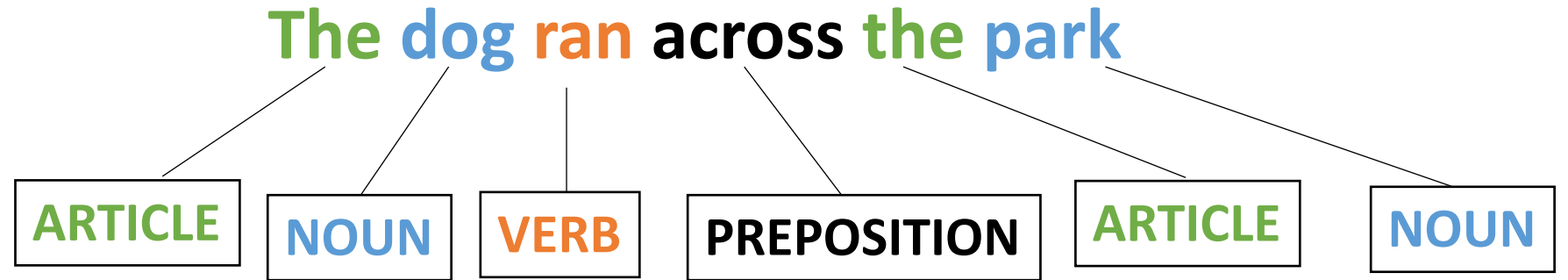


CSE110A: Compilers

April 6, 2022



- **Topics:**

- *Finishing regular expressions*
- *Using regular expressions in scanners*
 - Exact match scanner
 - Start-of-string Scanner
 - Named group matcher

Announcements

- HW 1 is released
 - By the end of today you should have most of what you need
 - We will discuss token actions on Friday which you will need for keywords
 - We've updated the spec once, there are likely more issues, please let us know!
- Due April 18 by midnight
- We have office hours this week; come see us!
- Let us know any feedback you have about the assignments; they are new!

Announcements

- My office hours:
 - Thursday, 3 - 5 PM
 - Sign-up sheet goes live around noon
 - 10 minute slots

- Other office hours:

TA Office Hours:

Mondays from 1 PM to 2 PM (Virtual)

Fridays from 2 PM to 3 PM (Room BE-151)

Yanwen's office hours will be hybrid and he will use a similar sign-up sheet.

Mentoring Hours:

Arrian is Tuesday from 1 PM to 3 PM, virtual.

Neal is Wednesday 1:30 PM - 2:30 PM, virtual; and Friday 2 PM to 3 PM sharing a room with Yanwen.

Announcements

- Please enroll in Piazza!
 - Only 50 students are enrolled. There are ~68 students in the class. People are asking good clarification questions that will help you
- Docker instructions are on the website. Let us know ASAP if you have issues

Announcements

- Homework clarifications

For part 1

- the **NUM** token can an optional decimal point “.”, but it cannot contain more than one decimal point.

Any sequence of digits with a . anywhere in it

Announcements

- Homework clarifications

For part 2 - which feeds into part 3,4

- **NUM:** a non-negative integer number. It can optionally contain a decimal point, but if it contains a decimal point, it must contain a number after the decimal point (e.g. “56.” is not allowed). A number before the decimal point is not required.

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?

Review

- Some syntactic sugar and useful interfaces

Regular expressions

- strict repeat operator: +
- one or more repeats (the * operator is 0 or more repeats)
- derivation: “r+” = “rr*”
- *Let's revisit binary numbers and decimal numbers*

“(0|1)+”

Regular expressions

- Ranges:
 - digits [0-9]
 - alpha [a-z], [A-Z]
- Derivation: [0-9] = "1|2|3|4|5|6|7|8|9"
- Lets try C style IDs: "[a-zA-Z][0-9a-zA-Z]*"
- Hexadecimal numbers: "0x[0-9a-fA-F]+"

Regular expressions

- optional operator ?
 - optional characters
- “r?” = “|r”
- Example: “ab?”
- Let’s do simple floating point numbers: “[0-9]+(\.[0-9]+)?”

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 delimitate 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)”

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?

We need a new token definition language

Can we express these tokens using REs?

• ARTICLE

• NOUN

• VERB

• ADJECTIVE

= {The, A, My, Your}

= {Dog, Car, Computer}

= {Ran, Crashed, Accelerated}

= {Purple, Spotted, Old}

Tokens

Tokens Definitions

Can we express these tokens using REs? Yes!

- ARTICLE
- NOUN
- VERB
- ADJECTIVE

= "The | A | Mine | Your"
= "Dog | Car | Computer"
= "Ran | Crashed | Accelerated"
= "Purple | Spotted | Old"

Tokens

Tokens Definitions

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 parenthesis 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 sensitive (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

What if we start “peeking” characters

```
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	=	"[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

Start the process over

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

" = 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")

Look at the code

- 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

Feed full string into each token definition

```
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

Feed full string into each token definition

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

```
"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

Chop the string

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

"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

Chop the string

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

" = 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

What about the next one

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

" = 50 + 30 * 20 ;"

1 match: IGNORE

(ID, "variable")

SOS Scanner

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

Chop the string

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

" = 50 + 30 * 20 ;"

1 match: IGNORE

(ID, "variable")

SOS Scanner

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

Chop the string

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

"= 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?

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?

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

"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

and so on

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

*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

It's a giant RE, but you can construct it automatically

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

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 ()

```
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 ()

```
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 ()

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

chop!

```
" = 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

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

How to scan this string?

"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?

- 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!

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?

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

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!