CSE110A: Compilers

April 14, 2023



- Topics:
 - Finishing up Scanners
 - Scanner actions
 - PLY Scanner

Announcements

- Remote lecture again today 😕
 - Hoping to get feeling better by Monday
 - Couldn't speak at all yesterday
 - Rescheduling office hours for Tuesday, 3 5

Office hours

TA Office Hours:

Devon is available on Tuesdays from 3:00PM to 5:00PM, virtual.

Devon will use the Zoom waiting room feature, and answer questions first-come-first-serve. Please try to keep your question within 10 minutes.

Zoom Link

Rithik is available on Fridays from 3:00PM to 5:00PM, TBD.

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

Mentoring Hours:

Arrian is available on Tuesdays and Thursdays from 1:00PM - 3:00PM, virtual.

Neal is available on Tuesdays and Thursdays, 6:30PM - 7:30PM, virtual.

Sanya is available on Wednesdays 5:30PM - 6:30PM and Fridays, 2:00PM - 3:00PM, virtual.

Announcements

- HW 1 is released
 - By the end of today you should have everything that you need
 - Please let us know of any issues you find, these are still new homeworks!
- Due in 10 days (April 24) by midnight

Quiz

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

 \bigcirc The number of tokens

 $\bigcirc\,$ The length of the string that is being scanned

 \bigcirc Both of the above

 \bigcirc 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	=	<i>"="</i>
PLUS	=	"+"
MULT	=	<i>''</i> * <i>''</i>
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	=	<i>"="</i>
PLUS	=	"+"
MULT	=	<i>''</i> * <i>''</i>
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"

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"

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.

Regex API calls

• Other considerations?

Regex API calls

- Other considerations?
 - Named groups?
 - Operators to escape?
 - How it handles choice?
 - Speed?

Finishing up scanner implementations

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

Scanners

- 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()
- 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

• 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.

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

1			
ID	=	"[a-z]+"	
NUM	=	"[0-9]+"	SINGLE_RE =
ASSIGN	=	<i>"="</i>	
PLUS	=	"+"	
MULT	=	<i>"</i> * <i>"</i>	
IGNORE	=	" \n"	
SEMI	=	// • // /	
1			

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

ID	=	"[a-z]+"
NUM	=	"[0-9]+"
ASSIGN	=	<i>"="</i>
PLUS	=	"+"
MULT	=	<i>''</i> * <i>''</i>
IGNORE	=	" \\n"
SEMI	=	// • //

SINGLE_RE =
$$"[a-z]+"$$

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

ID	=	"[a-z]+"
NUM	=	"[0-9]+"
ASSIGN	=	<i>"="</i>
PLUS	=	"+"
MULT	=	<i>''</i> * <i>''</i>
IGNORE	=	" \\n"
SEMI	=	// • //

SINGLE_RE = "
$$([a-z]+)$$
"

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

SINGLE_RE = "(
$$[a-z]+$$
) ($[0-9]+$)"

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

- 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	=	<i>"="</i>
PLUS	=	"+"
MULT	=	"*"
IGNORE	=	" \\n"
SEMI	=	// • // /

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

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

to implement token()

Try to match the whole string to the single RE

"variable = 50 + 30 * 20;"

to implement token()

Try to match the whole string to the single RE

"variable = 50 + 30 * 20;"

Check the group dictionary in the result

{"ID" : "variable" "NUM" : None "ASSIGN" : None "PLUS" : None "MULT" : None "IGNORE" : None "SEMI" : None}

"variable = 50 + 30 * 20;"

Try to match the whole string to the single RE

NG Scanner

to implement token()

 $\frac{\text{SINGLE RE}}{\text{SINGLE RE}} = "(?P < ID > [a - z] +)$ (?P<NUM>[0-9]+) (?P<ASSIGN>=) (?P<PLUS>+) (?P<MULT>*) (?P < IGNORE > | n)(?P<SEMI>;)"

'"ID" : "variable" "NUM" : None "ASSIGN" : None "PLUS" : None "MULT" : None "IGNORE" : None "SEMI" : None}

Try to match the whole string to the single RE

to implement token()

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")

Return the lexeme (ID, "variable")

{ "ID"	:	"variable"
"NUM"	:	None
"ASSIGN"	:	None
"PLUS"	:	None
"MULT"	:	None
"IGNORE"	:	None
"SEMI"	:	None}

chop!

(?P<MULT>*)

(?P<SEMI>;)"

(?P < IGNORE > | n)

to implement token()

to implement token()

chop!
• Recall from SOS scanner:

LETTERS

NUM

CLASS

How to scan this string?

=	"[A-Z]+"
=	"[0-9]+"

= "CSE110A"

"CSE110A"

• Convert to a single RE

How to scan this string?

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

"CSE110A"

• 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?

• Convert to a single RE

How to scan this string?

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

"CSE110A"

{"LETTERS"	:	"CSE"
"NUM"	:	None
"CLASS"	:	None
}		

• Convert to a single RE

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

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

"CSE1	L10A"
-------	-------

{"LETTERS"	:	"CSE"
"NUM"	:	None
"CLASS"	:	None
}		

• Careful with these tokens

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

In practice

- Most scanner generators that I am aware of have SOS semantics
 - You can reason about tokens independently
 - Use fast "match" implementations under the hood
- Mainstream compilers:
 - have hand coded and hand optimized scanners
 - _very_ fast
 - _very_ hard to modify
 - Only worth it to do this if you have the need and time

Moving on

- Token actions
 - Replacement
 - Keywords
 - Error reporting
- Scanner error recovery

Moving on

Token actions

- Replacement
- Keywords
- Error reporting
- Scanner error recovery

First class functions

- A programming language is said to have first class functions if functions can be stored as variables
- Python has great support for this
- Functional languages have great support (and compiler helps out by checking types)
- In C++
 - Classically: function pointers
 - Newer: supports lambdas

Functions as part of a token definition

- In our scanners, we give them as the 3rd element in the token tuple definition
- A token action takes in a lexeme and returns a lexeme.
 - Possibly the same lexeme

They generally do three things:

- modify a token
- refine a token
- modify the scanner state

Functions as part of a token definition

- Once a token is matched, its token action is called on its lexeme,
- and the lexeme it returns is returned from the scanner,
- Code example in the EM

Examples

Token actions generally do three things:

- modify a token
- refine a token
- modify the scanner state

• Example using natural language

• PRONOUN

NOUN

- {His, Her, Their}
- VERB =
- ADJECTIVE
- {Dog, Cat, Car, Park}
- {Slept, Ate, Ran}
 - {Purple, Spotted, Old} =

Tokens

Tokens Definitions

• PRONOUN

- NOUN
- VERB

• ADJECTIVE

= {His, Her, Their}

- = {Dog, Cat, Car, Park}
- = {Slept, Ate, Ran}
- = {Purple, Spotted, Old}

Example: Can change any pronoun value to gender neutral ("Their")

Tokens

Tokens Definitions

• Example using types

Some ML frameworks have **float16** types in their programming languages

Many devices only support **float** types (32 bit)

float16 x, y;
return x+y;

Scanner can easily change float16 to float with a token action



Examples

Token actions generally do three things:

- modify a token
- refine a token
- modify the scanner state

Keywords: (finally!)

Keywords

TOKENS					
ID	=	[a-z]+			
NUM	=	[0-9]+			
ASSIGN	=	<i>"="</i>			
PLUS	=	"+"			
MULT	=	<i>4</i> ★ <i>1</i>			
IGNORE	=	["","\n"]			
KEYWORDS					
[(INT,'	'ir	t"), (FLOAT, "float")]			

Keywords

TOKENS						
ID	=	[a-z]+				
NUM	=	[0-9]+				
ASSIGN	=	<i>"="</i>				
PLUS	=	"+"				
MULT	=	<i>" * "</i>				
IGNORE	=	["","\n"]				
<u>KEYWORDS</u>						
[(INT,"	ʻin	t"), (FLOAT, "float")	•••]			

Code example in EM Scanner

Examples

Token actions generally do three things:

- modify a token
- refine a token
- modify the scanner state

Modifying state

Our big use case here is error reporting

- Line number
- Column number

Doesn't work in our homework

- Our homework has scanners import tokens
- Usually it is the other way around!!
- Maybe some of you can think of a design where it does work in our homework

Modifying state

In the common case, we can create a scanner and then update a class member in a token action

EM Scanner example:

Advanced topic

- Recovering from errors (syntax highlighting)
 - show Godbolt example
 - use the command line option: -fsyntax-only -Xclang -dump-tokens
 - try to tokenize weird symbols, such as `
- return an error token and try to recover
 - eating one character
 - eating until a space
 - eating until a newline

On Monday

- Enjoy your weekend!
- We will be starting Module 2 on parsing!

Next topic

- Using a scanner generator:
 - They have their own designs and it is important to understand trade-offs and design decisions
- Classically:
 - Lex and Flex
- Modern:
 - Antlr (ANother Tool for Language Recognition)
- A good in-between:
 - PLY a Lex and Yacc implementation in Python



- Old tools input is a token specification file. Produces a complicated C file that you would include in your project
- New language technology makes things a lot easier (higher order functions, fast RE matchers, etc.)

- written mostly for education purposes. Uses only core python features
- Personally, I have used it many times for little compiler projects
- Documented to be a python implementation of Lex, but uses a much nicer interface

How to use PLY's Scanner

Scanner Demo

• Library import

import ply.lex as lex

• Token list

tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE"]

• Token specification

t_ADJECTIVE = "old|purple|spotted"
t_NOUN = "dog|computer|car"
t_ARTICLE = "the|my|a|your"
t_VERB = "ran|crashed|accelerated"

Scanner Demo

• Build the lexer

lexer = lex.lex() what happens?

• Need an error function

```
# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    exit(1)
```

Scanner Demo

• Now give the lexer some input

lexer.input("dog")

• The lexer streams the input, we need to stream the tokens:


• try a longer string:

lexer.input("dog computer")

What happens?

• Need to add a token for whitespace!

tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "WHITESPACE"]
...
t_WHITESPACE = '\ '

• Now we can lex:

```
LexToken(NOUN, 'dog',1,0)
LexToken(WHITESPACE, ' ',1,3)
LexToken(NOUN, 'computer',1,4)
```

• Now we can do a sentence

lexer.input("my spotted dog ran")

```
LexToken(ARTICLE, 'my',1,0)
LexToken(WHITESPACE, '',1,2)
LexToken(ADJECTIVE, 'spotted',1,3)
LexToken(WHITESPACE, '',1,10)
LexToken(NOUN, 'dog',1,11)
LexToken(WHITESPACE, '',1,14)
LexToken(VERB, 'ran',1,15)
```

Can we clean this up?

• We can ignore whitespace

```
#t_WHITESPACE = '\
t_ignore = ' '
```

gets simplified to:

```
LexToken(ARTICLE, 'my',1,0)
LexToken(WHITESPACE, ',1,2)
LexToken(ADJECTIVE, 'spotted',1,3)
LexToken(WHITESPACE, ',1,10)
LexToken(NOUN, 'dog',1,11)
LexToken(WHITESPACE, ',1,14)
LexToken(VERB, 'ran',1,15)
```

LexToken(ARTICLE, 'my',1,0) LexToken(ADJECTIVE, 'spotted',1,3) LexToken(NOUN, 'dog',1,11) LexToken(VERB, 'ran',1,15)

• What about newlines?

lexer.input("""
my spotted dog ran
the old computer crashed
""")

• Need to add a newline token!

• What about newlines?

lexer.input("""
my spotted dog ran
the old computer crashed
""")

• Need to add a newline token!

tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "NEWLINE"]
t_NEWLINE = "\\n"

```
LexToken(NEWLINE, '\n',1,0)
LexToken(ARTICLE, 'my',1,1)
LexToken(ADJECTIVE, 'spotted',1,4)
LexToken(NOUN, 'dog',1,12)
LexToken(VERB, 'ran',1,16)
LexToken(NEWLINE, '\n',1,19)
LexToken(ARTICLE, 'the',1,20)
```

Line numbers are not updating

• Token actions, similar to production actions

 $t_NEWLINE = " \setminus n"$

Changes into:

```
def t_NEWLINE(t):
    "\\n"
    t.lexer.lineno += 1
    return t
```

docstring is the regex, lexer object which has a linenumber attribute.

If we don't return anything, then it is ignored.

• Example: changing gendered pronouns into gender neutral pronouns

tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "NEWLINE", "PRONOUN"]
t_PRONOUN = "her|his|their"

lexer.input("""
his spotted dog ran
her old computer crashed
""")

• Add a token action:

```
def t_PRONOUN(t):
    "her|his|their"
    if t.value in ["his", "her"]:
        t.value = "their"
    return t
```

Now output will have all gender neutral pronouns!