CSE 450
Translation of Programming Languages

Lecture 2: Lexical Analysis with Lex
Administration

• If you forgot your index cards, grab another set now.
• You need to fill out the GitHub survey **tonight**!
  • I need the info to create your private repos.
• Next lecture (Thursday), I'll be teaching you git Version Control, and you should bring a laptop.
  • You need this info to start Project 1.
How well do you know C++?

Have you used the **down-to** operator?  -->

```cpp
#include <iostream>

int main()
{
    int x = 10;
    while (x --> 0) {
        std::cout << x << std::endl;
    }
}
```
Structure Today! of a Compiler

Source Language

Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Int. Code Generator

Intermediate Code

Code Optimizer
Target Code Generator

Target Language

Front End

Back End
What exactly is lexing?

Consider the code:

```c
int z=123;
z*=7;
if(z>3)
    print z-91;
```

This is really nothing more than a string of characters:

```c
int z=123; \n z*=7; \n if(z>3) \n \t print z-91;
```

We must divide this string into meaningful sub-strings.
A **token** is a category of meaningful substrings in the source language. In English this would be types of words or punctuation, such as a **noun**, **verb**, **adjective**, or **end-mark**.

In a program or script, this could be an **identifier**, a **floating point number**, a **math symbol**, a **keyword**, etc.

A substring that represents an instance of a token is called a **lexeme**. The class of possible lexemes in a token is described by a **pattern**.

Patterns are typically defined using **regular expressions**.
Example: identifying *integers* in the code.

Token: **STATIC_INT**

Pattern: `[0-9]+` *(a series of one or more digits)*

Lexemes:
Example: identifying integers in the code.

Token: **STATIC_INT**

Pattern: `[0-9]+`  
(a series of one or more digits)

Lexemes: **123**
Example: identifying integers in the code.

Token: **STATIC_INT**

Pattern: `[0-9]+`  *(a series of one or more digits)*

Lexemes: **123, 7**
Example: identifying integers in the code.

Token: **STATIC_INT**

Pattern: `[0-9]+`  *(a series of one or more digits)*

Lexemes: **123, 7, 3**
Example: identifying integers in the code.

Token: **STATIC_INT**

Pattern: `[0-9]+`  (a series of one or more digits)

Lexemes: **123, 7, 3, 91**
int z = 123; // z* = 7; if (z > 3) printf z - 91;

Can we just search for one type of token at a time? No!

Consider:

```c
int my_var2;

string test_str = "This is test #12";

x = 23; // Set x to the value 23.
```
Can we just search for one type of token at a time?  

**No!**

Consider:

```
int my_var2;

string test_str = "This is test #12";

x = 23;  // Set x to the value 23.
```
What is a token?

A meaningful substring of a source language

A series of characters

A number, word, identifier, etc.

Something you put in arcade machines
Identifying Tokens

A lexical analyzer must be able to do three things:

1. Correctly identify all tokens within a string
2. Discard uninformative tokens (whitespace and comments)
3. Return each remaining token with its lexeme and the line number it was found on
Why would we care about line number?

- Line number is useful for \texttt{goto} jumps
- Line number is useful for debugging your compiler
- Line number is useful for debugging your source code
- Line number is useful for instructors to take hard earned points away from students
```plaintext
int z = 123; \nz* = 7; \nif (z > 3) \n\tprint z - 91;
```

Patterns:

- **CONDITION**
- **TYPE** int|float|char
- **COMMAND** print|return|define
- **STATIC_INT** [0-9]+
- **ID** [a-zA-Z][a-zA-Z0-9]*
- **MATH** [+-*/()]
- **ASSIGN** =
- **CASSIGN** [+-*/]=
- **COMPARE** "==" | "!=" | "<" | ">" | "<=" | ">="
- **WHITESPACE** [ \t\n]
- **ENDLINE** ;
- **UNKNOWN** .
int z = 123; \n z *= 7; \n if (z > 3) \n \t print z - 91;

Current lexeme: i

Patterns:
CONDITION if
TYPE int|float|char
COMMAND print|return|define
STATIC_INT [0-9]+
ID [a-zA-Z][a-zA-Z0-9]*
MATH [+\-*][0-9]
ASSIGN =
CASSIGN [+\-*][=]
COMPARE "==" | "!=" | "<" | ">" | "<=" | ">="
WHITESPACE [ \t\n]
ENDLINE ;
UNKNOWN .
int z = 123; 
z* = 7; 
if (z > 3) 
print z - 91;

Current lexeme: i

Which option do we choose??

Patterns:
- **CONDITION**: if
- **TYPE**: int|float|char
- **COMMAND**: print|return|define
- **STATIC_INT**: [0-9]+ 
- **ID**: [a-zA-Z][a-zA-Z0-9]* 
- **MATH**: [+-*/()] 
- **ASSIGN**: =
- **CASSIGN**: [+-*/]= 
- **COMPARE**: "==" | "!=" | "<" | ">" | "<=" | ">="
- **WHITESPACE**: [ \t\n] 
- **ENDLINE**: ; 
- **UNKNOWN**: .
Lookahead will be important for lexical analysis.

Tokens are typically read in from left-to-right, recognized one at a time from the input string.

It is not always possible to instantly decide if a token is finished without looking ahead at the next character. For example...

Is 'i' a variable, or the first character of 'if' or 'int'?

Is '=' an assignment, or the beginning of a comparison '=='?
Lookahead example

Some languages require more lookahead than others. For example: Fortran removes all whitespace before processing and thus cannot get clues from it.

```
DO 5 I = 1.25
'D05I' is a variable!

DO 5 I = 1,25
Here, 'DO' is a keyword!
```
Uglier Lookahead example

PL/I is another example of a difficult to lex language because it allows identifiers to be the same as keywords. Consider this legal statement:

```
IF THEN THEN THEN = ELSE;
ELSE ELSE = THEN;
```

`ELSE` and `THEN` were both previously defined as variables.

```
IF THEN THEN THEN = ELSE;
ELSE ELSE = THEN;
```
Details...

How should we figure out ambiguities?

If we see 'interact', are we declaring an integer called 'eract' or making use of a previously defined identifier by the name 'interact'?

We need rules that ensure only one possible answer, ideally with only one character of lookahead.
int z = 123; \n z *= 7; \n if (z > 3) \n \t print z - 91;

Current lexeme: i

Which option do we choose??

Take the first one that matches!
int z = 123; \n z *= 7; \n if (z > 3) \n \t print z - 91;

Current lexeme: int

Patterns:
- CONDITION: if
- TYPE: int|float|char
- COMMAND: print|return|define
- STATIC_INT: [0-9]+
- ID: [a-zA-Z][a-zA-Z0-9]*
- MATH: [+\-*/()]
- ASSIGN: =
- CASSIGN: [+\-*/]=
- COMARE: "==" | "!=" | "<" | "<=" | "<=" | ">" | ">="
- WHITESPACE: [ \t\n]
- ENDLNE: ;
- UNKNOWN: .
int z = 123; \n z* = 7; \n if (z > 3) \n \n \n int

Current lexeme: int

Patterns:
- CONDITION: if
- TYPE: int|float|char
- COMMAND: print|return|define
- STATIC_INT: [0-9]+
- ID: [a-zA-Z][a-zA-Z0-9]*
- MATH: [+\-*/(\)]
- ASSIGN: =
- CASSIGN: [+\-*/]=
- COMPARE: "==" | "!=" | "<" | ">" | "<=" | ">="
- WHITESPACE: [ \t\n]
- ENDLIE
- UNKNOWN
.
int z = 123; \n z* = 7; \n if (z > 3) \n \t print z - 91;

**TYPE(“int”)**

Current lexeme: _
int z = 123; nz* = 7; if (z > 3) printf z - 91;
int z = 123; \n z* = 7; \n if (z > 3) \n \t print z - 91;

Patterns:
CONDITION if
TYPE int|float|char
COMMAND print|return|define
STATIC_INT [0-9]+
ID [a-zA-Z][a-zA-Z0-9]*
MATH [+\-*/()]
ASSIGN =
CASSIGN [+\-*/]=
COMPARE "==" | "!=" | "<" | ">" | "<=" | ">="
WHITESPACE [ \t\n]
ENDLINE ;
UNKNOWN .
int z = 123; \n z* = 7; \n if (z > 3) \n \t print z - 91;

TYPE("int")
WHITESPACE("_")
ID("z")
ASSIGN("=")
Current lexeme: 1

Patterns:
CONDITION if
TYPE int|float|char
COMMAND print|return|define
STATIC_INT [0-9]+
ID [a-zA-Z][a-zA-Z0-9]*
MATH [+\-*/()]
ASSIGN =
CASSIGN [+\-*/]=
COMPARE "==" | "," | "<" | ">" | "<=" | ">="
WHITESPACE [ \t\n]
ENDLINE ;
UNKNOWN .
int z = 123; \n z* = 7; \n if (z > 3) \n \t print z - 91;

Patterns:
CONDITION if
TYPE int|float|char
COMMAND print|return|define
STATIC_INT [0-9]+
ID [a-zA-Z][a-zA-Z0-9]*
MATH [+\-*/()] 
ASSIGN =
CASSIGN [+\-*/]=
COMPARE "=="|"!="|"<"|">"|"<="|">="
WHITESPACE [ \t\n]
ENDLINE ;
UNKNOWN .
int z = 123; \n z* = 7; \n if (z > 3) \n \t print z - 91;

Patterns:
- **TYPE** ("int")
- **WHITESPACE** ("_")
- **ID** ("z")
- **ASSIGN** ("=")

Current lexeme: 123
int z = 123; \n z* = 7; \n if (z > 3) \n \t print z - 91;

Patterns:
CONDITION if
TYPE int|float|char
COMMAND print|return|define
STATIC_INT [0-9]+ 
ID [a-zA-Z][a-zA-Z0-9]* 
MATH [+\-*/(] 
ASSIGN = 
CASSIGN [+\-*/]=
COMPARE "==" | "!=" | "<" | ">" | "<=" | ">=" 
WHITESPACE [ \t\n] 
ENDLINE ; 
UNKNOWN .
int z = 123; \n z* = 7; \n if ( z > 3 ) \n \tprint z - 91;

Patterns:
- CONDITION if
- TYPE int|float|char
- COMMAND print|return|define
- STATIC_INT [0-9]+  
- ID [a-zA-Z][a-zA-Z0-9]*
- MATH [+\-*/()]  
- ASSIGN =
- CASSIGN [+\-*/]=
-COMPARE "==" | "!=" | "<" | ">" | "<=" | ">="

WHITESPACE [ \t\n]  
ENDLINE ;
UNKNOWN .
\texttt{int\_z = 123; \textbackslash n z* = 7; \textbackslash n if (z > 3) \textbackslash n \textbackslash t print z - 91;}

Patterns:
- **CONDITION**: if
- **TYPE**: int|float|char
- **COMMAND**: print|return|define
- **STATIC\_INT**: [0-9]+
- **ID**: [a-zA-Z][a-zA-Z0-9]*
- **MATH**: [+\-*/()]
- **ASSIGN**: =
- **CASSIGN**: [+\-*/]=
- **COMPARE**: "==" | "!=" | "<" | ">" | "<=" | ">="
- **WHITESPACE**: [ \t\n]
- **ENDLINE**: ;
- **UNKNOWN**: .
int z = 123; \n z* = 7; \n if (z > 3) \n \t print z - 91;
int z = 123; \n z* = 7; \n if ( z > 3 ) \n \t printf( z - 91 );
int z = 123; \n z *= 7; \n if (z > 3) \n \t print z - 91;
Using Regular Expressions

Can we test if an input string fits into a regular expression? How? And is that enough?

We need to divide up an input stream into lexemes.

1. Design a set of regular expressions.
2. Convert to a Non-deterministic Finite Automata (NFA).
3. Convert to a Deterministic Finite Automata (DFA).
4. Convert to a table that we can easily implement.

And we're done! (We'll pick most of this up later...)
Using Regular Expressions

Can we test if an input string fits into a regular expression? How? And is that enough?

We need to divide up an input stream into lexemes.

1. Design a set of regular expressions.
2. Plug them into Lex

And we’re done! (We’ll pick most of this up later...)
3 Minute Break

Project 1: Lexical Analysis

- Correctness is job #1.
  - And job #2 and #3!

- Remember that this project will be used in all future projects.
- Tips on building large systems (such as our compiler):
  - Keep it simple
  - Design systems that can be tested
  - Don't optimize prematurely
  - It is easier to modify a working system than to get a system working
The Simplest PLY Program

In Lex, you provide a set of regular expression and the action that should be taken with each.

PLY is a library for using Lex (and later Yacc) in Python.

For example, what does this PLY program do?:

```
import ply.lex as lex

tokens = ('CHARACTER', )
def t_CHARACTER(t):
    r'.|
    return t
data = '''
3 + 4 * 10
+ -20 *
2
'''

# Build the lexer
lexer = lex.lex()

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break  # No more input
    print(tok)
```
Running Lex (With PLY)

First, you need to import the lex class from the ply library:

```python
import ply.lex as lex
```

Then, you need to supply a tuple of all the token types for your language:

```python
tokens = ('CHARACTER', )
```

Next, you need to supply RegEx's to denote what input qualifies as a token:

```python
def t_CHARACTER(t):
    r'.|
    return t
```
Lastly, you need to supply the lexer with input and query it for tokens:

```python
data = '''
3 + 4 * 10
    + -20 * 2
'''

# Build the lexer
lexer = lex.lex()

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break  # No more input
    print(tok)
```
A slightly more complex program

tokens = ('DAYS', 'OTHER', "WHITESPACE")

def t_WHITESPACE(t):
    r'[^\t\n ]'
    pass

def t_DAYS(t):
    r'Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday'
    print(t.value + " is a day")

def t_OTHER(t):
    r'[^a-zA-Z]+'
    print(t.value + " is not a day")

data = '''
Monday Someday
Tuesday Stop
'''
A tiny bit more...

tokens = ('WEEKDAY', 'WEEKEND', 'OTHER', "WHITESPACE")

def t_WHITESPACE(t):
    r'[	
    \n ]'
    pass

def t_WEEKDAY(t):
    r'''Monday|Tuesday|Wednesday|
    Thursday|Friday'''
    print(t.value + " is a week day")

def t_WEEKEND(t):
    r'Saturday|Sunday'
    print(t.value + " is a week end day")

def t_OTHER(t):
    r'[a-zA-Z]+'
    print(t.value + " is not a day")

data = ''
Monday Someday
Tuesday Stop Saturday
''
Counting Words

```python
CHAR_COUNT = 0
WORD_COUNT = 0
LINE_COUNT = 0
tokens = ('word', 'eol', 'other')
def t_word(t):
    r'^\t
global WORD_COUNT
global CHAR_COUNT
WORD_COUNT += 1
CHAR_COUNT += len(t.value)
def t_eol(t):
    r'\n'
global CHAR_COUNT
global LINE_COUNT
CHAR_COUNT += 1
LINE_COUNT += 1
def t_other(t):
    r'.'
global CHAR_COUNT
CHAR_COUNT += 1
data = "one
two three"
# Build the lexer
import ply.lex as lex
lexer = lex.lex()
# Give the lexer some input
lexer.input(data)
# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break  # No more input
    print(tok)
    print("line count = {}").format(LINE_COUNT))
    print("word count = {}").format(WORD_COUNT))
    print("char count = {}").format(CHAR_COUNT))```
Designing Patterns

Designing the proper regular expressions for patterns can be tricky, but you are provided with a broad range of building blocks.

- A dot will match any single character except a newline.
- Star and plus are used to match zero/one or more of the preceding expressions.
- Matches zero or one copy of the preceding expression.
- A logical 'or' statement - matches either the pattern before it, or the pattern after it.
Brackets are used to denote a character class, which matches any single character within the brackets. If the first character is a '^', this negates the brackets, causing them to match any character except those listed. The '-' can be used in a set of brackets to denote a range. Escape sequences must use a '\'.

```
[ ]
```

Match everything within the quotes literally - doesn't use any special meanings for characters.

```
""
```

Group everything within the parentheses as a single unit for the rest of the expression.

```
()
```
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{x}</code></td>
<td>Match the preceding expression $x$ times.</td>
</tr>
<tr>
<td><code>{x,y}</code></td>
<td>Match the preceding expression $x$ to $y$ times ($x &lt; y$)</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Matches the very beginning of a line.</td>
</tr>
<tr>
<td><code>$</code></td>
<td>Matches the end of a line.</td>
</tr>
</tbody>
</table>
Example Patterns

\[[0-9]\]
A single digit.

\[[0-9]+\]
An integer.

\[[0-9]+(\.[0-9]+)\]?
An integer or floating point number (always starting with a digit).

\[\+[\-]?[0-9]+(\.[0-9]+)\]?\((eE)\+[\-]?[0-9]+\)?
Positive or negative integer, floating point, or scientific notation.
More patterns…

What regular expression can we use to detect comments?

#.*

What about literal strings?

Does this work? ".*"

What about: "[^]*"

No multi-line strings: "[^\n]*"

Allow escape chars: "([\n\t]|[^\s])*"
Problems for you

Give a regular expression that defines all email addresses?

What sets of binary strings are defined by the following regular expression:

\[ 0 \ (0 \mid 1)^* \ 0 \]

\[ (0 \mid 1)^* \ 0 \ (0 \mid 1)^* \]

\[ 0^? \ 1^+ \]

\[ (0^? \ 1^*)^+ \]
More problems for you

Given the regular expressions:

\[ r_1 = 0 (10)^* 1^+ \]
\[ r_2 = (01)^+ 0? \]

1. Find a string corresponding to both \( r_1 \) and \( r_2 \).
2. Find a string corresponding to \( r_1 \) but not \( r_2 \).
3. Find a string corresponding to \( r_2 \) but not \( r_1 \).
4. Find a string corresponding to neither \( r_1 \) nor \( r_2 \).