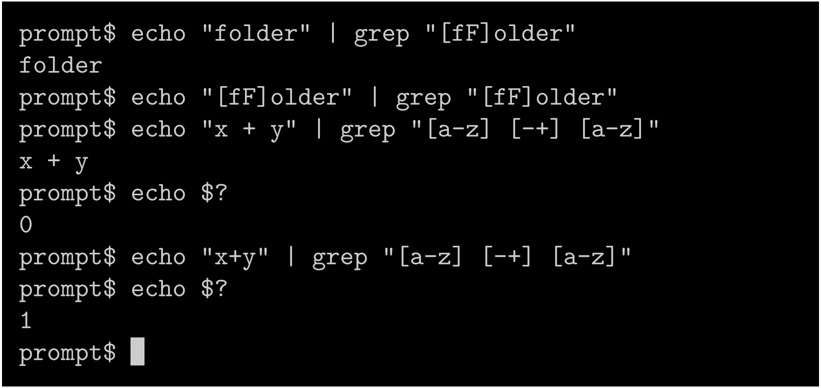
The easiest way is to send a string through grep:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
x + y
prompt$ echo $?
0
prompt$ echo "x+y" | grep "[a-z] [-+] [a-z]"
prompt$ echo $?
```

Testing these things

The easiest way is to send a string through grep:



```
prompt$ echo "folder" | grep "[fF]older"
folder
prompt$ echo "[fF]older" | grep "[fF]older"
prompt$ echo "x + y" | grep "[a-z] [-+] [a-z]"
x + y
prompt$ echo $?
0
prompt$ echo "x+y" | grep "[a-z] [-+] [a-z]"
prompt$ echo $?
1
prompt$ 
```

grep has a zero exit code if there was at least one matching line

Inverting a class

Use “`^`” as the first character in `[]` to invert the class

- E.g., use regex “`[^f]`” to match every character except `f`
- E.g., use regex “`[^aeiou]`” to match consonants

Inverting a class

Use “^” as the first character in [] to invert the class

- E.g., use regex “`[^f]`” to match every character except `f`
- E.g., use regex “`[^aeiou]`” to match consonants
  and digits, and control characters, and many other things
  but not lower–case vowels

Tricky ones

What do the following regular expressions match?

```
[^^]
```

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except “^”

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except "^"

`[^-z]`

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except “^”

`[^-z]`

- Any character between “^” and “z”
- Any character except “–” and “z”

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except “^”

`[^-z]`

- ~~Any character between “^” and “z”~~ NO
- Any character except “-” and “z”

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except “^”

`[^-z]`

- ~~Any character between “^” and “z”~~ NO
- Any character except “-” and “z”

How do we write: any character between “^” and “z”?

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except “^”

`[^-z]`

- ~~Any character between “^” and “z”~~ NO
- Any character except “-” and “z”

How do we write: any character between “^” and “z”?

“`[z^-z]`”

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except “^”

`[^-z]`

- ~~Any character between “^” and “z”~~ NO
- Any character except “-” and “z”

How do we write: any character between “^” and “z”?

“`[z^-z]`”

What is a *safer* way to write: any character except “-” and “z”?

Tricky ones

What do the following regular expressions match?

`[^^]`

- Any character except “^”

`[^-z]`

- ~~Any character between “^” and “z”~~ NO
- Any character except “–” and “z”

How do we write: any character between “^” and “z”?

“`[z^-z]`”

What is a *safer* way to write: any character except “–” and “z”?

“`[^z-]`”

Names for classes

POSIX defines names for classes of characters, including:

`[:alpha:]` : Alphabetic characters

Names for classes

POSIX defines names for classes of characters, including:

`[:alpha:]` : Alphabetic characters

- Isn't this the same as "`a-zA-Z`"?

Names for classes

POSIX defines names for classes of characters, including:

`[:alpha:]` : Alphabetic characters

- Isn't this the same as "`a-zA-Z`"?
- Not always: `[:alpha:]` may match "è"

Names for classes

POSIX defines names for classes of characters, including:

`[:alpha:]` : Alphabetic characters

- Isn't this the same as "`a-zA-Z`"?
- Not always: `[:alpha:]` may match "è"

`[:alnum:]` : Alphanumeric characters

`[:cntrl:]` : Control characters

`[:digit:]` : Numeric characters

`[:lower:]` : Lower–case characters

`[:punct:]` : Punctuation characters

`[:space:]` : Space, tab, and other "whitespace"

`[:upper:]` : Upper–case characters

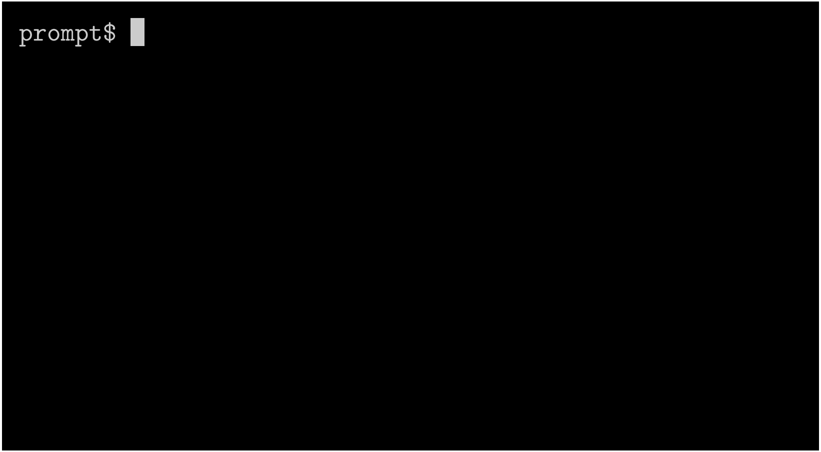
Names for classes

POSIX defines names for classes of characters, including:

`[:alpha:]`	:	Alphabetic characters ▶ Isn't this the same as "`a-zA-Z`"? ▶ Not always: `[:alpha:]` may match "è"
`[:alnum:]`	:	Alphanumeric characters
`[:cntrl:]`	:	Control characters
`[:digit:]`	:	Numeric characters
`[:lower:]`	:	Lower–case characters
`[:punct:]`	:	Punctuation characters
`[:space:]`	:	Space, tab, and other "whitespace"
`[:upper:]`	:	Upper–case characters

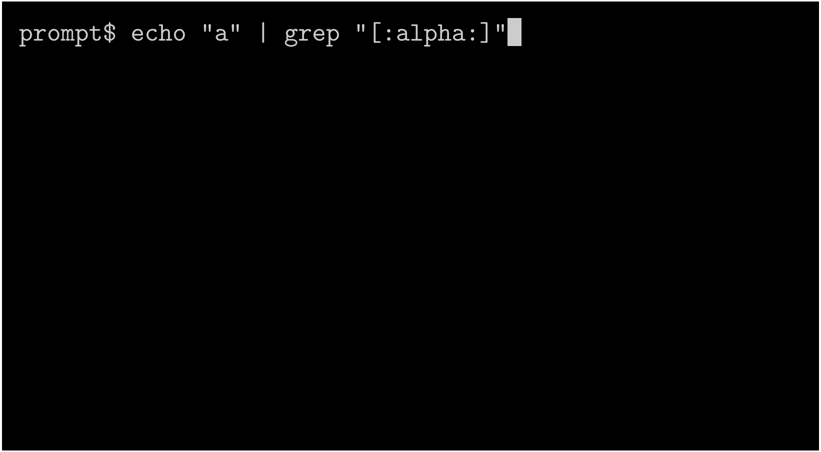
These must be used within `[]`

Example



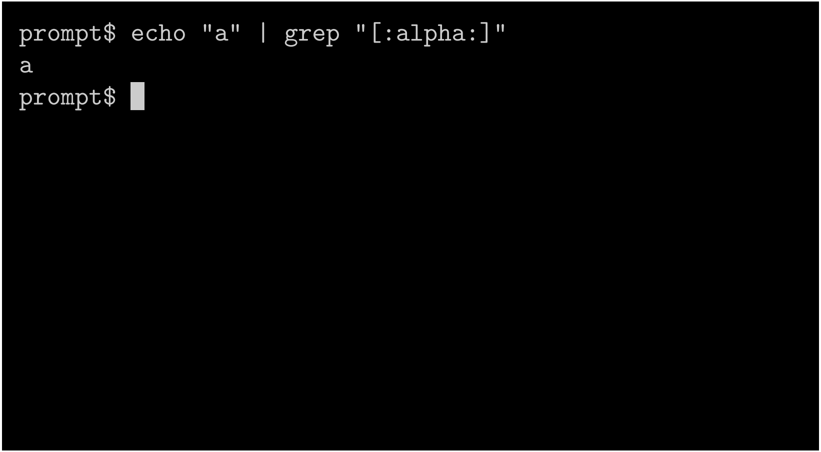
```
prompt$ █
```

Example



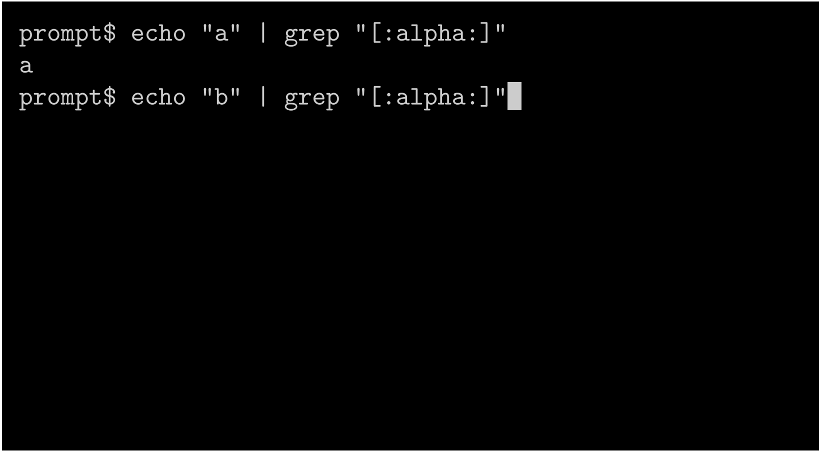
```
prompt$ echo "a" | grep "[:alpha:]"
```

Example



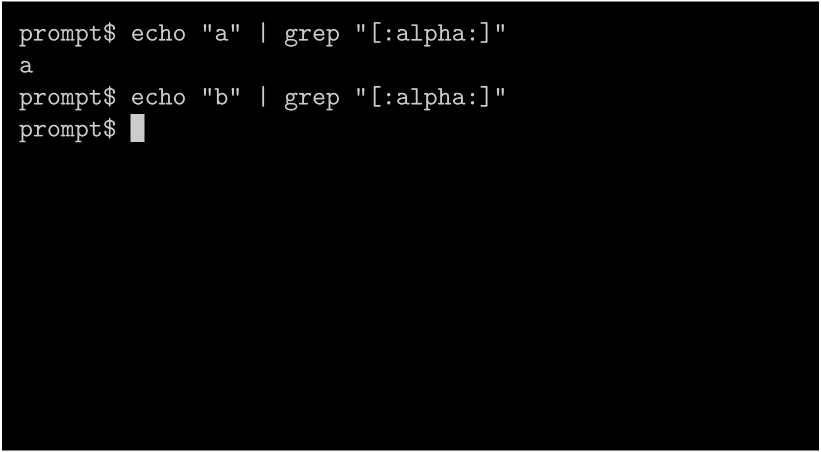
```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ █
```

Example



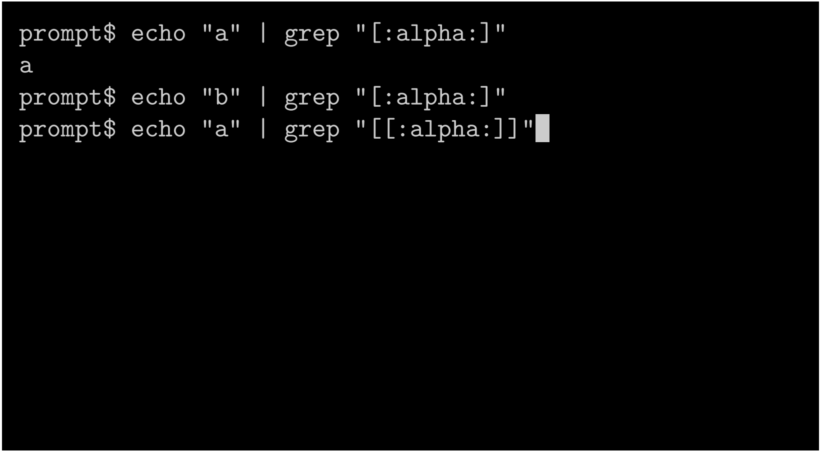
```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
```

Example



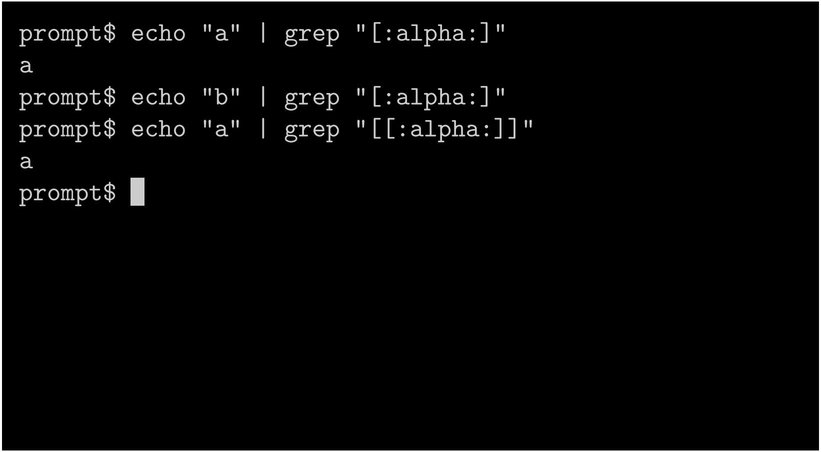
```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$
```

Example



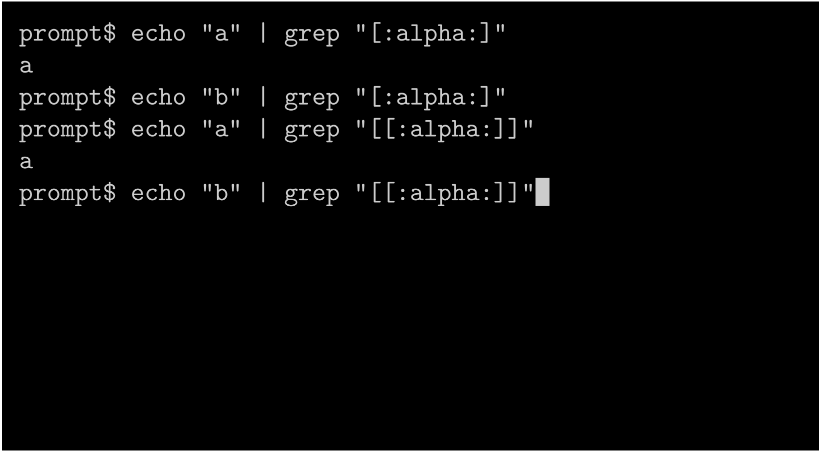
```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
```

Example



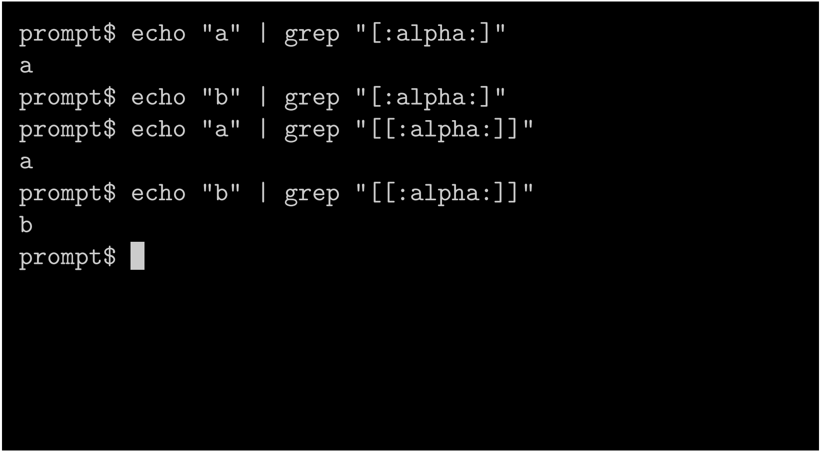
```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ █
```

Example



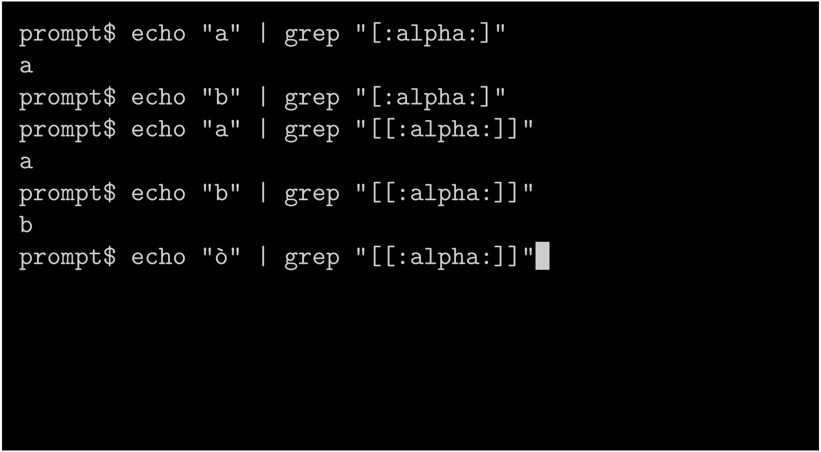
```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
```

Example



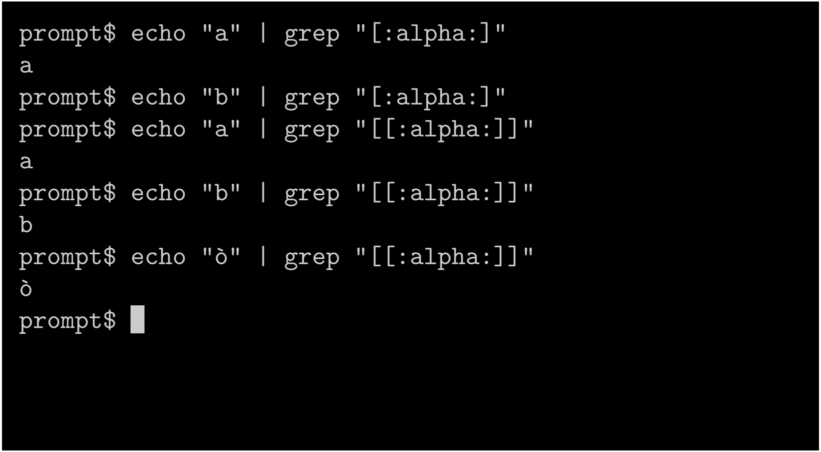
```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
b
prompt$ █
```

Example



```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
b
prompt$ echo "ò" | grep "[[:alpha:]]"
```

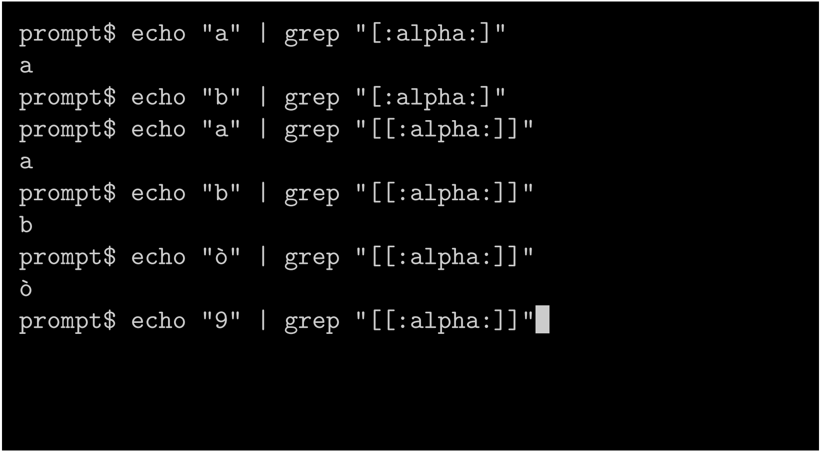
Example



```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
b
prompt$ echo "ò" | grep "[[:alpha:]]"
ò
prompt$ █
```

I tested this on a Macintosh, and used `Option-‘o` to get “ò”

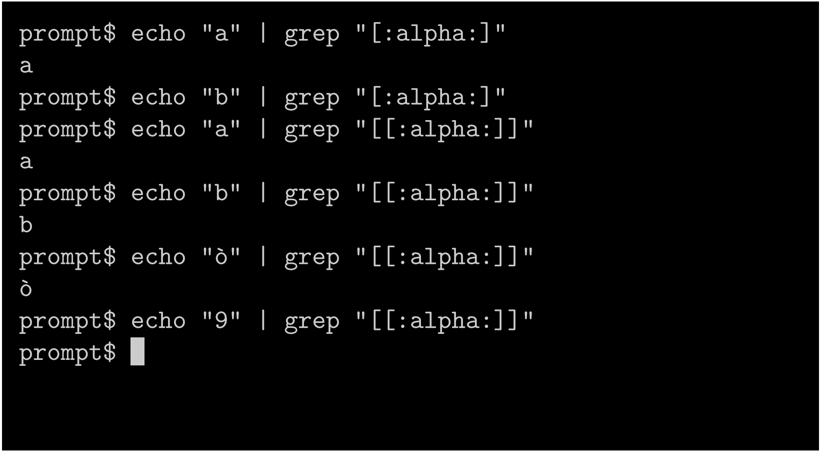
Example



```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
b
prompt$ echo "ò" | grep "[[:alpha:]]"
ò
prompt$ echo "9" | grep "[[:alpha:]]"
```

I tested this on a Macintosh, and used `Option-‘o` to get “ò”

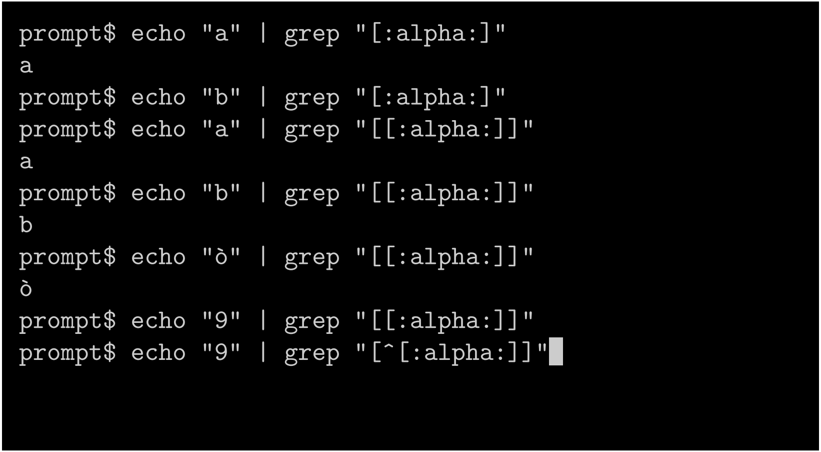
Example



```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
b
prompt$ echo "ò" | grep "[[:alpha:]]"
ò
prompt$ echo "9" | grep "[[:alpha:]]"
prompt$
```

I tested this on a Macintosh, and used `Option-‘o` to get “ò”

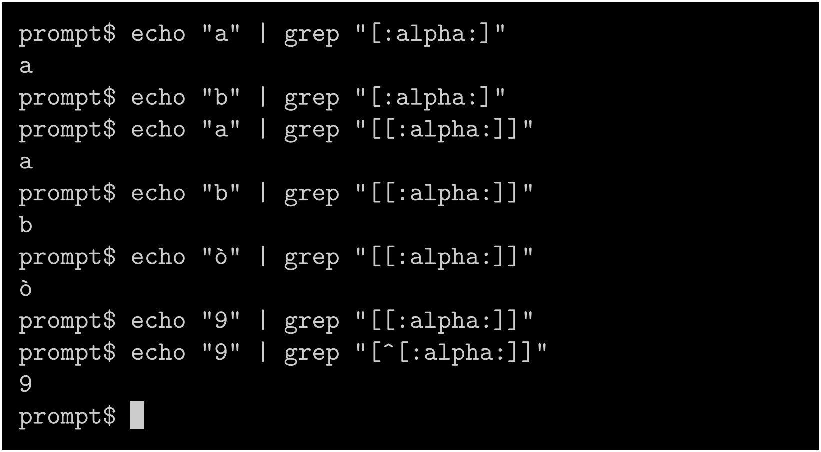
Example



```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
b
prompt$ echo "ò" | grep "[[:alpha:]]"
ò
prompt$ echo "9" | grep "[[:alpha:]]"
prompt$ echo "9" | grep "[^[:alpha:]]"
```

I tested this on a Macintosh, and used `Option-‘o` to get “ò”

Example



```
prompt$ echo "a" | grep "[:alpha:]"
a
prompt$ echo "b" | grep "[:alpha:]"
prompt$ echo "a" | grep "[[:alpha:]]"
a
prompt$ echo "b" | grep "[[:alpha:]]"
b
prompt$ echo "ò" | grep "[[:alpha:]]"
ò
prompt$ echo "9" | grep "[[:alpha:]]"
prompt$ echo "9" | grep "[^[:alpha:]]"
9
prompt$ █
```

I tested this on a Macintosh, and used `Option-‘o` to get “ò”

Some special characters

The "." character matches (almost) any single character

- The newline character usually does not match "."
- "." would be equivalent to "`[^<newline>]`"
  if you could specify the newline character in a class

```
prompt$ 
```

Some special characters

The "." character matches (almost) any single character

- The newline character usually does not match "."
- "." would be equivalent to "`[^<newline>]`"
  if you could specify the newline character in a class

```
prompt$ echo "ab" | grep "a."
```

Some special characters

The “.” character matches (almost) any single character

- The newline character usually does not match “.”
- “.” would be equivalent to “`[^<newline>]`”
  if you could specify the newline character in a class

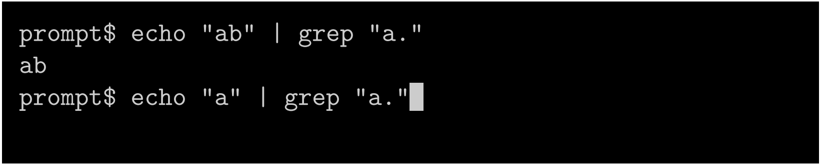
```
prompt$ echo "ab" | grep "a."
ab
prompt$ █
```

Some special characters

The “.” character matches (almost) any single character

- The newline character usually does not match “.”
- “.” would be equivalent to “`[^<newline>]`”

  if you could specify the newline character in a class



```
prompt$ echo "ab" | grep "a."
ab
prompt$ echo "a" | grep "a."█
```

Some special characters

The “.” character matches (almost) any single character

- The newline character usually does not match “.”
- “.” would be equivalent to “`[^<newline>]`”

  if you could specify the newline character in a class

```
prompt$ echo "ab" | grep "a."
ab
prompt$ echo "a" | grep "a."
prompt$ █
```

How to match an actual "." (only)?

How to match an actual "." (only)?

1. Use "`[.]`"

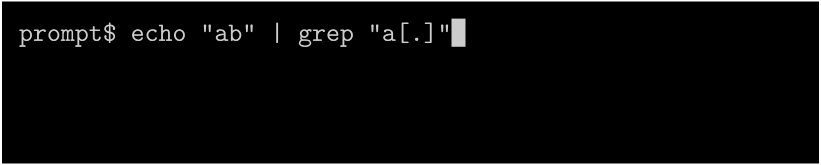
How to match an actual “.” (only)?

1. Use “`[.]`”

```
prompt$ █
```

How to match an actual “.” (only)?

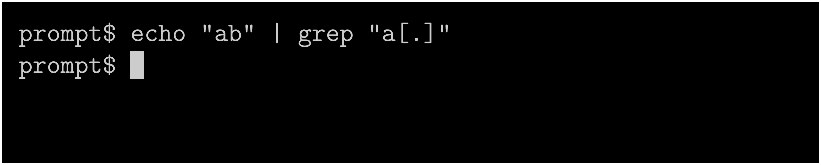
1. Use “[.]”



```
prompt$ echo "ab" | grep "a[.]"
```

How to match an actual “.” (only)?

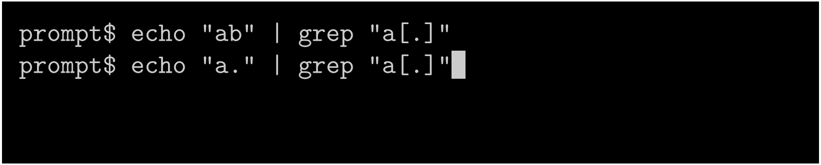
1. Use “`[.]`”



```
prompt$ echo "ab" | grep "a[.]"
prompt$ █
```

How to match an actual "." (only)?

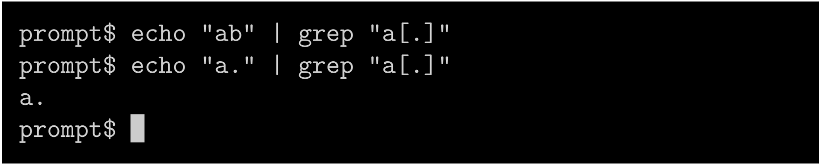
1. Use "[.]"



```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
```

How to match an actual “.” (only)?

1. Use “`[.]`”



```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
a.
prompt$ █
```

How to match an actual “.” (only)?

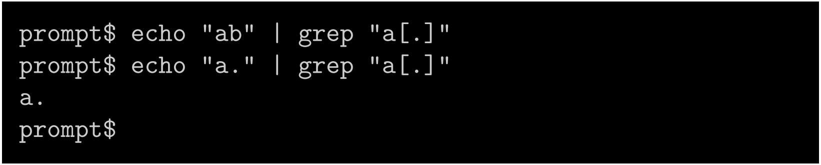
1. Use “[.]”

```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
a.
prompt$
```

2. Use “\.”

How to match an actual “.” (only)?

1. Use “[.]”



```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
a.
prompt$
```

2. Use “\.”

```
prompt$ 
```

How to match an actual “.” (only)?

1. Use “[.]”

```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
a.
prompt$
```

2. Use “\.”

```
prompt$ echo "ab" | grep "a\."
```

How to match an actual "." (only)?

1. Use "[.]"

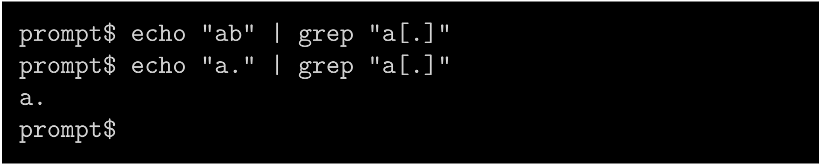
```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
a.
prompt$
```

2. Use "\."

```
prompt$ echo "ab" | grep "a\."
prompt$ █
```

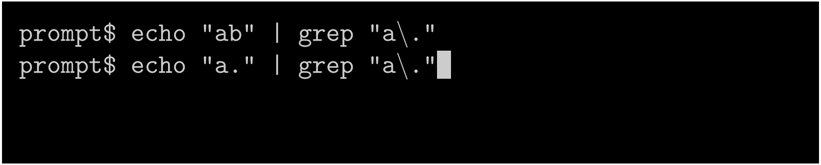
How to match an actual "." (only)?

1. Use "[.]"



```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
a.
prompt$
```

2. Use "\."



```
prompt$ echo "ab" | grep "a\."
prompt$ echo "a." | grep "a\."
```

How to match an actual "." (only)?

1. Use "[.]"

```
prompt$ echo "ab" | grep "a[.]"
prompt$ echo "a." | grep "a[.]"
a.
prompt$
```

2. Use "\."

```
prompt$ echo "ab" | grep "a\."
prompt$ echo "a." | grep "a\."
a.
prompt$ █
```

Escaping characters

The “\” character will “escape” the following character

- Converts meta characters into ordinary ones
- Be careful because this is also a special shell character

Example: match the string “[”

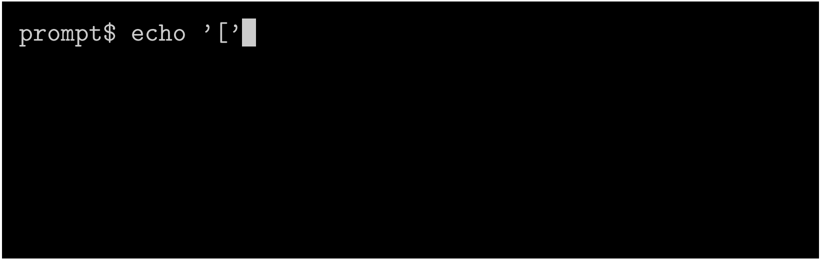
```
prompt$ █
```

Escaping characters

The “\” character will “escape” the following character

- Converts meta characters into ordinary ones
- Be careful because this is also a special shell character

Example: match the string “[”



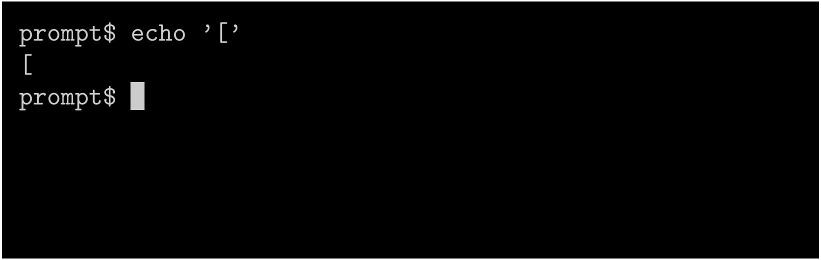
```
prompt$ echo '['
```

Escaping characters

The “\” character will “escape” the following character

- Converts meta characters into ordinary ones
- Be careful because this is also a special shell character

Example: match the string “[”



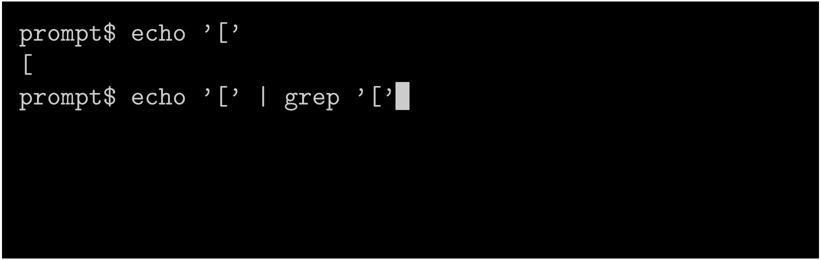
```
prompt$ echo '['
[
prompt$ █
```

Escaping characters

The “\” character will “escape” the following character

- Converts meta characters into ordinary ones
- Be careful because this is also a special shell character

Example: match the string “[”



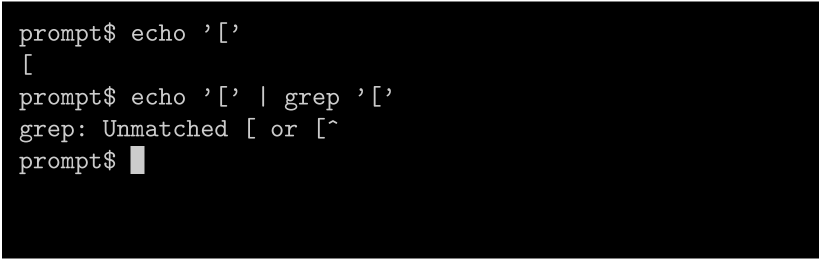
```
prompt$ echo '['
[
prompt$ echo '[' | grep '['
```

Escaping characters

The “\” character will “escape” the following character

- Converts meta characters into ordinary ones
- Be careful because this is also a special shell character

Example: match the string “[”



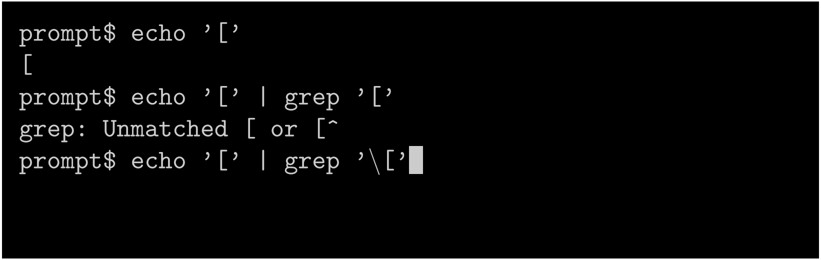
```
prompt$ echo '['
[
prompt$ echo '[' | grep '['
grep: Unmatched [ or [^
prompt$ █
```

Escaping characters

The “\” character will “escape” the following character

- Converts meta characters into ordinary ones
- Be careful because this is also a special shell character

Example: match the string “[”



```
prompt$ echo ’[’
[
prompt$ echo ’[’ | grep ’[’
grep: Unmatched [ or [^
prompt$ echo ’[’ | grep ’\[’
```

Escaping characters

The “\” character will “escape” the following character

- Converts meta characters into ordinary ones
- Be careful because this is also a special shell character

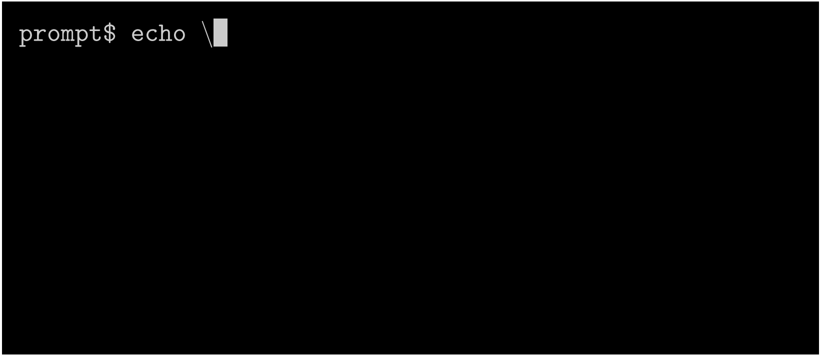
Example: match the string “[”

```
prompt$ echo '['
[
prompt$ echo '[' | grep '['
grep: Unmatched [ or [^
prompt$ echo '[' | grep '\['
[
prompt$ █
```

Example: match the string “\”

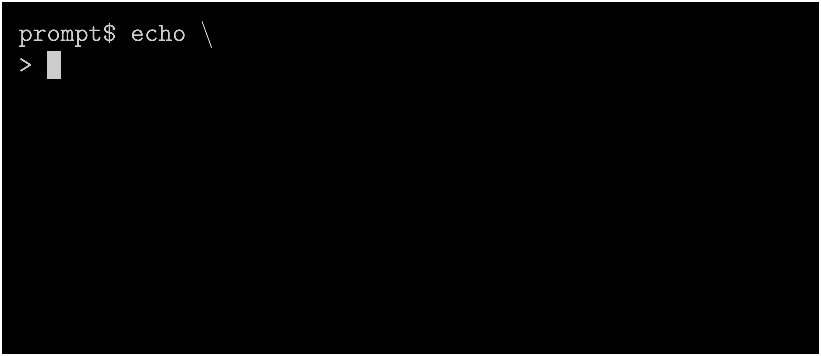
```
prompt$ 
```

Example: match the string “\”



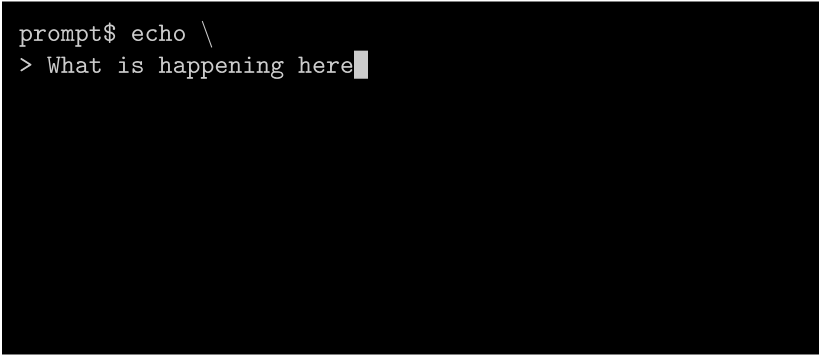
```
prompt$ echo \
```

Example: match the string “\”



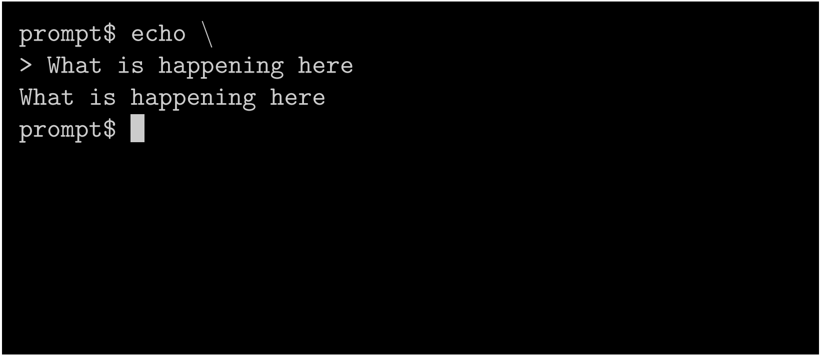
```
prompt$ echo \
> 
```

Example: match the string "\"



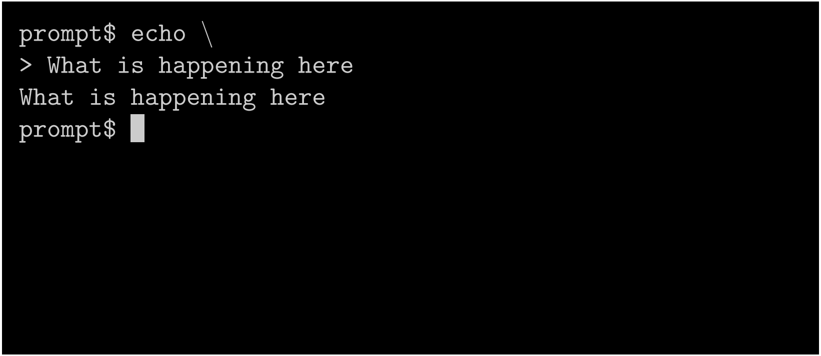
```
prompt$ echo \
> What is happening here
```

Example: match the string “\”



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ █
```

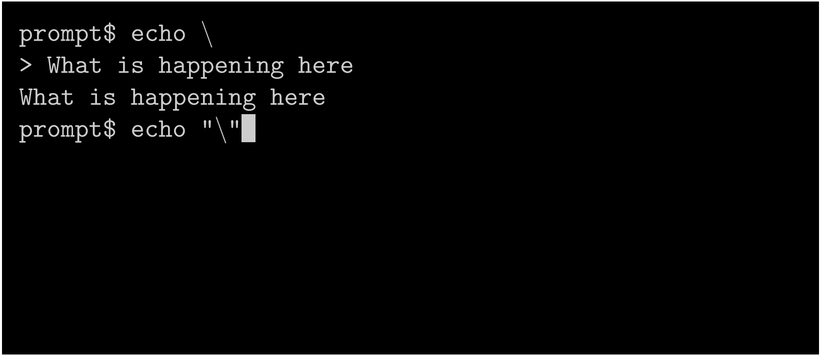
Example: match the string “\”



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ █
```

- \ at the end of a line means “let me continue on the next line”

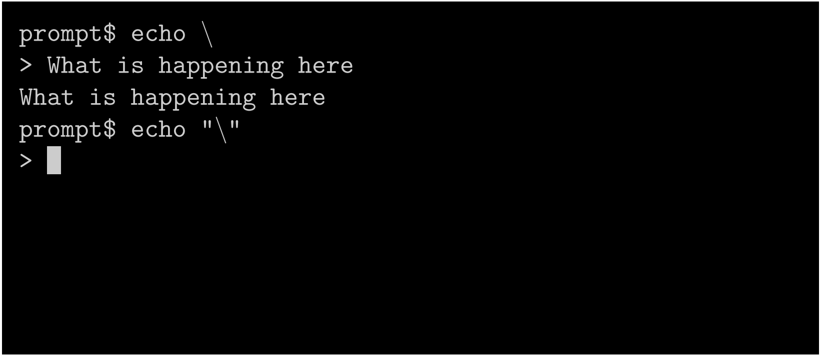
Example: match the string "\"



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
```

- \ at the end of a line means "let me continue on the next line"

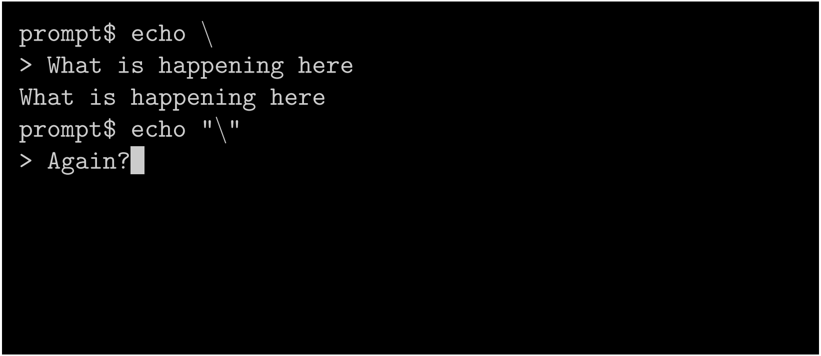
Example: match the string “\”



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
> 
```

- \ at the end of a line means “let me continue on the next line”

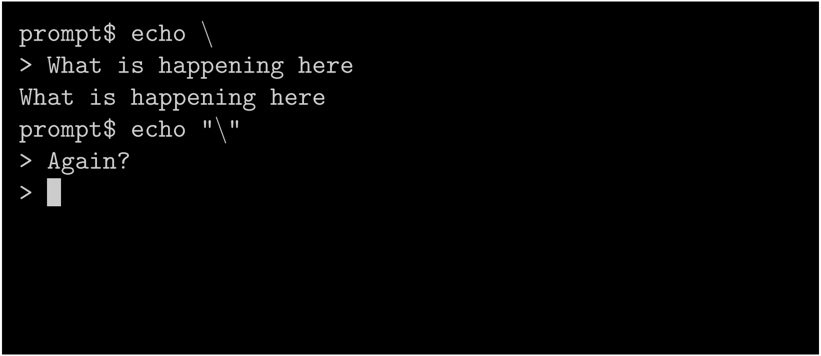
Example: match the string “\”



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
> Again?
```

- \ at the end of a line means “let me continue on the next line”

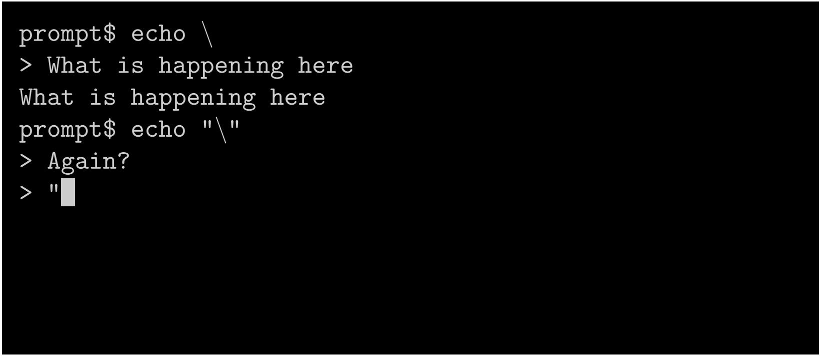
Example: match the string "\"



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
> Again?
> █
```

- \ at the end of a line means "let me continue on the next line"

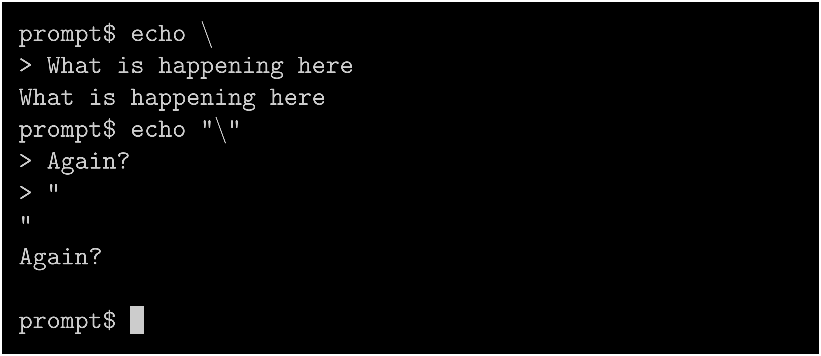
Example: match the string “\”



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
> Again?
> "
```

- \ at the end of a line means “let me continue on the next line”

Example: match the string "\"

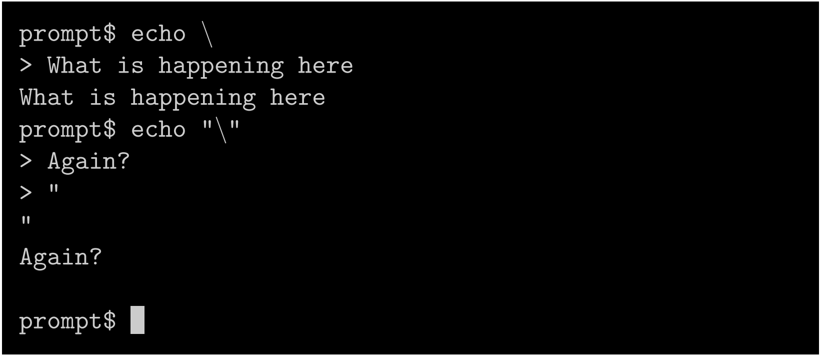


```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
> Again?
> "
"
Again?

prompt$ █
```

- \ at the end of a line means "let me continue on the next line"

Example: match the string "\"



```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
> Again?
> "
"
Again?

prompt$ █
```

- \ at the end of a line means "let me continue on the next line"
- \" converts " to a normal shell character

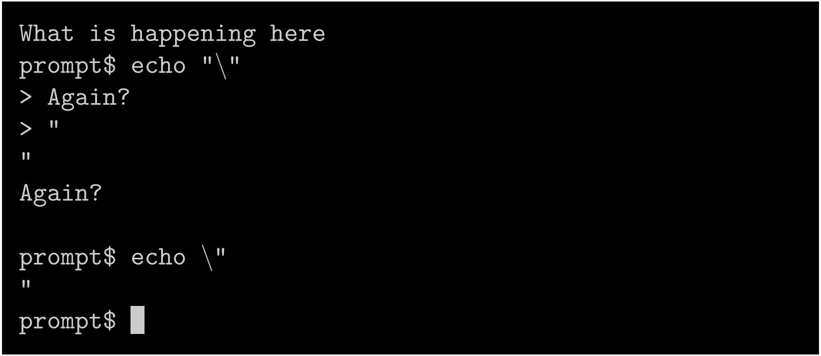
Example: match the string “\”

```
prompt$ echo \
> What is happening here
What is happening here
prompt$ echo "\"
> Again?
> "
"
Again?

prompt$ echo \"
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character

Example: match the string “\”

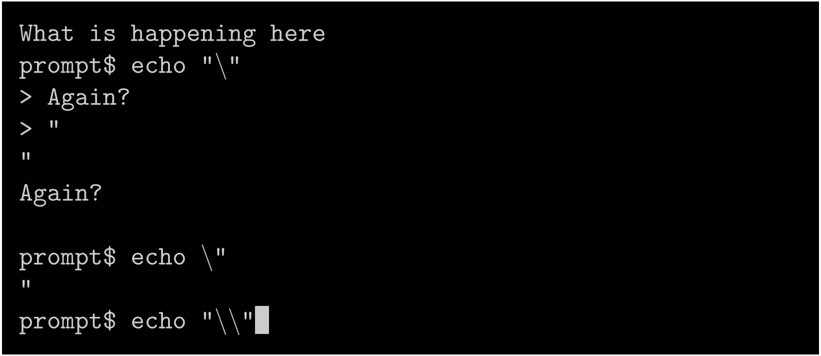


```
What is happening here
prompt$ echo "\"
> Again?
> "
"
Again?

prompt$ echo \"
"
prompt$ █
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character

Example: match the string “\”

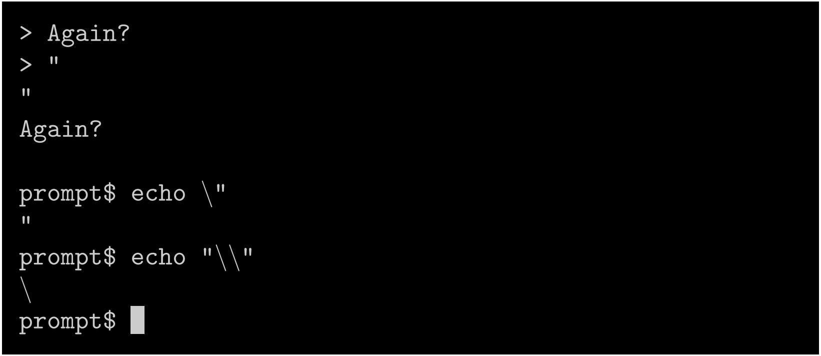


```
What is happening here
prompt$ echo "\"
> Again?
> "
"
Again?

prompt$ echo \"
"
prompt$ echo "\\"
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character

Example: match the string “\”

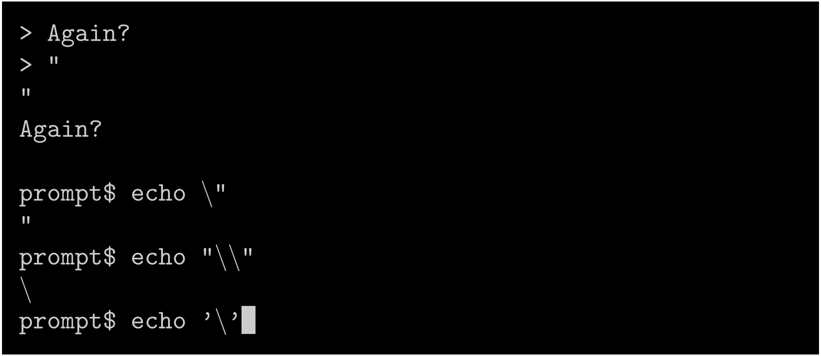


```
> Again?
> "
"
Again?

prompt$ echo \"
"
prompt$ echo "\\"
\
prompt$ █
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

Example: match the string “\”

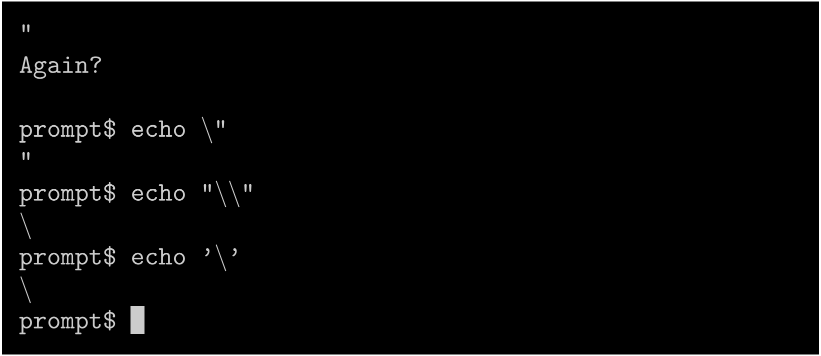


```
> Again?
> "
"
Again?

prompt$ echo \"
"
prompt$ echo "\\"
\
prompt$ echo ’\’
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

Example: match the string “\”

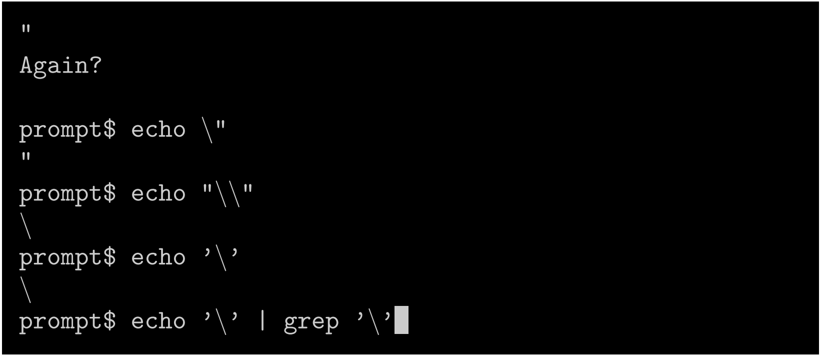


```
"
Again?

prompt$ echo \"
"
prompt$ echo "\\"
\
prompt$ echo ’\’
\
prompt$ █
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

Example: match the string “\”

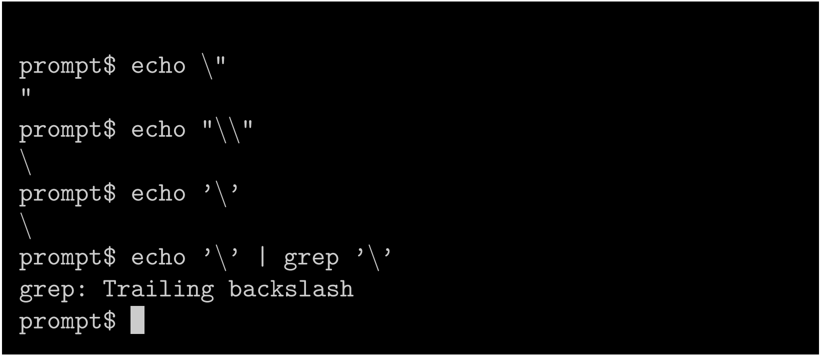


```
"
Again?

prompt$ echo \"
"
prompt$ echo "\\"
\
prompt$ echo ’\’
\
prompt$ echo ’\’ | grep ’\’
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

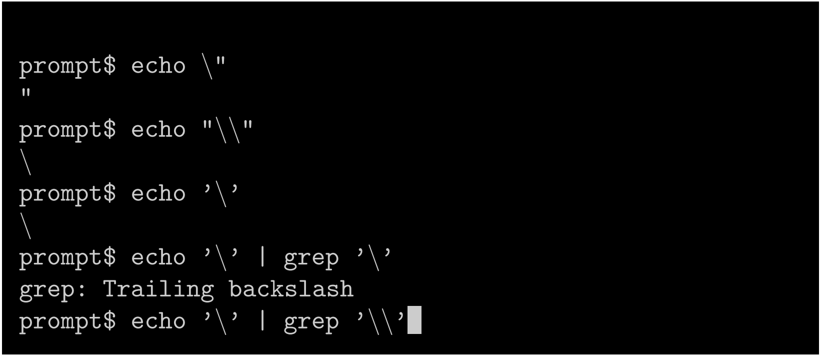
Example: match the string “\”



```
prompt$ echo \"
"
prompt$ echo "\\"
\
prompt$ echo ’\’
\
prompt$ echo ’\’ | grep ’\’
grep: Trailing backslash
prompt$ █
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

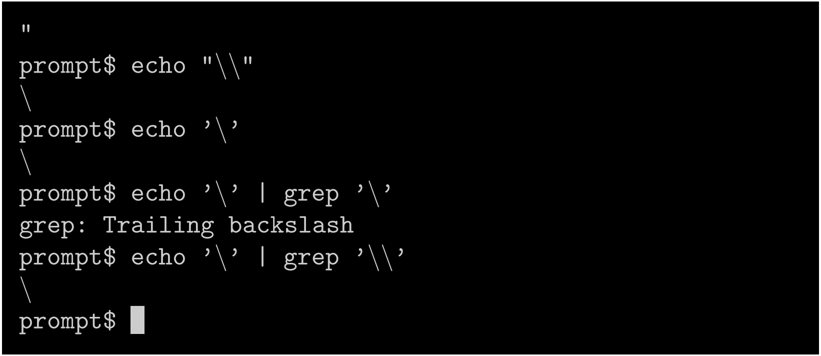
Example: match the string “\”



```
prompt$ echo \"
"
prompt$ echo "\\"
\
prompt$ echo ’\’
\
prompt$ echo ’\’ | grep ’\’
grep: Trailing backslash
prompt$ echo ’\’ | grep ’\\’
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

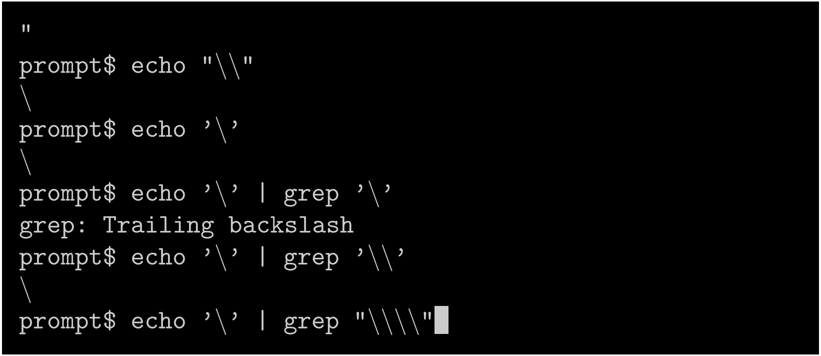
Example: match the string “\”



```
"
prompt$ echo "\\"
\
prompt$ echo ’\’
\
prompt$ echo ’\’ | grep ’\’
grep: Trailing backslash
prompt$ echo ’\’ | grep ’\\’
\
prompt$ █
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

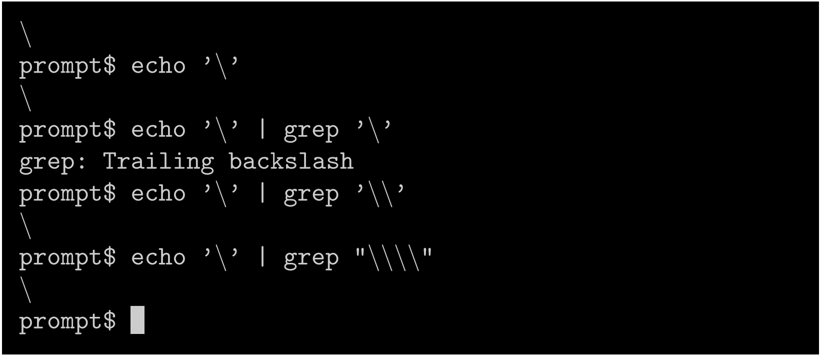
Example: match the string "\"



```
"
prompt$ echo "\\"
\
prompt$ echo '\'
\
prompt$ echo '\' | grep '\'
grep: Trailing backslash
prompt$ echo '\' | grep '\\'
\
prompt$ echo '\' | grep "\\\\"
```

- \ at the end of a line means "let me continue on the next line"
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character

Example: match the string “\”



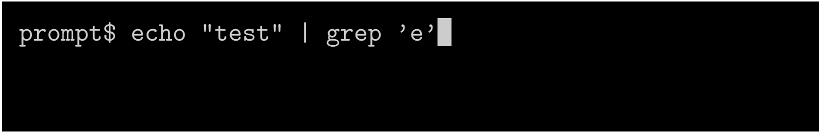
```
\
prompt$ echo ’\’
\
prompt$ echo ’\’ | grep ’\’
grep: Trailing backslash
prompt$ echo ’\’ | grep ’\\’
\
prompt$ echo ’\’ | grep "\\\\"
\
prompt$ █
```

- \ at the end of a line means “let me continue on the next line”
- \" converts " to a normal shell character
- \\ converts \ to a normal shell character
- Probably best to use single quotes for regular expressions

Motivating example

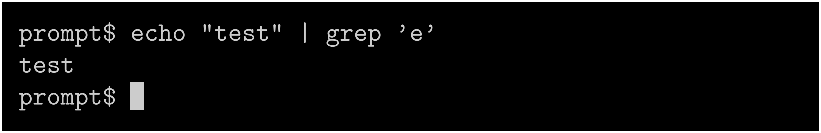
```
prompt$ █
```

Motivating example



```
prompt$ echo "test" | grep 'e'
```

Motivating example



```
prompt$ echo "test" | grep 'e'
test
prompt$
```

Motivating example

```
prompt$ echo "test" | grep 'e'
test
prompt$
```

What just happened?

Motivating example

```
prompt$ echo "test" | grep ’e’
test
prompt$
```

What just happened?

- ”e” matches regular expression ’e’

Motivating example

```
prompt$ echo "test" | grep 'e'
test
prompt$
```

What just happened?

- "e” matches regular expression ’e’
- The line “test” contains text that matches the pattern
  - So the whole line is printed
  - Remember how grep works?
  - Print lines containing text that matches a pattern

Motivating example

```
prompt$ echo "test" | grep ’e’
test
prompt$
```

What just happened?

- ”e” matches regular expression ’e’
- The line “`test`” contains text that matches the pattern
  - So the whole line is printed
  - Remember how `grep` works?
  - Print lines containing text that matches a pattern

How can I force `grep` to match only the line ’e’?

More special characters

Character “^” matches “beginning of line”

- There is no “beginning of line” character
- It is a special, “imaginary character”

Character “$” matches “end of line”

- Not the newline character
- It is a special, “imaginary character”

More special characters

Character “^” matches “beginning of line”

- There is no “beginning of line” character
- It is a special, “imaginary character”

Character “$” matches “end of line”

- Not the newline character
- It is a special, “imaginary character”

Fun fact: guess what “^” and “$” do in `vi`?

More special characters

Character “ˆ” matches “beginning of line”

- There is no “beginning of line” character
- It is a special, “imaginary character”

Character “$” matches “end of line”

- Not the newline character
- It is a special, “imaginary character”

Fun fact: guess what “ˆ” and “$” do in `vi`?

ˆ moves the cursor to the beginning of the line

$ moves the cursor to the end of the line

Examples

- To match lines that start with “a”:

Examples

- To match lines that start with “a”:

```
grep '^a'
```

Examples

- To match lines that start with “a”:

  `grep '^a'`
- To match lines that end with “a”:

Examples

- To match lines that start with “a”:

```
grep '^a'
```

- To match lines that end with “a”:

```
grep 'a$'
```

Examples

- To match lines that start with “a”:

  ```
  grep ’^a’
  ```

- To match lines that end with “a”:

  ```
  grep ’a$’
  ```

- To match the line “a”:

Examples

- To match lines that start with “a”:

  ```
  grep '^a'
  ```

- To match lines that end with “a”:

  ```
  grep 'a$'
  ```

- To match the line “a”:

  ```
  grep '^a$'
  ```

Examples

- To match lines that start with “a”:

```
grep '^a'
```

- To match lines that end with “a”:

```
grep 'a$'
```

- To match the line “a”:

```
grep '^a$'
```

- To match a line containing “^”:

Examples

- To match lines that start with “a”:

  ```
  grep '^a'
  ```
- To match lines that end with “a”:

  ```
  grep 'a$'
  ```
- To match the line “a”:

  ```
  grep '^a$'
  ```
- To match a line containing “^”:

  ```
  grep '\^'
  ```

Examples

- To match lines that start with “a”:

  `grep '^a'`

- To match lines that end with “a”:

  `grep 'a$'`

- To match the line “a”:

  `grep '^a$'`

- To match a line containing “^”:

  `grep '\^'`

- To match a line containing “$”:

Examples

- To match lines that start with “a”:

  ```
  grep ’^a’
  ```

- To match lines that end with “a”:

  ```
  grep ’a$’
  ```

- To match the line “a”:

  ```
  grep ’^a$’
  ```

- To match a line containing “^”:

  ```
  grep ’\^’
  ```

- To match a line containing “$”:

  ```
  grep ’\$’
  ```

Repeating things

This is where regular expressions get tricky to match, by hand

Repeating things

This is where regular expressions get tricky to match, by hand

"*" means, repeat the previous thing, zero or more times

- The "previous thing" is a character
    - For basic regular expressions, anyway

Simple example with “∗”

- Does string ‘bd’ match regex ‘ba*d’?

Simple example with “∗”

- Does string ‘`bd`’ match regex ‘`ba*d`’?
  - ‘`b`’ matches ‘`b`’
  - ‘’ matches ‘`a*`’
  - ‘`d`’ matches ‘`d`’

Simple example with “∗”

- Does string ‘`bd`’ match regex ‘`ba*d`’?
  - ‘`b`’ matches ‘`b`’
  - ‘’ matches ‘`a*`’
  - ‘`d`’ matches ‘`d`’

Yes.

Simple example with “∗”

- Does string ‘bd’ match regex ‘ba∗d’?
  - ‘b’ matches ‘b’
  - ‘’ matches ‘a∗’
  - ‘d’ matches ‘d’

  Yes.
- Does string ‘baaad’ match regex ‘ba∗d’?

Simple example with “∗”

- Does string ‘bd’ match regex ‘ba∗d’?
  - ‘b’ matches ‘b’
  - ‘’ matches ‘a∗’
  - ‘d’ matches ‘d’

  Yes.
- Does string ‘baaad’ match regex ‘ba∗d’?
  - ‘b’ matches ‘b’
  - ‘aaa’ matches ‘a∗’
  - ‘d’ matches ‘d’

Simple example with “∗”

- Does string ‘bd’ match regex ‘ba∗d’?
  - ‘b’ matches ‘b’
  - ‘’ matches ‘a∗’
  - ‘d’ matches ‘d’

  Yes.
- Does string ‘baaad’ match regex ‘ba∗d’?
  - ‘b’ matches ‘b’
  - ‘aaa’ matches ‘a∗’
  - ‘d’ matches ‘d’

  Yes.

Trickier example with “∗”

- Does string ‘7’ match regex ‘`[0-9]*[02468]`’?

Trickier example with “∗”

- Does string ‘7’ match regex ‘[0-9]*[02468]’?
  - ‘7’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ but ‘7’ does not match ‘[02468]’

Trickier example with “∗”

- Does string ‘7’ match regex ‘[0-9]*[02468]’?
  - ‘7’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ but ‘7’ does not match ‘[02468]’

No.

Trickier example with “*”

- Does string ‘7’ match regex ‘[0-9]*[02468]’?
  - ‘7’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ but ‘7’ does not match ‘[02468]’

  No.
- Does string ‘8’ match regex ‘[0-9]*[02468]’?

Trickier example with “∗”

- Does string ‘7’ match regex ‘[0-9]*[02468]’?
  - ‘7’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ but ‘7’ does not match ‘[02468]’

  No.
- Does string ‘8’ match regex ‘[0-9]*[02468]’?
  - ‘8’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ and ‘8’ matches ‘[02468]’

Trickier example with “∗”

- Does string ‘7’ match regex ‘`[0-9]*[02468]`’?
  - ‘7’ matches ‘`[0-9]*`’ but ‘’ does not match ‘`[02468]`’
  - ‘’ matches ‘`[0-9]*`’ but ‘7’ does not match ‘`[02468]`’

  No.
- Does string ‘8’ match regex ‘`[0-9]*[02468]`’?
  - ‘8’ matches ‘`[0-9]*`’ but ‘’ does not match ‘`[02468]`’
  - ‘’ matches ‘`[0-9]*`’ and ‘8’ matches ‘`[02468]`’

  Yes.

Trickier example with “∗”

- Does string ‘7’ match regex ‘[0-9]*[02468]’?
  - ‘7’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ but ‘7’ does not match ‘[02468]’

No.

- Does string ‘8’ match regex ‘[0-9]*[02468]’?
  - ‘8’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ and ‘8’ matches ‘[02468]’

Yes.

- Does string ‘84’ match regex ‘[0-9]*[02468]’?
  - ‘8’ matches ‘[0-9]*’ and ‘4’ matches ‘[02468]’
  - ‘’ matches ‘[0-9]*’ and ‘8’ matches ‘[02468]’
    but then we have an extra ‘4’ that matches nothing

Trickier example with “*”

- Does string ‘7’ match regex ‘[0-9]*[02468]’?
  - ‘7’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ but ‘7’ does not match ‘[02468]’

  No.
- Does string ‘8’ match regex ‘[0-9]*[02468]’?
  - ‘8’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ and ‘8’ matches ‘[02468]’

  Yes.
- Does string ‘84’ match regex ‘[0-9]*[02468]’?
  - ‘8’ matches ‘[0-9]*’ and ‘4’ matches ‘[02468]’
  - ‘’ matches ‘[0-9]*’ and ‘8’ matches ‘[02468]’ but then we have an extra ‘4’ that matches nothing

  Yes.

Trickier example with “∗”

- Does string ‘7’ match regex ‘[0-9]*[02468]’?
  - ‘7’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ but ‘7’ does not match ‘[02468]’

  No.
- Does string ‘8’ match regex ‘[0-9]*[02468]’?
  - ‘8’ matches ‘[0-9]*’ but ‘’ does not match ‘[02468]’
  - ‘’ matches ‘[0-9]*’ and ‘8’ matches ‘[02468]’

  Yes.
- Does string ‘84’ match regex ‘[0-9]*[02468]’?
  - ‘8’ matches ‘[0-9]*’ and ‘4’ matches ‘[02468]’
  - ‘’ matches ‘[0-9]*’ and ‘8’ matches ‘[02468]’
    but then we have an extra ‘4’ that matches nothing

  Yes.

  Remember: regular expressions match as much as they can

Some example regular expressions

- What does regular expression ‘`hello.*`’ mean?

Some example regular expressions

- What does regular expression ‘`hello.*`’ mean?
  - ‘`hello`’ followed by any other characters

Some example regular expressions

- What does regular expression ‘`hello.*`’ mean?
    - ‘`hello`’ followed by any other characters
- Legal variable names in `bash` (and C, and Java. . .):

Some example regular expressions

- What does regular expression ‘`hello.*`’ mean?
  - ‘`hello`’ followed by any other characters
- Legal variable names in `bash` (and C, and Java. . .):

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Some example regular expressions

- What does regular expression '`hello.*`' mean?
  - '`hello`' followed by any other characters
- Legal variable names in `bash` (and C, and Java...):

```
[a-zA-Z_][a-zA-Z_0-9]*
```

- A string surrounded by double quotes
  (without allowing '`\"`' inside):

Some example regular expressions

- What does regular expression ‘`hello.*`’ mean?
  - ‘`hello`’ followed by any other characters
- Legal variable names in `bash` (and C, and Java. . .):

```
[a-zA-Z_][a-zA-Z_0-9]*
```

- A string surrounded by double quotes (without allowing ‘`\"`’ inside):

```
"[^"]*"
```

Some example regular expressions

- What does regular expression ‘`hello.*`’ mean?
  - ‘`hello`’ followed by any other characters
- Legal variable names in `bash` (and C, and Java. . .):

```
[a-zA-Z_][a-zA-Z_0-9]*
```

- A string surrounded by double quotes (without allowing ‘`\"`’ inside):

```
"[^"]*"
```

- Why not `".*"` for strings?

Some example regular expressions

- What does regular expression ‘`hello.*`’ mean?
  - ‘`hello`’ followed by any other characters
- Legal variable names in `bash` (and C, and Java...):

```
[a-zA-Z_][a-zA-Z_0-9]*
```

- A string surrounded by double quotes (without allowing ‘`\"`’ inside):

```
"[^"]*"
```

- Why not `".*"` for strings?
  - Because `"foo"bar"` matches `".*"`

Specified number of repeats

“$\backslash\{\ldots\backslash\}$”: repeat the previous thing, some number of times

$\backslash\{n\backslash\}$: repeat exactly *n* times

$\backslash\{n,m\backslash\}$: repeat at least *n* and no more than *m* times

$\backslash\{n,\backslash\}$: repeat at least *n* times

Not supported everywhere

Note: “$*$” is the same as “$\backslash\{0,\backslash\}$”

Some examples

- Date strings (`mm/dd/yyyy` or `mm-dd-yyyy`)

Some examples

- Date strings (`mm/dd/yyyy` or `mm-dd-yyyy`)

```
[0-9]\{2\}[/-][0-9]\{2\}[/-][0-9]\{4\}
```

Some examples

- Date strings (`mm/dd/yyyy` or `mm-dd-yyyy`)

```
[0-9]\{2\}[/-][0-9]\{2\}[/-][0-9]\{4\}
```

Note that `mm/dd-yyyy` and `mm-dd/yyyy` will match also

Some examples

- Date strings (`mm/dd/yyyy` or `mm-dd-yyyy`)

  ```
  [0-9]\{2\}[/-][0-9]\{2\}[/-][0-9]\{4\}
  ```

  Note that `mm/dd-yyyy` and `mm-dd/yyyy` will match also

- City, state, zip code

Some examples

- Date strings (mm/dd/yyyy or mm-dd-yyyy)

  ```
  [0-9]\{2\}[/-][0-9]\{2\}[/-][0-9]\{4\}
  ```

  Note that mm/dd-yyyy and mm-dd/yyyy will match also

- City, state, zip code

  ```
  [- A-Za-z]*, [A-Z]\{2\}, [0-9]\{5\}
  ```

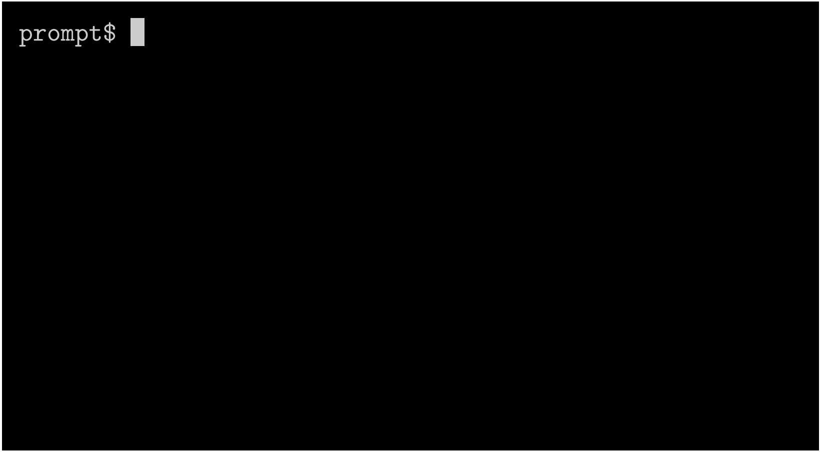
Extended regular expressions

- Have additional features and a few small differences
- Used by `egrep` or `grep -e`
- Used by `sed` and `awk`

Simple changes in extended regular expressions

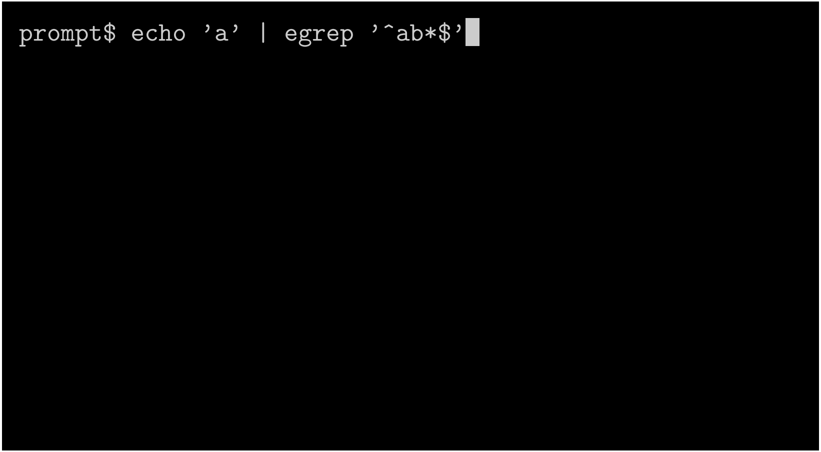
- It is not necessary to escape the braces. E.g.:
  - Use ‘{*n*}’ instead of ‘\{*n*\}’
- ‘+’ is shorthand for ‘{1,}’ (repeat one or more times)
- ‘?’ is shorthand for ‘{0,1}’ (previous thing is optional)

Examples using egrep



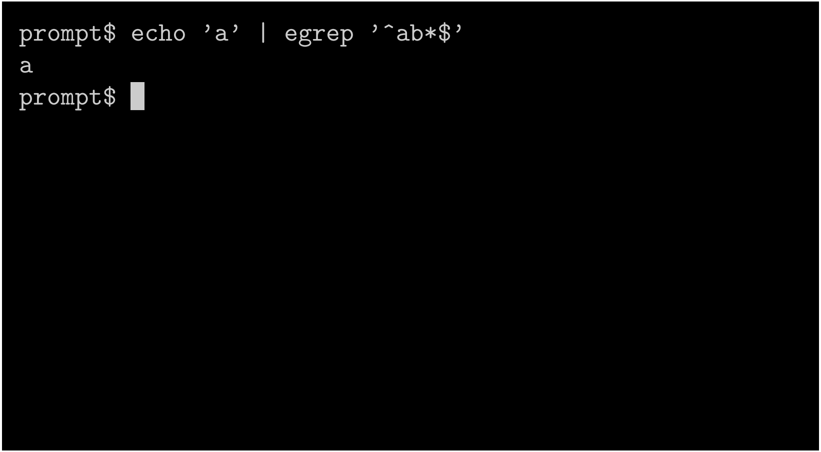
```
prompt$ 
```

Examples using egrep



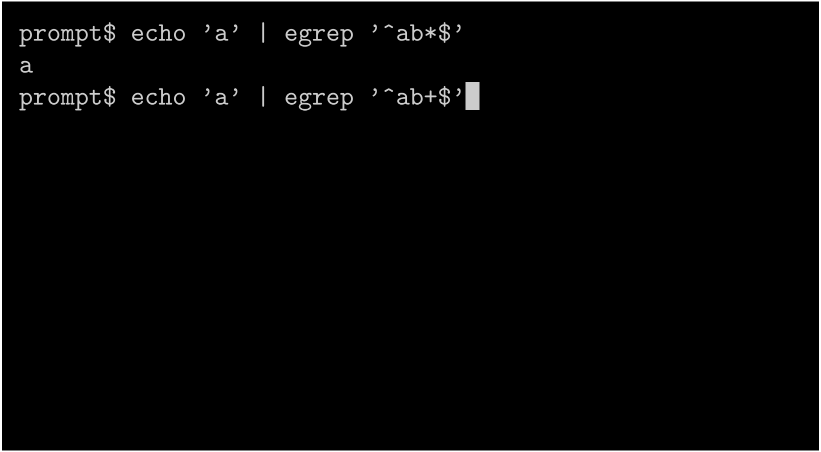
```
prompt$ echo ’a’ | egrep ’^ab*$’
```

Examples using egrep



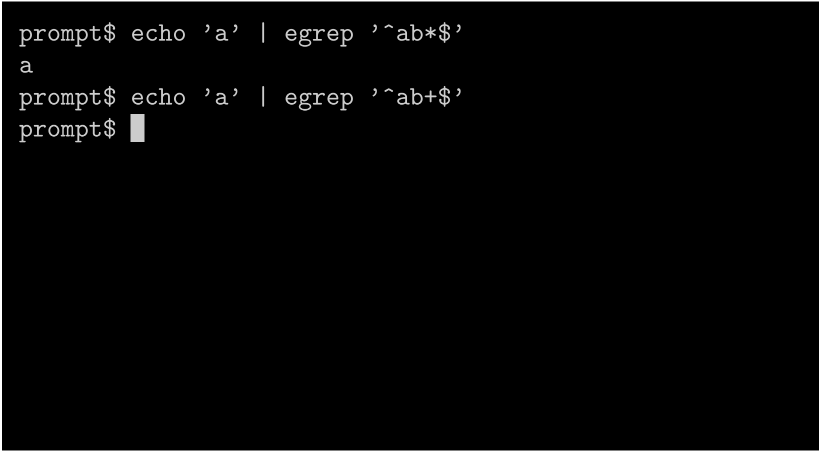
```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ █
```

Examples using egrep



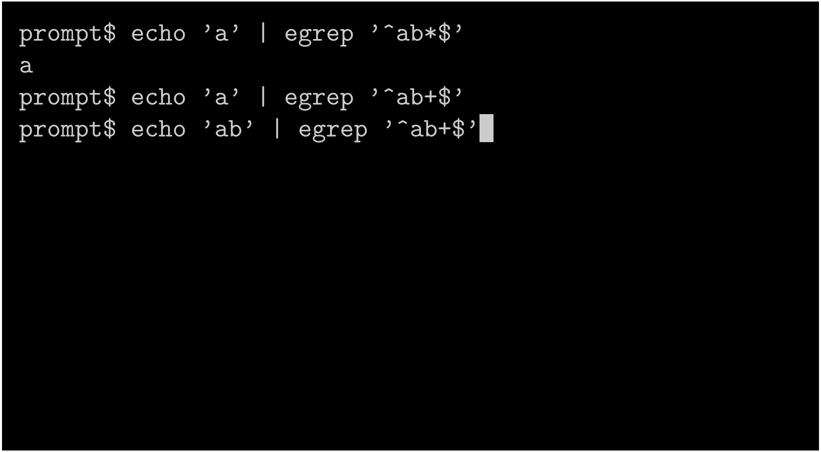
```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
```

Examples using egrep



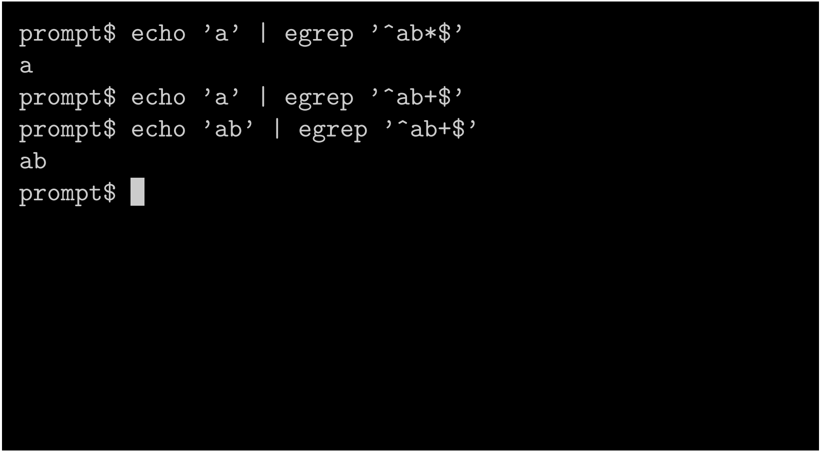
```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ 
```

Examples using egrep



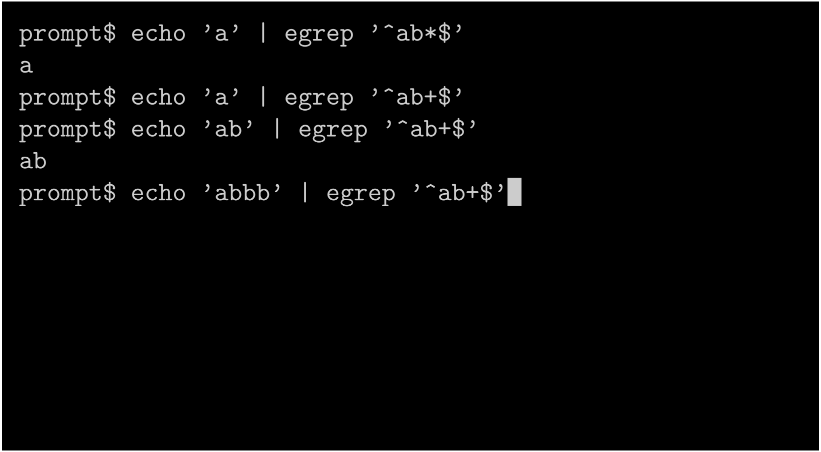
```
prompt$ echo ’a’ | egrep ’ˆab*$’
a
prompt$ echo ’a’ | egrep ’ˆab+$’
prompt$ echo ’ab’ | egrep ’ˆab+$’
```

Examples using egrep



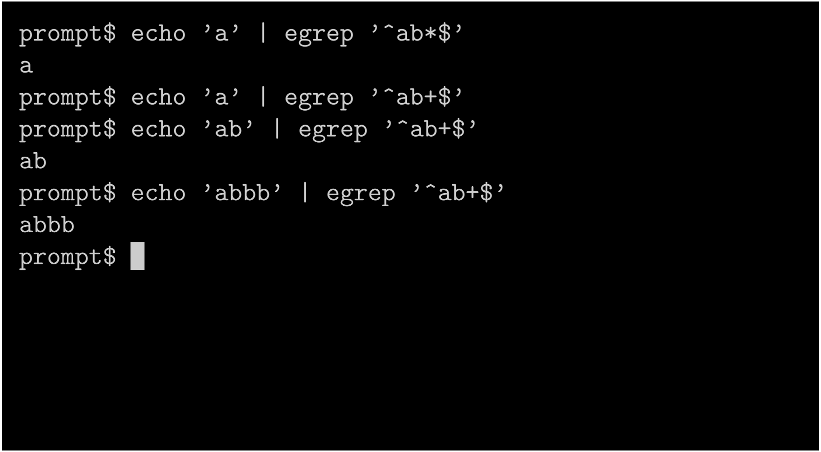
```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ █
```

Examples using egrep



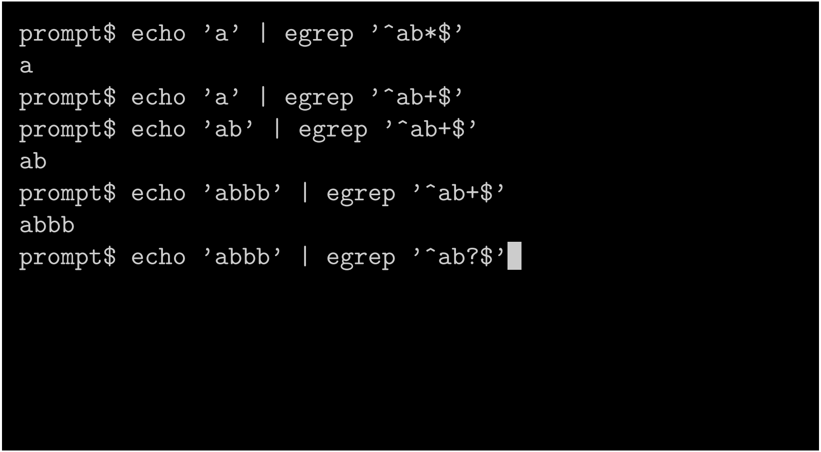
```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ echo ’abbb’ | egrep ’^ab+$’
```

Examples using egrep



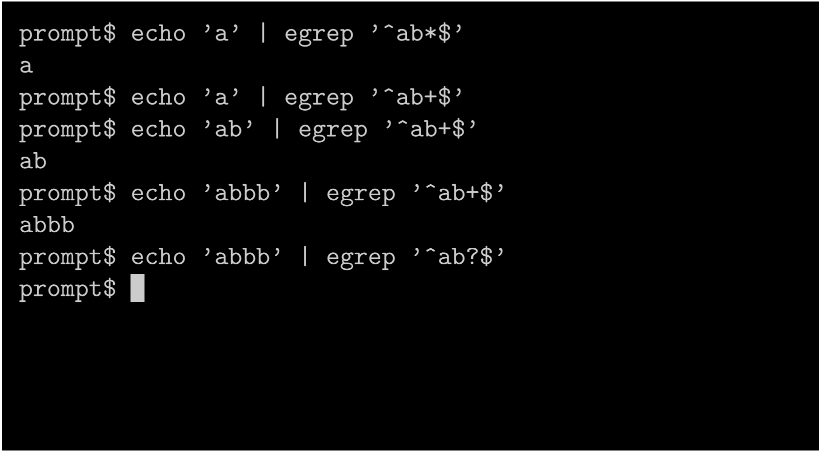
```
prompt$ echo ’a’ | egrep ’ˆab*$’
a
prompt$ echo ’a’ | egrep ’ˆab+$’
prompt$ echo ’ab’ | egrep ’ˆab+$’
ab
prompt$ echo ’abbb’ | egrep ’ˆab+$’
abbb
prompt$ █
```

Examples using egrep



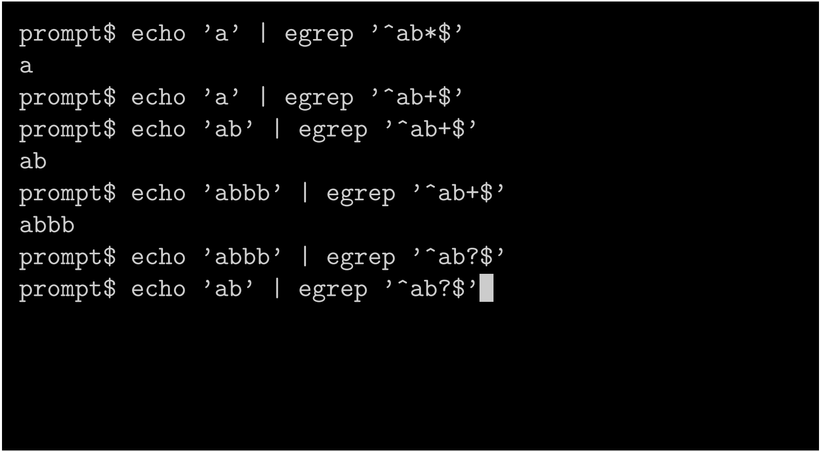
```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ echo ’abbb’ | egrep ’^ab+$’
abbb
prompt$ echo ’abbb’ | egrep ’^ab?$’
```

Examples using egrep



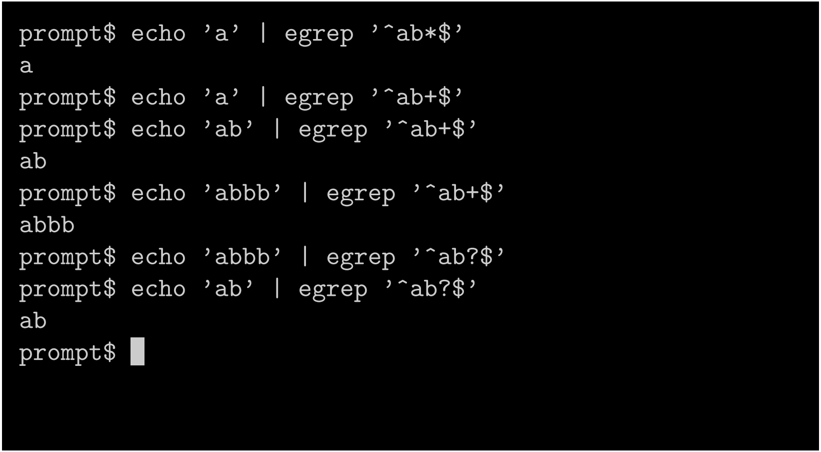
```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ echo ’abbb’ | egrep ’^ab+$’
abbb
prompt$ echo ’abbb’ | egrep ’^ab?$’
prompt$ █
```

Examples using egrep



```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ echo ’abbb’ | egrep ’^ab+$’
abbb
prompt$ echo ’abbb’ | egrep ’^ab?$’
prompt$ echo ’ab’ | egrep ’^ab?$’
```

Examples using egrep

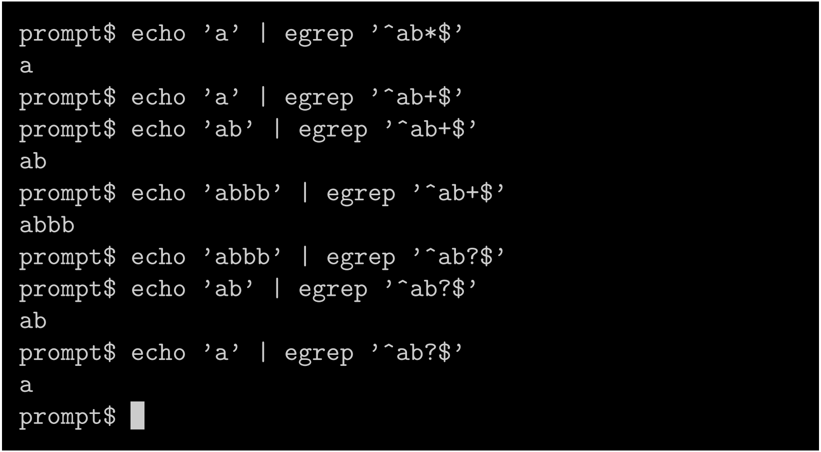


```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ echo ’abbb’ | egrep ’^ab+$’
abbb
prompt$ echo ’abbb’ | egrep ’^ab?$’
prompt$ echo ’ab’ | egrep ’^ab?$’
ab
prompt$ █
```

Examples using egrep

```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ echo ’abbb’ | egrep ’^ab+$’
abbb
prompt$ echo ’abbb’ | egrep ’^ab?$’
prompt$ echo ’ab’ | egrep ’^ab?$’
ab
prompt$ echo ’a’ | egrep ’^ab?$’
```

Examples using egrep



```
prompt$ echo ’a’ | egrep ’^ab*$’
a
prompt$ echo ’a’ | egrep ’^ab+$’
prompt$ echo ’ab’ | egrep ’^ab+$’
ab
prompt$ echo ’abbb’ | egrep ’^ab+$’
abbb
prompt$ echo ’abbb’ | egrep ’^ab?$’
prompt$ echo ’ab’ | egrep ’^ab?$’
ab
prompt$ echo ’a’ | egrep ’^ab?$’
a
prompt$ █
```

Significant extensions

- There are 2 more things to discuss
- These give regular expressions much more power
- They also make things much more complicated

Significant extensions

- There are 2 more things to discuss
- These give regular expressions much more power
- They also make things much more complicated

- Now is a good time to ask questions before things get crazy (or review if you are reading this at home)

Grouping

Regular expressions may be grouped using ()

Grouping

Regular expressions may be grouped using ()

- The “previous thing” for a repeat operator may be a regex

Grouping

Regular expressions may be grouped using ()

- The “previous thing” for a repeat operator may be a regex
- We can repeat entire subexpressions
- We can make entire subexpressions optional

Simple example:

`ab+` means:

Grouping

Regular expressions may be grouped using ()

- The "previous thing" for a repeat operator may be a regex
- We can repeat entire subexpressions
- We can make entire subexpressions optional

Simple example:

`ab+` means: 'a' followed by one or more 'b's

- Strings 'ab', 'abb', 'abbb', . . . , match this

`(ab)+` means:

Grouping

Regular expressions may be grouped using ()

- The "previous thing" for a repeat operator may be a regex
- We can repeat entire subexpressions
- We can make entire subexpressions optional

Simple example:

`ab+` means: 'a' followed by one or more 'b's

- Strings 'ab', 'abb', 'abbb', . . . , match this

`(ab)+` means: a sequence of one or more 'ab's

- Strings 'ab', 'abab', 'ababab', . . . , match this

`(ab+)+` means:

Grouping

Regular expressions may be grouped using ()

- The "previous thing" for a repeat operator may be a regex
- We can repeat entire subexpressions
- We can make entire subexpressions optional

Simple example:

`ab+` means: 'a' followed by one or more 'b's

- Strings 'ab', 'abb', 'abbb', . . . , match this

`(ab)+` means: a sequence of one or more 'ab's

- Strings 'ab', 'abab', 'ababab', . . . , match this

`(ab+)+` means:

- One or more strings matching ab+, concatenated

Grouping

Regular expressions may be grouped using ()

- The "previous thing" for a repeat operator may be a regex
- We can repeat entire subexpressions
- We can make entire subexpressions optional

Simple example:

`ab+` means: 'a' followed by one or more 'b's

- Strings 'ab', 'abb', 'abbb', . . . , match this

`(ab)+` means: a sequence of one or more 'ab's

- Strings 'ab', 'abab', 'ababab', . . . , match this

`(ab+)+` means:

- One or more strings matching ab+, concatenated
- Start with 'a', end with 'b', never 2 'a's together

Serious grouping examples

- Zip code with optional “extra 4”:

Serious grouping examples

- Zip code with optional “extra 4”:

```
[0-9]{5}(-[0-9]{4})?
```

Serious grouping examples

- Zip code with optional "extra 4":

```
[0-9]{5}(-[0-9]{4})?
```

- Well–formed strings, where '\"' is allowed inside

Serious grouping examples

- Zip code with optional “extra 4”:

```
[0-9]{5}(-[0-9]{4})?
```

- Well–formed strings, where ‘\"’ is allowed inside

```
"([^"]?(\\")?)*"
```

Serious grouping examples

- Zip code with optional “extra 4”:

```
[0-9]{5}(-[0-9]{4})?
```

- Well–formed strings, where ‘\"’ is allowed inside

```
"([^"]?(\\")?)*"
```

- A “real value” constant as allowed in C or Java

Serious grouping examples

- Zip code with optional “extra 4”:

```
[0-9]{5}(-[0-9]{4})?
```

- Well–formed strings, where ‘\"’ is allowed inside

```
"([^"]?(\\")?)*"
```

- A “real value” constant as allowed in C or Java

```
-?[0-9]+(\.[0-9]+)?([eE]-?[0-9]+)?
```

Choosing between expressions

‘|’ means “or”

- Not needed for single characters — use `[]` instead
- Makes sense when we group things

Simple example:

`(ab)|(cd)|(ef)` means: ‘ab’ or ‘cd’ or ‘ef’

```
prompt$ █
```

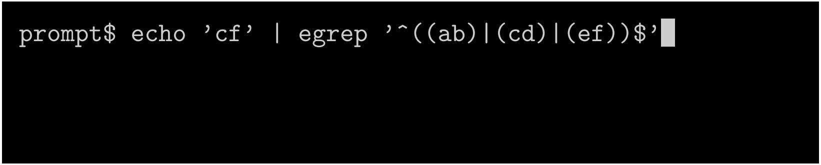
Choosing between expressions

‘|’ means “or”

- Not needed for single characters — use [] instead
- Makes sense when we group things

Simple example:

`(ab)|(cd)|(ef)` means: ‘ab’ or ‘cd’ or ‘ef’



```
prompt$ echo 'cf' | egrep '^((ab)|(cd)|(ef))$'
```

Choosing between expressions

'|' means "or"

- Not needed for single characters — use [] instead
- Makes sense when we group things

Simple example:

`(ab)|(cd)|(ef)` means: 'ab' or 'cd' or 'ef'

```
prompt$ echo 'cf' | egrep '^((ab)|(cd)|(ef))$'
prompt$ █
```

Choosing between expressions

‘|’ means “or”

- Not needed for single characters — use [] instead
- Makes sense when we group things

Simple example:

(ab)|(cd)|(ef) means: ‘ab’ or ‘cd’ or ‘ef’

```
prompt$ echo 'cf' | egrep '^((ab)|(cd)|(ef))$'
prompt$ echo 'ab' | egrep '^((ab)|(cd)|(ef))$'
```

Choosing between expressions

'|' means "or"

- Not needed for single characters — use [] instead
- Makes sense when we group things

Simple example:

`(ab)|(cd)|(ef)` means: 'ab' or 'cd' or 'ef'

```
prompt$ echo 'cf' | egrep '^((ab)|(cd)|(ef))$'
prompt$ echo 'ab' | egrep '^((ab)|(cd)|(ef))$'
ab
prompt$ █
```

Final examples

- Date strings (`mm/dd/yyyy` or `mm-dd-yyyy`)

Final examples

- Date strings (`mm/dd/yyyy` or `mm-dd-yyyy`)

```
[0-9]{2}(-[0-9]{2}-)|(/[0-9]{2}/)[0-9]{4}
```

- Date strings where:
  - The month is one or two digits, and is a legal month
  - The day is one or two digits
  - The year is two or four digits
  - ‘/’ separators only (let’s make it easy)

Final examples

- Date strings (`mm/dd/yyyy` or `mm-dd-yyyy`)

```
[0-9]{2}(-[0-9]{2}-)|(/[0-9]{2}/)[0-9]{4}
```

- Date strings where:
  - The month is one or two digits, and is a legal month
  - The day is one or two digits
  - The year is two or four digits
  - '/' separators only (let's make it easy)

```
((1[0-2])|[1-9])/[0-9]{1,2}/([0-9]{2}){1,2}
```

Basic regular expressions

ordinary character	match itself
[]	match one character from the list
-	set a range (unless it is first or last)
^	invert the list (if it appears first)
.	match any character
\	convert meta to ordinary character
^	imaginary beginning of line character
$	imaginary end of line character
*	match previous character zero or more times
$\backslash\{n,m\backslash\}$	match previous character between n and m times

Extended regular expressions

Same as “basic” except change or add:

( )	group patterns
$*$	match previous pattern zero or more times
+	match previous pattern one or more times
?	match previous pattern zero or one times
$\{n, m\}$	match previous pattern between n and m times
|	choose between patterns

An appropriate xkcd comic: `http://www.xkcd.com/208`

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

End of lecture