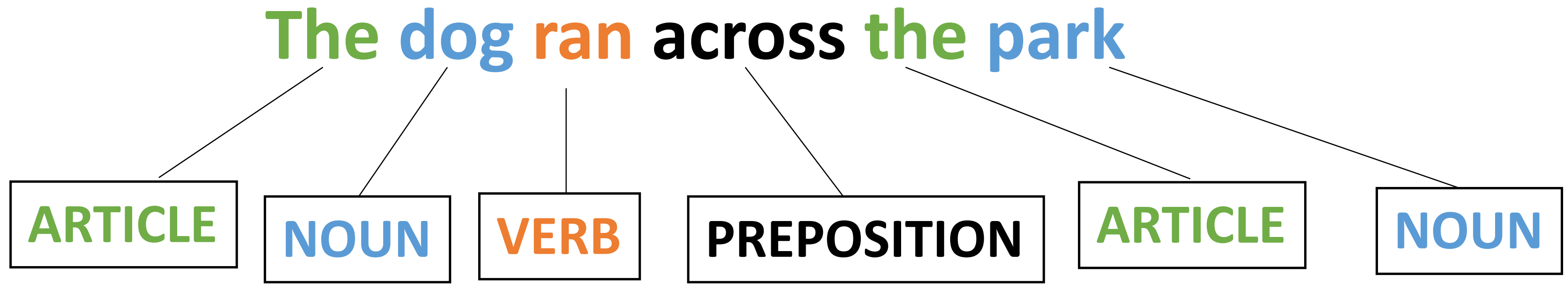
CSE110A: Compilers

April 10, 2024



- **Topics**:
  - *Finishing regular expressions*

  - *Using regular expression's in scanners*
    - Exact match scanner
    - Start-of-string Scanner
    - Named group matcher

Announcements

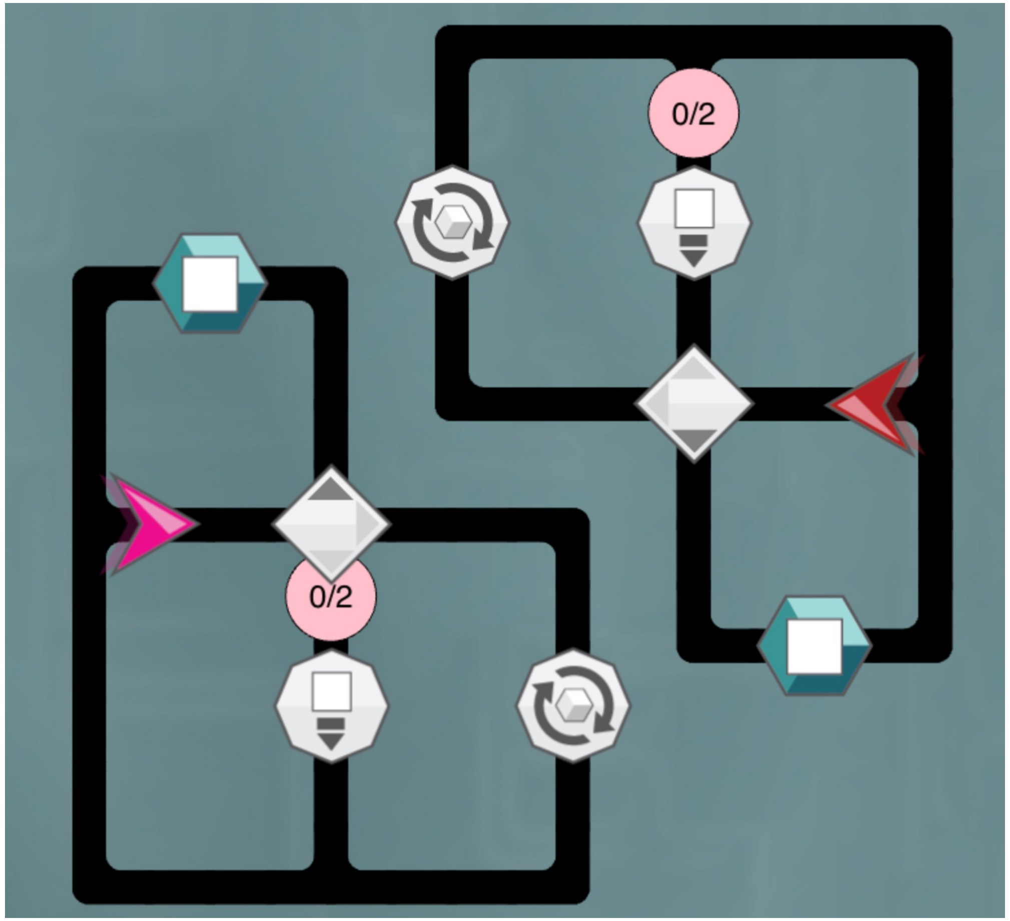
- Please enroll in Piazza!

- Homework 1 is out. You have 10 days to do it
  - Due on the 18^{th}
  - Late assignments are not accepted! Please get started
  - Good questions already on Piazza
  - You'll have what you need for part 2 and 3 today

- Lots of tutoring/office hours
  - I'll have mine tomorrow

Recruiting for for parallel programming educational game user study

- PARALLEL developed by HCI researchers at UCSC
- A game about how to use semaphores to protect resources
- Location: UCSC silicon valley campus
  - Although if there is enough interest they will move the gear to UCSC for a few days
- $30 for 160 minute (max) study
  - Tour of silicon valley campus and meeting some UCSC HCI researchers also!
- More info on Canvas



Quiz

Integer RE

The following RE is a good candidate for non-negative integers: "[0-9]*"

- ○ True
- ○ False

Integer RE

The following RE is a good candidate for non-negative integers: "[0-9]*"

- ○ True
- ○ False

Does the ”” match the RE?

Fundamental RE operators

All regular expressions can be expressed in terms of concatenation or choice operators

- ○ True
- ○ False

Fundamental RE operators

- Fundamental RE operators are:
  - Concatenate: put the regexes next to each other
  - “|” : Choice: one or the other
  - “*” : Repeat: 0 or more copies

- Practically:
  - a* roughly is the same as “” | “a” | “aa” | “aaa” ...
  - in theory, REs can accept strings of arbitrary length (not infinite strings though).
  - in practice, strings have a reasonable bound. Repeat (*) is a good abstraction though!

RE examples

which of the following strings do NOT match ac*|b*

- [ ] "" (empty string)
- [ ] ab
- [ ] acac
- [ ] acccc
- [ ] bbb

RE examples

`ac*|b*`

- “”
- “ab”
- “acac”
- “acccc”
- “bbb”

Let’s work through them

RE experiences

Have you used regular expressions before? If so, in what language or tool did you use them, and for what application?

Resuming Regular expressions

Regular expressions

- any character ‘.’

- example using email (this is probably too general!)

- ”.*@.*\.com”

Using REs

- What if we want either the domain or user name from the email?

- We can use groups!
  - use ()s to deliminate groups

- `"(.*)@(.*\\.com)"`

- Index the resulting object with [1] and [2] to get to the user name and domain respectively

Using REs

- you can give groups id names rather than using indices
- “(?P<name>.+)@(?P<domain>.+\.com)”
- (easier to copy: `"(?P<name>.+)@(?P<domain>.+\\.com)"`

Review

- Why do we want REs?

Naïve Scanner

simple string stream, peek/eat model

```
class NaiveScanner:

    def token(self):
        ...
        if self.ss.peek_char() in NUMS:
            value = ""
            while self.ss.peek_char() in NUMS:
                value += self.ss.peek_char()
                self.ss.eat_char()
            return ("NUM", value)
```

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

Shortcomings of Naïve scanner

- IDs with numbers in them?
  - `x1, y1, etc.`
  - how would you solve?

- Numbers with a decimal point in them?
  - `4.5, 9999.99998`
  - how would you solve this?

- Two character operators:
  - `++, +=`
  - how would you solve this?

Regular expressions to the rescue

Let’s write our tokens as regular expressions

- For our simple programming language

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = [“ “, “\n”]
```

Let’s write our tokens as regular expressions

- For our simple programming language

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ″=‶
PLUS   = ‶+″
MULT   = ‶*″
IGNORE = [‶ ‶, ‶\n″]
```

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

Some benefits of REs? Let’s try adding some extensions:

Let’s write our tokens as regular expressions

- For our simple programming language

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = [“ “, “\n”]
```

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

Some benefits of REs? Let’s try adding some extensions:
** increment operator?*
** digits in IDs?*

Finishing up last lecture

- A few final thoughts:

RE examples

- **What can REs not do?**

- Nested structures, such as parathesis matching:
  - Try doing arithmetic expressions
  - You will not be able to match ()s

- Classical example: REs cannot capture same number of repeats:
  - A{N}B{N}

- REs cannot parse HTML!!!
  - One of the most upvoted answers on stackoverflow!
  - https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags/1732454#1732454

How to implement an RE matcher?

- Overview: first you have to parse the RE...
  - Chicken and egg problem
  - The language of REs is not a regular language. It is context free (because it has ()s)

  - But once you can parse the RE, there are several options

How to implement an RE matcher?

- parsing with derivatives
  - We discuss this in CSE211
  - Elegant solution, but difficult to make fast

- Convert to an automata
  - Learn more about this CSE103
  - A cool website
  - https://ivanzuzak.info/noam/webapps/fsm_simulator/

New material for today

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

New material for today

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

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

Match! (ID, "v") *but what is the issue? Not the longest 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

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, ";")]
```

New material for today

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

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   = “;”
```

```
“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 are exact!

- Cons
  - SLOW! Each lexeme requires many many many calls to each RE match!

New material for today

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

SOS 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 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 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. 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

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

What about the next one

```
" = 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

How to scan this string?

Try to match on each token

```
“CSE110A”
```

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

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

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!

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

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

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 another*
- *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

New material for today

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

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

NG 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’s a giant RE, but you can construct it automatically

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;"
```

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;”
```

Check the `group` dictionary in the result

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()`

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()`

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}
```

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>;)"
```

```
"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:

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

How to scan this string?

“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

How to scan this string?

```
"CSE110A"
```

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

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

- We will discuss a classic scanner generator: lex

- See you on Friday!