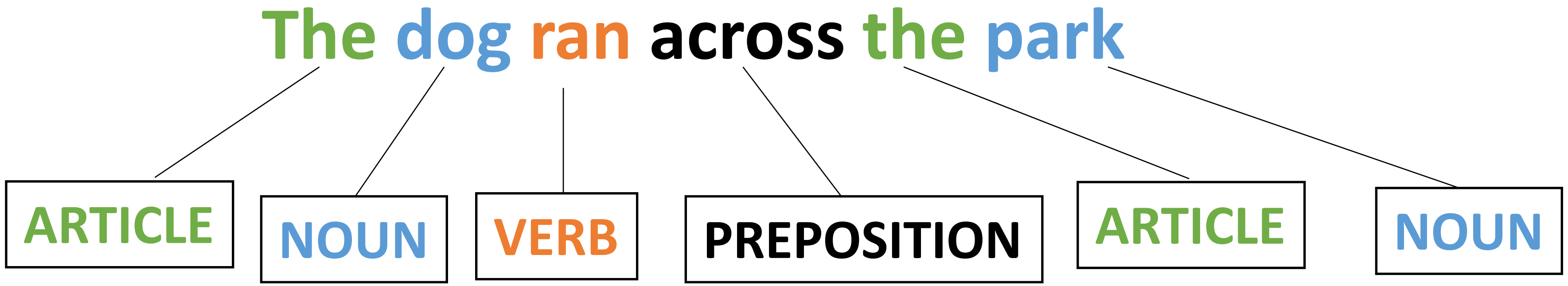
CSE110A: Compilers



- **Topics**:
  - ***Using regular expression's matchers to build scanners**:*
    - Exact match scanner (EM)
    - Start-of-string Scanner (SOS)
    - Named group matcher (NG)

The problem

- How do we move from an RE match to performing lexical analysis on a string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Do these match?

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

Do any of the tokens match?

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

What if we start “peeking” characters

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

Match!

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

Match! (ID, “v”)

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

Match! (ID, “v”) *but what is the issue?*

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Match! (ID, “v”)

but what is the issue? Not the longest match.
So, peeking a character and RE’s does not help.

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

The problem

- How do we move from an RE match to performing lexical analysis on a string

So what’s our strategy?

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

```
[(ID, “variable”), (ASSIGN, “=“),
 (NUM, “50”), (PLUS, “+”), (NUM, “30”),
 (MULT, “*”), (NUM, ”20”), (SEMI, “;”)]
```

Let’s consider 3 approaches:

- *Using RE matchers to build scanners*
  - **Exact Match (EM) scanners**
  - Start-of-string (SOS) scanners
  - Named group (NG) scanners

EM (Exact Match) Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

start with the whole string

“`variable = 50 + 30 * 20;`”

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

Try to match with all the tokens

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

start with the whole string

```
“variable = 50 + 30 * 20;”
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

Try to match with all the tokens. No match.

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

start with the whole string

```
“variable = 50 + 30 * 20;”
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

Try to match with all the tokens. No match.

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Try with one character chopped from back

```
“variable = 50 + 30 * 20;”
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

Try to match with all the tokens. No match.

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

So on

`“variable = 50 + 30 * 20;”`

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

Try to match with all the tokens. No match.

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

So on

```
“variable = 50 + 30 * 20;”
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

Try to match with all the tokens. No match.

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Where do find a match?

```
“variable = 50 + 30 * 20;”
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

we can match id

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

at this point

```
“variable = 50 + 30 * 20;”
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

we can match id

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

at this point

```
“variable = 50 + 30 * 20;”
```

Return the lexeme

```
(ID, “variable”)
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Chop the string

```
“ = 50 + 30 * 20;”
```

```
(ID, “variable”)
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Start the process over

```
“ = 50 + 30 * 20;”
```

```
(ID, “variable”)
```

EM Scanner

- Start with the whole string, remove one character at the end until a match is found. Then return the lexeme

Start the process over Where is our next match?

```
ID     = "[a-z]+"
NUM    = "[0-9]+"
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = "  |\n"
SEMI   = ";"
```

```
" = 50 + 30 * 20;"
```

```
(ID, "variable")
```

code for exact match scanner

- Provided in your homework

EM Scanner

- Pros
- Cons

EM Scanner

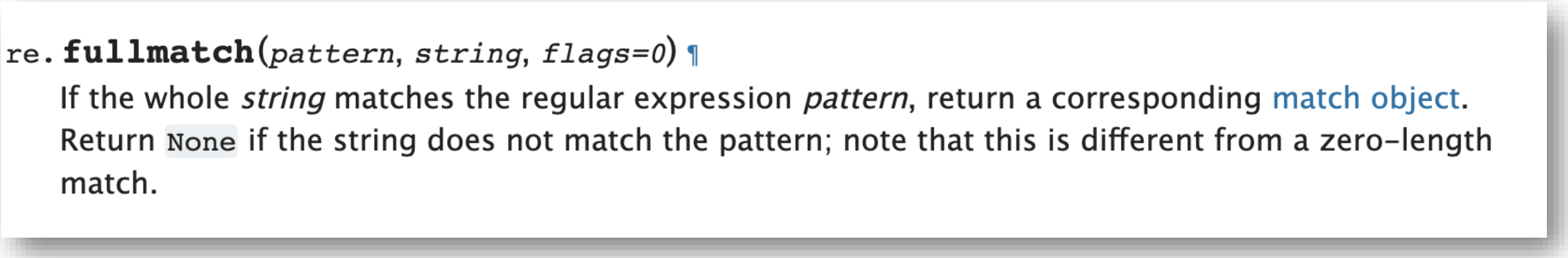
- Pros
  - Uses an exact RE matcher. Many RE match algorithms use exact matches!
- Cons
  - SLOW! Each lexeme requires many many many calls to each RE match!

Another approach

- *Using RE matchers to build scanners*
  - Exact match (EM) scanners
  - **Start-of-string (SOS) scanners**
  - named group (NG) scanners

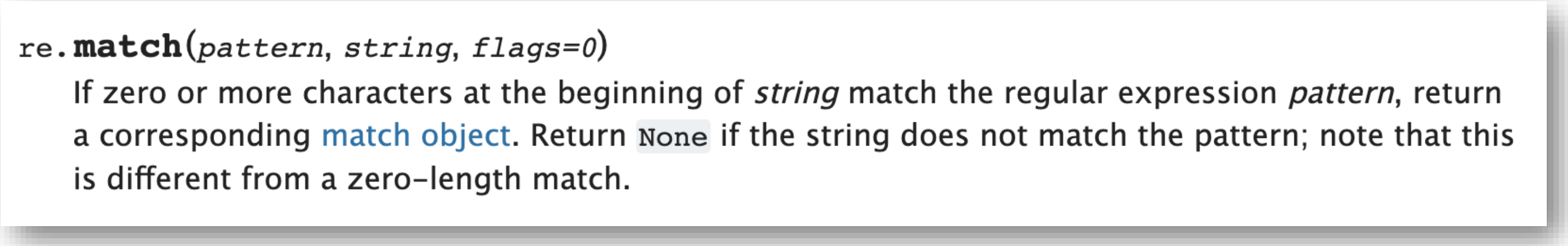
SOS (Start Of String) Scanner

- We will use a new RE match function



`re.`**`fullmatch`**(*pattern*, *string*, *flags=0*) ¶

If the whole *string* matches the regular expression *pattern*, return a corresponding match object. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.



`re.`**`match`**(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding match object. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

SOS Scanner

- The python match API gives us a match starting at the beginning of the string:

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

SOS Scanner

- The match API gives us a match starting at the beginning of the string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Feed full string into each token definition

```
“variable = 50 + 30 * 20;”
```

SOS Scanner

- The match API gives us a “greedy” match starting at the beginning of the string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Feed full string into each token definition

```
“variable = 50 + 30 * 20;”
```

We get 1 match on ID. So, we can return the lexeme

```
(ID, “variable”)
```

SOS Scanner

- The match API gives us a match starting at the beginning of the string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Chop the string

```
“variable = 50 + 30 * 20;”
```

We get 1 match. We can return the lexeme

```
(ID, “variable”)
```

SOS Scanner

- The match API gives us a match starting at the beginning of the string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Chop the string

```
“ = 50 + 30 * 20;”
```

We get 1 match. We can return the lexeme

```
(ID, “variable”)
```

SOS Scanner

- The match API gives us a match starting at the beginning of the string

What about the next one

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“ = 50 + 30 * 20;”
```

```
(ID, “variable”)
```

SOS Scanner

- The match API gives us a match starting at the beginning of the string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

What about the next one

```
“ = 50 + 30 * 20;”
```

1 match: IGNORE

```
(ID, “variable”)
```

SOS Scanner

- The match API gives us a match starting at the beginning of the string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Chop the string

```
“ = 50 + 30 * 20;”
```

1 match: IGNORE

```
(ID, “variable”)
```

SOS Scanner

- The match API gives us a match starting at the beginning of the string

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

Chop the string

```
“= 50 + 30 * 20;”
```

1 match: IGNORE

```
(ID, “variable”)
```

SOS Scanner

- Consideration

```
LETTERS = “[A-Z]+”
NUM     = “[0-9]+”
CLASS   = ”CSE110A“
```

How to scan this string?

```
“CSE110A”
```

SOS Scanner

- Consideration

Try to match on each token

```
LETTERS = “[A-Z]+”
NUM     = “[0-9]+”
CLASS   = ”CSE110A“
```

How to scan this string?

`“CSE110A”`

SOS Scanner

- Consideration

How to scan this string?

Try to match on each token

```
“CSE110A”
```

```
LETTERS  = “[A-Z]+”
NUM      = “[0-9]+”
CLASS    = ”CSE110A“
```

Two matches:

```
LETTERS: “CSE”
CLASS: ”CSE110A”
```

Which one do we choose?

SOS Scanner

- Consideration

How to scan this string?

```
“CSE110A”
```

Try to match on each token

```
LETTERS = “[A-Z]+”
NUM     = “[0-9]+”
CLASS   = ”CSE110A“
```

Two matches:

```
LETTERS: “CSE”
CLASS: ”CSE110A”
```

Which one do we choose?
The longest one!

After each pass through token REs
we have to measure match length

SOS Scanner

- Consideration

How to scan this string?

Try to match on each token

```
“CSE110A”
```

```
LETTERS = “[A-Z]+”
NUM     = “[0-9]+”
CLASS   = ”CSE110A“
```

Two matches:

```
LETTERS: “CSE”
CLASS: ”CSE110A”
```

Which one do we choose?
The longest one!

After each pass through token REs
we have to measure match length

Why didn’t we have to do this for the exact match Scanner?

SOS Scanner

- One more consideration

Within 1 RE, how does this match?

```
“CSE110A”
```

```
CLASS = “CSE|110A|CSE110A”
```

SOS Scanner

- One more consideration

Within 1 RE, how does this match?

```
"CSE110A"
```

```
CLASS = "CSE|110A|CSE110A"
```

Returns "CSE", but this isn't what we want!!!

SOS Scanner

- One more consideration

Within 1 RE, how does this match?

```
"CSE110A"
```

```
CLASS = "CSE|110A|CSE110A"
```

Returns "CSE", but this isn't what we want!!!

When using the SOS Scanner: A token definition either should not:

- *contain choices where one choice is a prefix of a subsequent choice*
- *order choices such that the longest choice is the first one*

SOS Scanner

- One more consideration

Within 1 RE, how does this match?

“CSE110A”

```
~~CLASS = “CSE|110A|CSE110A”~~
```

Returns “CSE”, but this isn’t what we want!!!

When using the SOS Scanner: A token definition either should not:

- *contain choices where one choice is a prefix of another*
- *order choices such that the longest choice is the first one*

```
CLASS = “CSE110A|110A|CSE”
```

SOS Scanner

- Pros
- Cons

SOS Scanner

- Pros
  - Much faster than EM scanner. Only 1 call to each RE per `token()` call

- Cons
  - Depends on an efficient implementation of `match()`
    - Typically provided in most RE libraries (for this exact reason)

  - Requires some care in token definitions and prefixes

SOS Scanner

We're going to optimize this to 1 RE call!
It can really help if you have many tokens

- Pros
  - Much faster than EM scanner. Only 1 call to each RE per `token()` call

- Cons
  - Depends on an efficient implementation of `match()`
    - Typically provided in most RE libraries (for this exact reason)

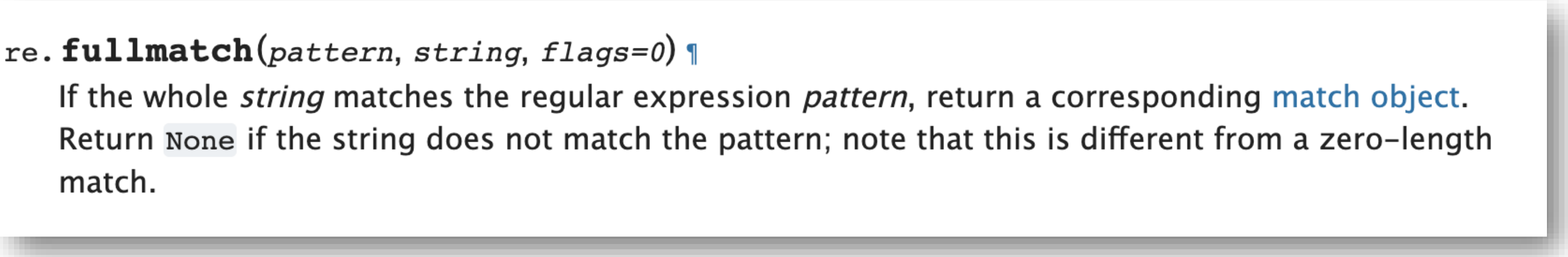
  - Requires some care in token definitions and prefixes

A third alternative:

- *Using RE matchers to build scanners*
    - Exact match (EM) scanners
    - Start-of-string (SOS) scanners
    - **Named Group (NG) scanners**

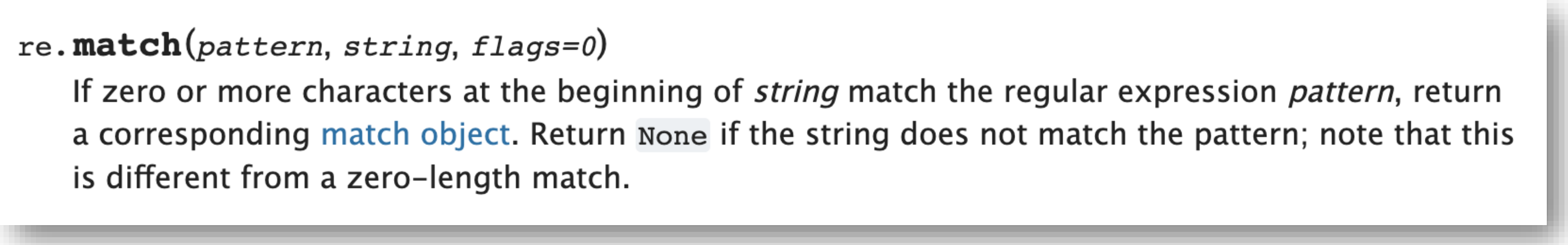
NG (Named Group) Scanner

- We will still use the `match` API call



`re.`**`fullmatch`**`(`*`pattern, string, flags=0`*`)` ¶

If the whole *string* matches the regular expression *pattern*, return a corresponding match object. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.



`re.`**`match`**`(`*`pattern, string, flags=0`*`)`

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding match object. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

NG Scanner

- Start out with token definitions
- Merge them into one RE definition

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
SINGLE_RE =
```

NG Scanner

- Start out with token definitions
- Merge them into one RE definition

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
SINGLE_RE = “[a-z]+”
```

NG Scanner

- Start out with token definitions
- Merge them into one RE definition

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
SINGLE_RE = “([a-z]+)”
```

NG Scanner

- Start out with token definitions
- Merge them into one RE definition

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
SINGLE_RE = “([a-z]+)|([0-9]+)”
```

NG Scanner

- Start out with token definitions
- Merge them into one RE definition

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

and so on

```
SINGLE_RE = “([a-z]+)|([0-9]+)|(..)|”
```

NG Scanner

- Start out with token definitions
- Merge them into one RE definition

Give each group a name corresponding to its token

```
ID     = “[a-z]+”
NUM    = “[0-9]+”
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
SINGLE_RE = “(?P<ID>[a-z]+)|
             (?P<NUM>[0-9]+)
             |(..)|”
```

NG Scanner

- Start out with token definitions
- Merge them into one RE definition

```
SINGLE_RE = “(?P<ID>[a-z]+)|
             (?P<NUM>[0-9]+)|
             (?P<ASSIGN>=)|
             (?P<PLUS>+)|
             (?P<MULT>*)|
             (?P<IGNORE> |\n)|
             (?P<SEMI>;)”
```

It’ can be a giant RE, but it can be constructed automatically.

NG Scanner

- to implement `token()`

```
SINGLE_RE = “(?P<ID>[a-z]+)|
             (?P<NUM>[0-9]+)|
             (?P<ASSIGN>=)|
             (?P<PLUS>+)|
             (?P<MULT>*)|
             (?P<IGNORE> |\n)|
             (?P<SEMI>;)”
```

Try to match the whole string to the single RE

```
“variable = 50 + 30 * 20;”
```

NG Scanner

- to implement `token()`

```
SINGLE_RE = “(?P<ID>[a-z]+)|
             (?P<NUM>[0-9]+)|
             (?P<ASSIGN>=)|
             (?P<PLUS>+)|
             (?P<MULT>*)|
             (?P<IGNORE> |\n)|
             (?P<SEMI>;)”
```

Try to match the whole string to the single RE

```
“variable = 50 + 30 * 20;”
```

Check the `group` dictionary in the result

NG Scanner

- to implement `token()`

```
SINGLE_RE = “(?P<ID>[a-z]+)|
             (?P<NUM>[0-9]+)|
             (?P<ASSIGN>=)|
             (?P<PLUS>+)|
             (?P<MULT>*)|
             (?P<IGNORE> |\n)|
             (?P<SEMI>;)”
```

Try to match the whole string to the single RE

```
“variable = 50 + 30 * 20;”
```

```
{”ID”      : “variable”
 “NUM”     : None
 “ASSIGN”  : None
 “PLUS”    : None
 “MULT”    : None
 “IGNORE”  : None
 “SEMI”    : None}
```

NG Scanner

- to implement `token()`

Try to match the whole string to the single RE

```
SINGLE_RE = “(?P<ID>[a-z]+)|
             (?P<NUM>[0-9]+)|
             (?P<ASSIGN>=)|
             (?P<PLUS>+)|
             (?P<MULT>*)|
             (?P<IGNORE> |\n)|
             (?P<SEMI>;)”
```

```
“variable = 50 + 30 * 20;”
```

```
{”ID”      : “variable”
 “NUM”     : None
 “ASSIGN”  : None
 “PLUS”    : None
 “MULT”    : None
 “IGNORE”  : None
 “SEMI”    : None}
```

NG Scanner

- to implement `token()`

```
SINGLE_RE = “(?P<ID>[a-z]+)|
            (?P<NUM>[0-9]+)|
            (?P<ASSIGN>=)|
            (?P<PLUS>+)|
            (?P<MULT>*)|
            (?P<IGNORE> |\n)|
            (?P<SEMI>;)”
```

Try to match the whole string to the single RE

```
“variable = 50 + 30 * 20;”
```

```
{”ID”      : “variable”
 “NUM”     : None
 “ASSIGN”  : None
 “PLUS”    : None
 “MULT”    : None
 “IGNORE”  : None
 “SEMI”    : None}
```

Return the lexeme `(ID, “variable”)`

NG Scanner

- to implement `token()`

```
SINGLE_RE = “(?P<ID>[a-z]+)|
             (?P<NUM>[0-9]+)|
             (?P<ASSIGN>=)|
             (?P<PLUS>+)|
             (?P<MULT>*)|
             (?P<IGNORE> |\n)|
             (?P<SEMI>;)”
```

chop!

```
“variable = 50 + 30 * 20;”
```

```
{”ID”      : “variable”
 “NUM”     : None
 “ASSIGN” : None
 “PLUS”   : None
 “MULT”   : None
 “IGNORE” : None
 “SEMI”   : None}
```

Return the lexeme `(ID, “variable”)`

NG Scanner

- to implement `token()`

chop!

```
SINGLE_RE = “(?P<ID>[a-z]+)|
            (?P<NUM>[0-9]+)|
            (?P<ASSIGN>=)|
            (?P<PLUS>+)|
            (?P<MULT>*)|
            (?P<IGNORE> |\n)|
            (?P<SEMI>;)”
```

```
“ = 50 + 30 * 20;”
```

How to deal with common prefixes in token definitions?

- Recall from SOS scanner:

How to scan this string?

“CSE110A”

```
LETTERS = “[A-Z]+”
NUM     = “[0-9]+”
CLASS   = ”CSE110A“
```

How to deal with common prefixes in token definitions?

- Convert to a single RE

How to scan this string?

“CSE110A”

```
SINGLE_RE = “
       (?P<LETTERS>([A-Z]+)|
       (?P<NUM>([0-9]+)|
       (?P<CLASS>CSE110A)”
```

How to deal with common prefixes in token definitions?

- Convert to a single RE

How to scan this string?

“CSE110A”

```
SINGLE_RE = “
       (?P<LETTERS>([A-Z]+)|
       (?P<NUM>([0-9]+)|
       (?P<CLASS>CSE110A)”
```

What do we think the dictionary will look like?

How to deal with common prefixes in token definitions?

- Convert to a single RE

```
SINGLE_RE = “
       (?P<LETTERS>([A-Z]+)|
       (?P<NUM>([0-9]+)|
       (?P<CLASS>CSE110A)”
```

How to scan this string?

```
“CSE110A”
```

```
{”LETTERS”  :  “CSE”
 “NUM”      :  None
 “CLASS”    :  None
}
```

How to deal with common prefixes in token definitions?

- Convert to a single RE

```
SINGLE_RE = “
       (?P<LETTERS>([A-Z]+)|
       (?P<NUM>([0-9]+)|
       (?P<CLASS>CSE110A)”
```

```
“CSE110A”
```

```
{”LETTERS” : “CSE”
 “NUM”      : None
 “CLASS”    : None
}
```

What does this mean?

- Tokens should not contain prefixes of each other

OR

- Tokens that share a common prefix should be ordered such that the longer token comes first

How to deal with common prefixes in token definitions?

- Careful with these tokens

```
INCR = “++”
ADD  = “+”

EQ = “==“
ASSIGN = “=“
```

Ensure that you provide them in the right order so that the longer one is first!

NG Scanner

- Pros

- Cons

NG Scanner

- Pros
  - FAST! Only 1 RE call per `token()`
- Cons
  - Requires a named group RE library
  - inter-token interactions need to be considered

Scanners we have discussed

- *Naïve Scanner*

- *RE based scanners*
  - Exact match (EM) scanners
  - Start-of-string (SOS) scanners
  - named group (NG) scanners

Which one to use?
Complex decision with performance, expressivity, and token requirements

On the next lecture:

- We will discuss token actions and how to use them to implement keywords and line numbers

- We will discuss a classic scanner generator: lex