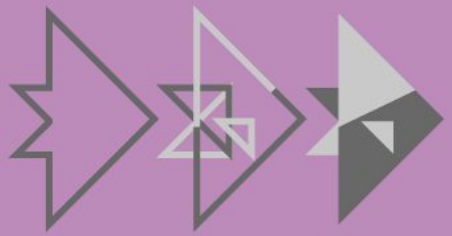
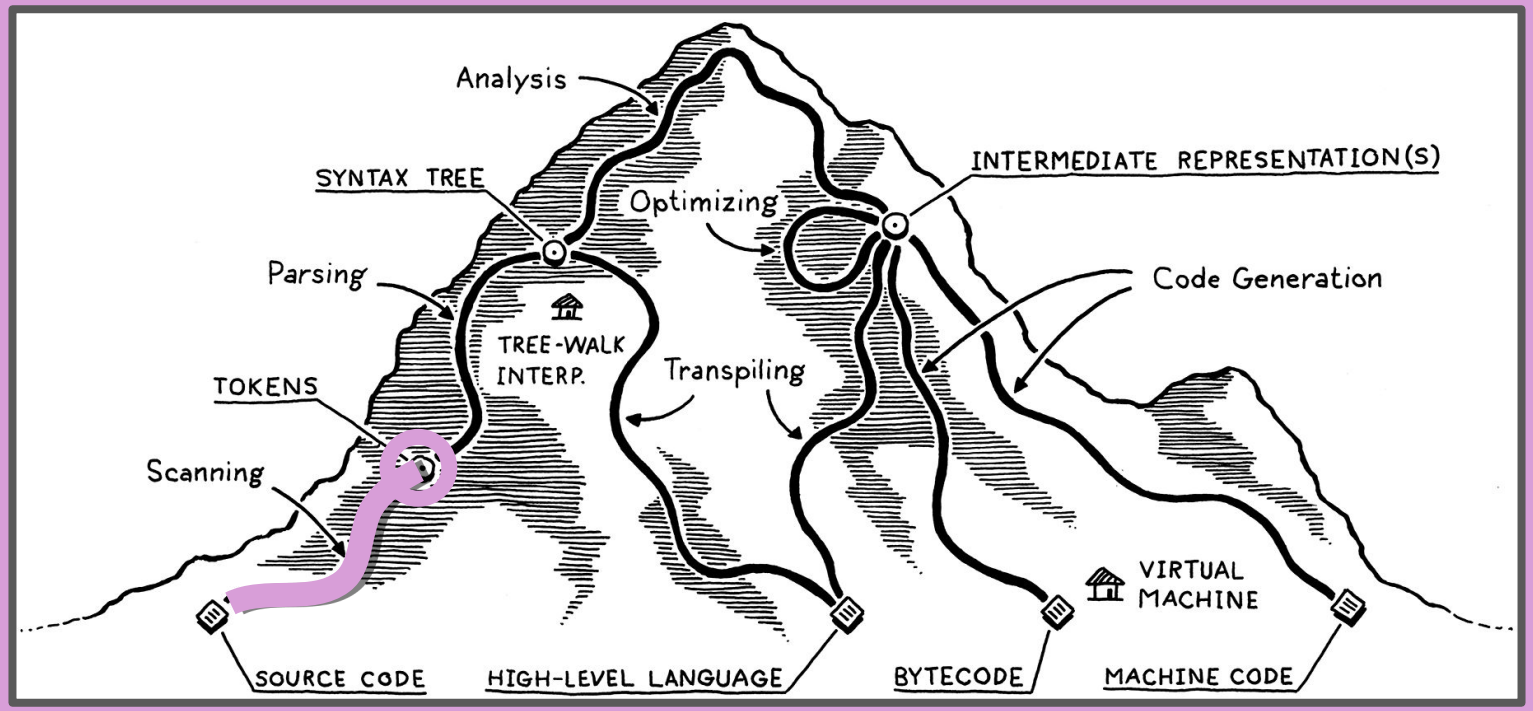
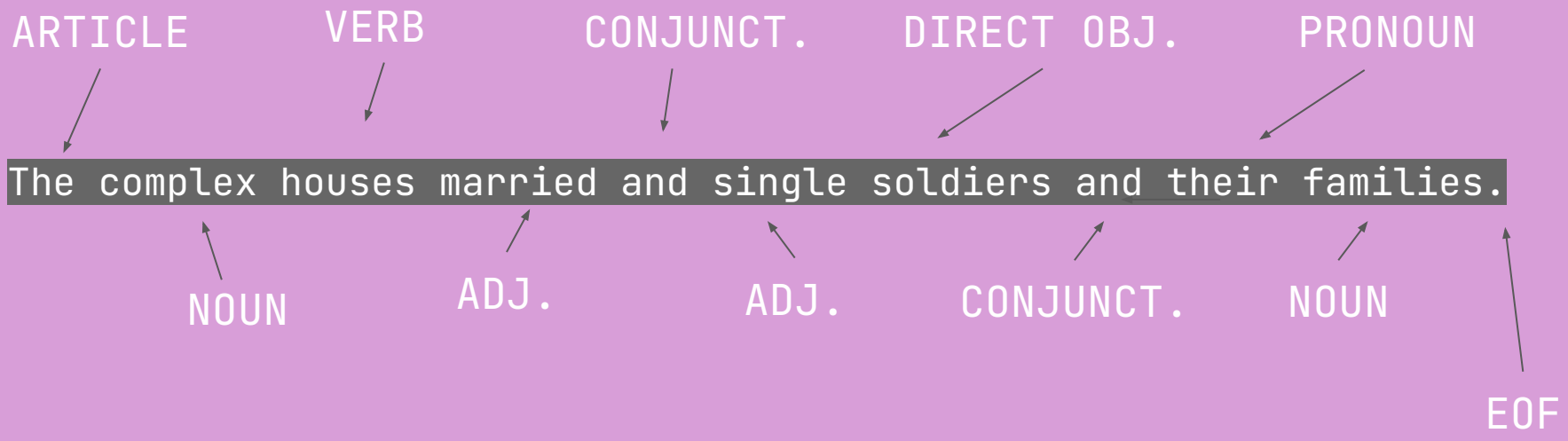


CMPSC 201: PROGRAMMING LANGUAGES


Analysis
SYNTAX TREE
Optimizing
INTERMEDIATE REPRESENTATION(S)
Parsing
Code Generation
TREE-WALK
INTERP.
TOKENS
Transpiling
Scanning
VIRTUAL
MACHINE
SOURCE CODE
HIGH-LEVEL LANGUAGE
BYTECODE
MACHINE CODE

Lexical Analysis: Review



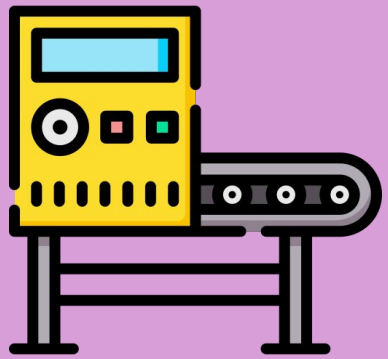
Lexical Analysis: Review

FUNCTION	FORM
ART	THE
NOUN	COMPLEX
VERB	HOUSES
ADJECTIVE	MARRIED
CONJUNCTION	AND
ADJECTIVE	SINGLE
SOLIDERS	SOLDIERS
CONJUNCTION	AND
PRONOUN	THEIR
NOUN	FAMILIES
EOF	.

Lexical Analysis (“Lexing”)

```
var num = 3.14;
print num;
```

Error



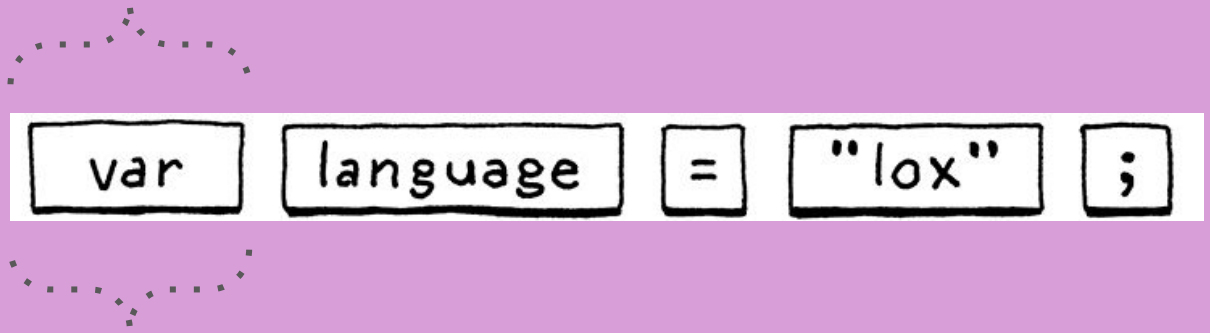
Character Stream

Token Stream

```
VAR var null
IDENTIFIER num null
EQUAL = null
NUMBER 3.14 3.14
SEMICOLON ; null
PRINT print null
IDENTIFIER num null
SEMICOLON ; null
EOF  null
```

Lexical Analysis: Lexemes vs. Tokens

FORM = LEXEME

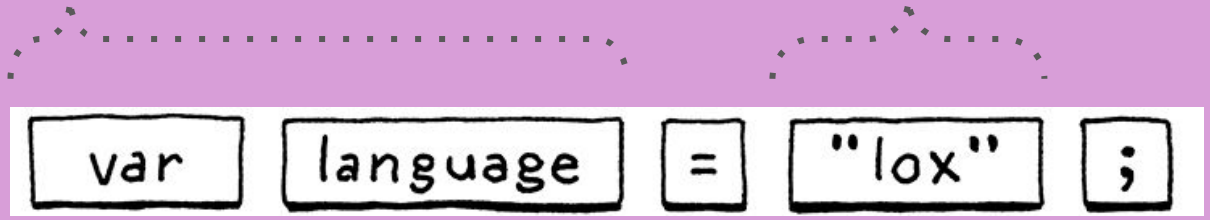


FUNCTION = TOKEN

Lexical Analysis: Regular Expressions

`var\s[a-z0-9_]+`

`(?<=(?P<quote>"|'))([a-z0-9_+]+)(?:(?(quote)))`



Typically might be handled by regular expressions...

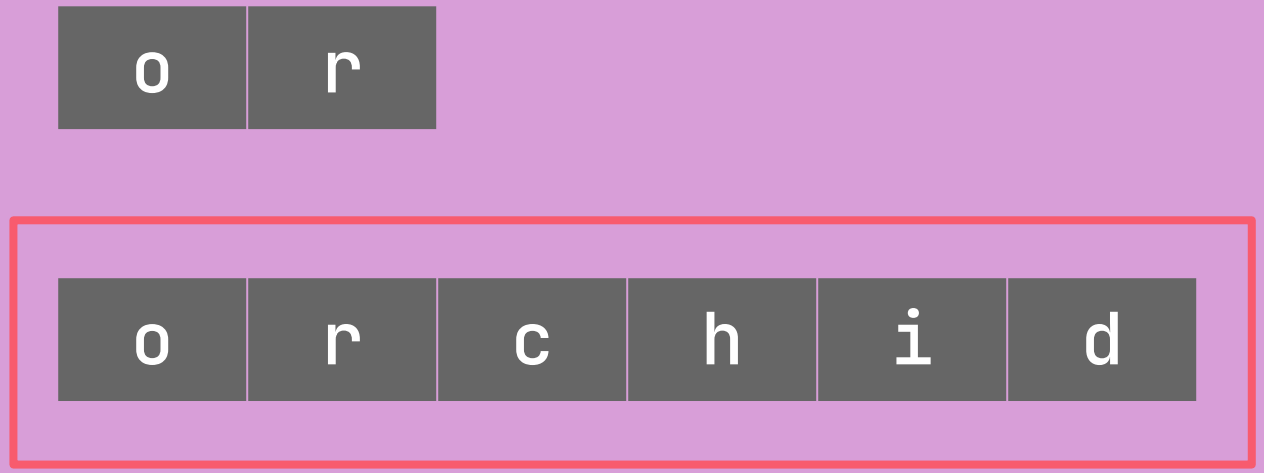
But, we ain’t got time for this nonsense.

Lexical Analysis: Maximal Munch

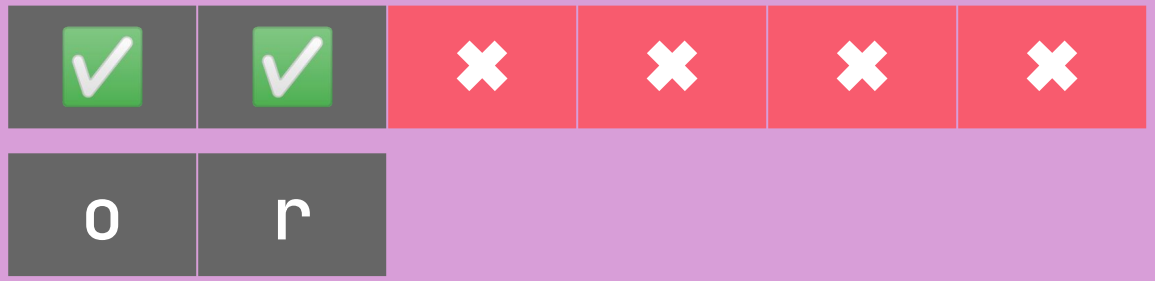
We're going to use *maximal munch*.

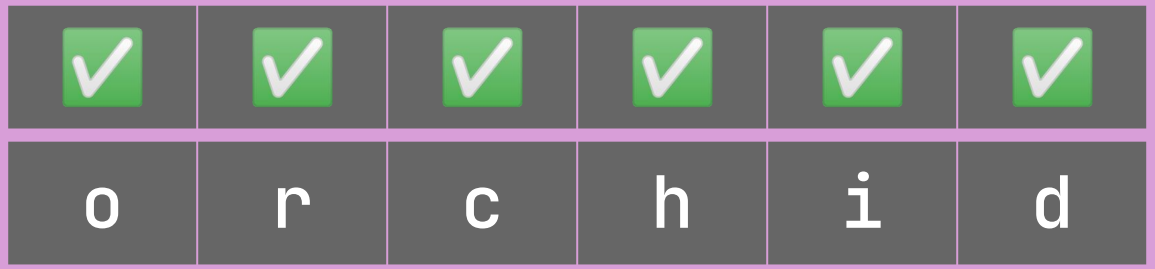
Rule #1 of *maximal munch:*
The most-matched token rule wins.

Lexical Analysis: Maximal Munch



Lexical Analysis: Maximal Munch





Lexical Analysis: Maximal Munch

This works because we reserve important words *first* using a switch statement.

AND	CLASS	ELSE	FALSE	FOR	FUN
IF	NIL	OR	PRINT	RETURN	SUPER
THIS	TRUE	VAR	WHILE		

RESERVED WORDS IN LOX

Lexical Analysis: Maximal Munch

It works the same way for things like multi-character symbols.

=	!=	<	>	>=	<=
==	;	(	)	{	}
/					

RESERVED SYMBOLS IN LOX

Language “regularity”

Consider:

Python

```
def main():
    print(“Hello, World!”)

if __name__ == “__main__”:
    main()
```

Haskell

```
main = putStrLn(hello)
    where
        hello = “Hello, world!”
```

Language “regularity”

Consider:

Haskell

```
main = putStrLn(hello)
    where
        hello = “Hello, world!”
```

Lox

```
def main(){
    print “Hello, World!”;
}

main();
```

Or

```
def main(){print “Hello, World!”;}
main();
```

Lexical “regularity”

Consider:

```
def say():
    def message():
        return “Hello, World!”
    return message()

print(say())
```

Language “regularity”

What can we intuit makes a language “regular”?

Common Lisp

```
 (if nil(print(list 1 2 "foo"))(print(list 3 4 "bar")))
```

Language “regularity”

Lexical regularity (our focus today)

- Concerns the TOKENS of the language
- Interested in the *form*, not the *function*
- Ask: Can I recognize this as valid in n language?

Syntactical regularity (our focus next week)

- Concerns the EXPRESSIONS of the language
- Interested in the *function*
- Ask: Can I recognize this as a correct statement in n language?

Lexical “regularity”

Lexical regularity’s short definition:

> We can model statements with regular expressions.

Question: can’t we model Python and/or Haskell with a regular expression?

Lexical “regularity”

Python

```
def main():
    print(“Hello, World!”)

if __name__ == “__main__”:
    main()
```

```
def\s[a-z0-9_]+\([a-z0-9_,]+\)\:
\t[a-z0-9_]+\((“|’)(.*)(“|’))

…
…
```