Quiz-05-EM-SOS-NG-Scanner-Actions

Quiz

When implementing a Scanner using an exact RE matcher, the number of calls to the RE matcher depends on what?

- ◯ The number of tokens
- ◯ The length of the string that is being scanned
- ◯ Both of the above
- ◯ how many operators each RE has

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;”
```

Quiz

For which scanners can token definitions be reasoned about independently (e.g. when reasoning about if they can match strings with the same prefix)

- [ ] exact match scanner
- [ ] start of string scanner
- [ ] named group scanner
- [ ] naive scanner

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   = “+”
INCR   = “++”
MULT   = “*”
IGNORE = “ |\n”
SEMI   = “;”
```

```
“variable = 50 + 30 * 20;”
```

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

- 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"
```

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}
```

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?

- Careful with these tokens

```
INCR = “++”
ADD  = “+”

EQ = “==“
ASSIGN = “=“
```

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

Quiz

For which scanners can token definitions be reasoned about independently (e.g. when reasoning about if they can match strings with the same prefix)

- [ ] exact match scanner
- [ ] start of string scanner
- [ ] named group scanner
- [ ] naive scanner

Quiz

Given C-style ids and numbers, can the following string be tokenized? If so? how many tokens will there be?

"123abc123"

- [ ] Token error
- [ ] 1 lexeme
- [ ] 2 lexeme
- [ ] 3 lexeme

tokenizing

“123abc123”

```
ID      = “[a-z][0-9a-z]+”
NUM     = “[0-9]+”
```

Quiz

Given a regular expression library, what sort of API calls would you look for in order to implement a scanner?

Regex API calls

`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.

Quiz

Which of the following are token actions NOT great for:

- ◯ Changing the value of a token
- ◯ Changing the token type
- ◯ Splitting a token into multiple tokens
- ◯ Keeping track of scanning statistics (e.g. the number of IDs seen)

Examples

Modifying a value

```
def cat_dog(x):
    if x[1] == "Cat":
        return (x[0], "Dog")
    return x
```

Modifying a token type e.g.: t is (“ID”, “float”)

```
keywords = [("INT", "int"), ("FLOAT", "float"), ("IF", "if")]

def check_keywords(t):
    keyvalues = [x[1] for x in keywords]
    if t[1] in keyvalues:
        lexeme = keywords[keyvalues.index(t[1])]
        return lexeme
    return t
```

Examples

Keeping track of statistics

```
def count_lines(x):
    if x[1] == "\n":
        s.lineno += 1
    return x
```

What other statistics might you want?

Quiz

Which of the following are token actions NOT great for:

- ○ Changing the value of a token
- ○ Changing the token type
- ○ Splitting a token into multiple tokens
- ○ Keeping track of scanning statistics (e.g. the number of IDs seen)

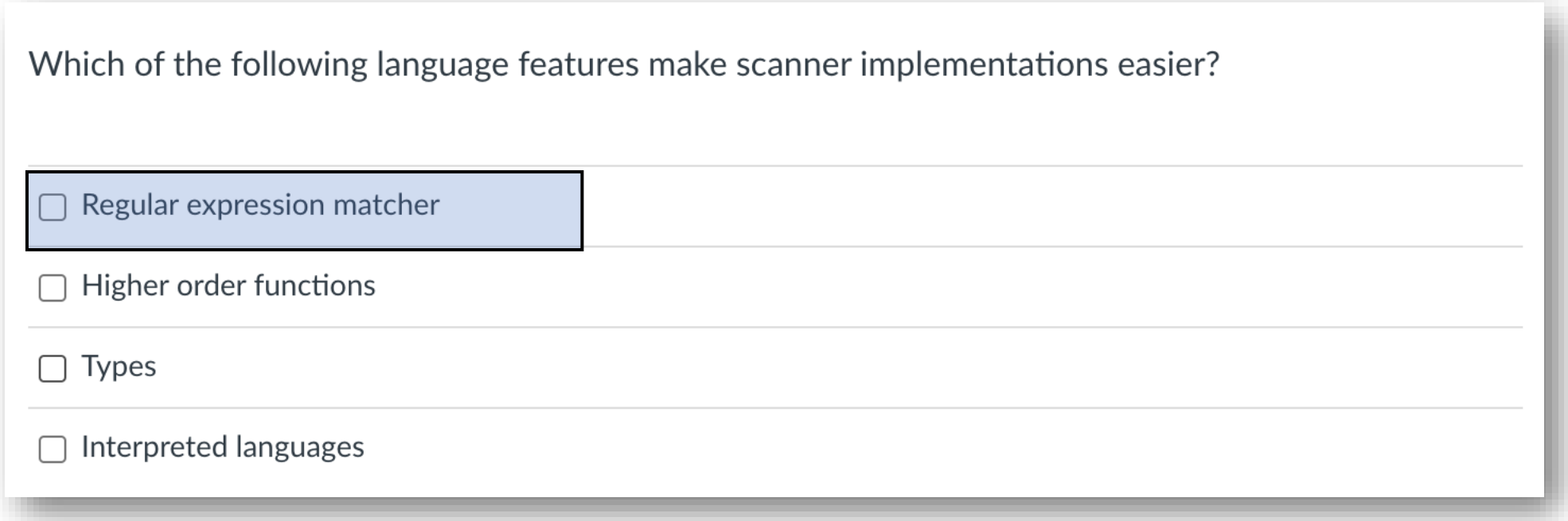
This is really difficult to do with token actions:
token actions take a single lexeme and return a single lexeme

Quiz

Which of the following language features make scanner implementations easier?

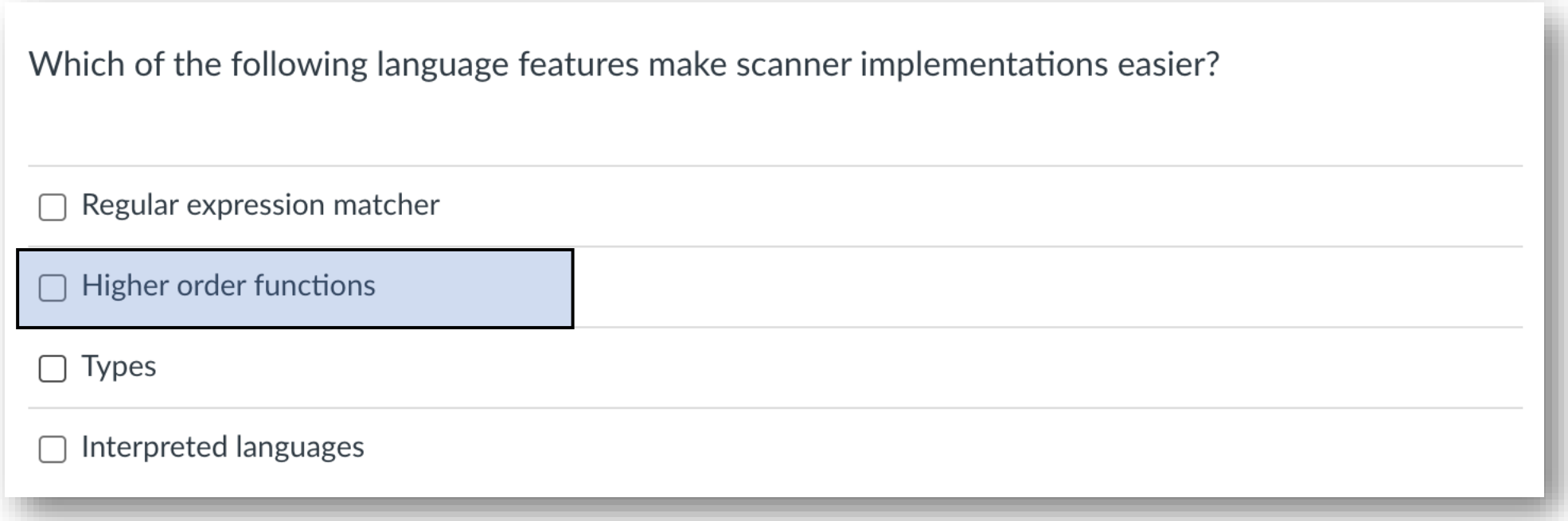
- [ ] Regular expression matcher
- [ ] Higher order functions
- [ ] Types
- [ ] Interpreted languages

Quiz



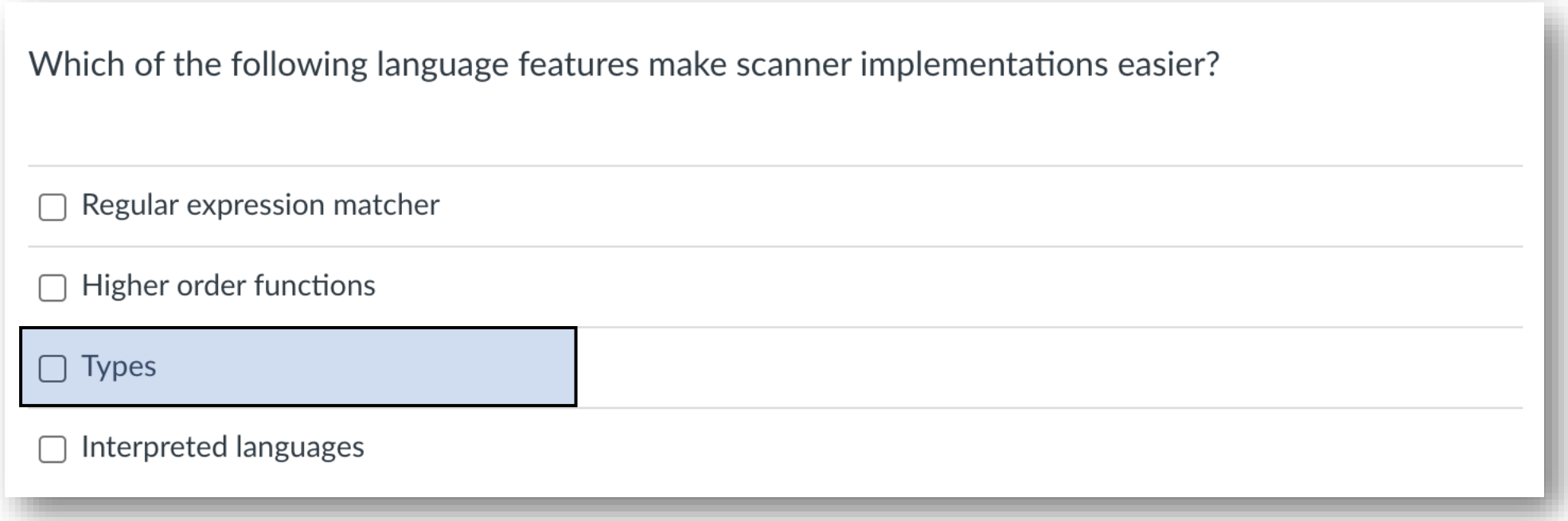
Required unless you want to write your own (take CSE211 for an example)

Quiz



Great for token actions, custom error functions, etc.

Quiz



Great for making sure your token actions are consistent. This is a shortcoming of Python

Quiz

Which of the following language features make scanner implementations easier?

- [ ] Regular expression matcher
- [ ] Higher order functions
- [ ] Types
- [ ] Interpreted languages

Doesn't really matter.
Ocaml is great for compilers (compiled)
Scheme is great for compilers (interpreted)

Quiz

If you were given a scanner that you knew was either an EM Scanner, SOS Scanner or NG scanner, and you could instantiate it with any token definitions you want, could you design an experiment (without using timing information) to determine which scanner implementation you had?