CSCI 2320
Principles of Programming Languages

Lexical Analysis
Reading: Chapter 3 of Tucker-Noonan

MOHAMMAD T. IRFAN

Plan

Chomsky's Hierarchy

Lexical Analysis

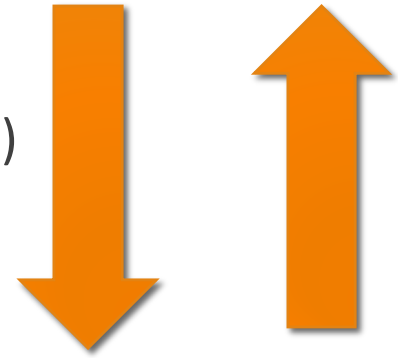
Chomsky's Hierarchy

Regular grammar

Context-free grammar (CFG/BNF)

Context-sensitive grammar

Unrestricted grammar



More expressive power

Faster computation (parsing)

Chomsky Hierarchy

$A, B \in N$
$\tau \in T^*$
$\alpha, \beta \in (T \cup N)^*$

Regular grammar — $A \rightarrow \tau B \mid \tau$

Context-free grammar (CFG/BNF) — $A \rightarrow \beta$

Context-sensitive grammar — $\alpha \rightarrow \beta$, where $|\alpha| <= |\beta|$

Unrestricted grammar — $\alpha \rightarrow \beta$

Regular grammar: pros and cons

$A \rightarrow \tau B \mid \tau$

$A, B \in N$
$\tau \in T^*$

Pros

- Can do the first layer of abstraction in PL syntax
- Very fast parsing

- Example:
  Integer → 0 *Integer* | 1 *Integer* | ... | 9 *Integer* | 0 | 1 | ... | 9

- Following is not a regular grammar (why?)
  - Integer → Integer Digit
  - Digit → 0 | 1 | ... | 9

Cons

- Cannot check balanced parenthesis, braces, etc.
- Cannot represent $\{a^n b^n \mid n >= 1\}$

CFG/BNF/EBNF: pros and cons

$A \rightarrow \beta$

$A \in N$
$\beta \in (T \cup N)^*$

Pros

- Can do all layers of abstractions in PL syntax
- Example: *Assignment* → *Identifier = Expression;*
- Parsing is polynomial time

Cons

- Cannot do lots of semantic-type things
  - Variable declared before use?
  - Operand and operator compatible?
- Cannot represent languages like $\{ww \mid w \in T^+\}$
  - Can do equality checking ($a^n b^n$), but cannot detect repetition

Context-sensitive: pros and cons

$\alpha \rightarrow \beta$, where $|\alpha| <= |\beta|$

$A, B \in N$
$\alpha, \beta \in (T \cup N)^*$

Pros

- Can represent languages like $\{a^n b^n c^n \mid n >= 1\}$

Cons

- Given a context-sensitive grammar, it is undecidable whether any given sentence can be derived from it.
  - Can't do parsing!
  - Can't write a compiler for context-sensitive grammar!

Unrestricted: pros and cons

$\alpha \rightarrow \beta$

$A, B \in N$
$\alpha, \beta \in (T \cup N)^*$

Pros

- Equivalent to the Turing machine
- That is, can compute any computable function

Cons

- Can we do parsing?

Lexical Analysis

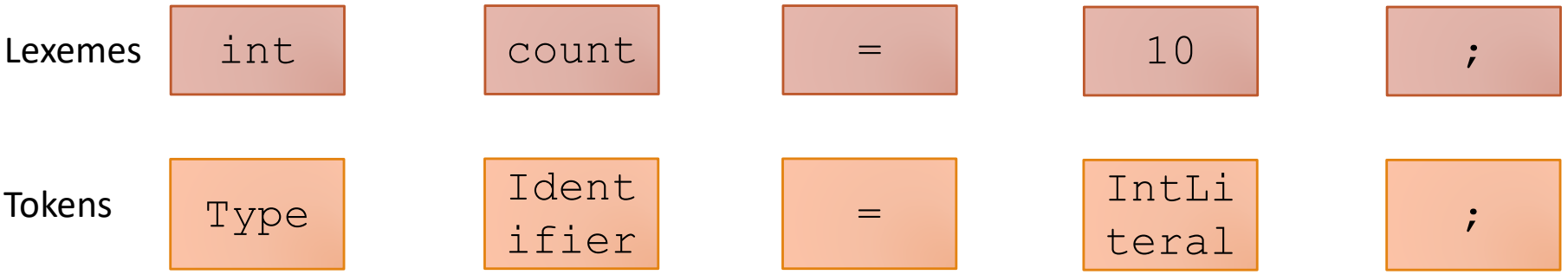
Lexical Analysis

Input: Lexeme(s) = sequence of input characters having a collective meaning

Output: Tokens (representing lexemes)

Discard: whitespace, comments

```
int count = 10;
```



Why do lexical analysis separately?

- 75% of compiling time spent in lexical analysis
- Simpler, faster grammar for parsing
  - Next: how?
- Take care of OS idiosyncrasies

Regular Expressions

Regular Expression (Regex)

Any of the following

1. A character from an alphabet
2. Empty string, $\in$
3. MN where M and N are reg. ex. — Concatenation
4. M|N " " " " " " " — Alteration
5. M^* " " " is " " — *: 0 or more
6. M^+ " " " " " " — +: 1 or more
7. M? " " " " " " — ?: 0 or 1
8. [ ] : Take exactly 1 character

[set of characters]

- Any one character

^ Not

{Z} : Reference to regex Z.

C Lite
Regular Definitions

CLite regular definition

Category	Definition	
AnyChar	[-~]	From space (ASCII 27) to tilde (126)
Letter	[a-zA-Z]	
Digit	[0-9]	
Whitespace	[\t]	Space and tab
Eol	\n	

Category	Definition
Keyword	bool | char | else | false | float | if | int | main | true | while
Identifier	{Letter}({Letter} | {Digit})*
IntegerLit	{Digit}+
FloatLit	{Digit}+\.{Digit}+
CharLit	'{AnyChar}'

Category	Definition
Operator	= | || | && | == | != | < | <= | > | + | - | * | / |! | [|]
Separator	; | . | { | } | (|)
Comment	// ({AnyChar} | {Whitespace})* {Eol}

CFG
vs
Regular grammar
vs
Regex

Implementation Using Python

Python's re package

https://docs.python.org/3/library/re.html

```
import re #regex
re.split(...)#Use regex argument to split a string into parts
```

Common string-matching regex:

Symbol	Definition
\d	[0-9]
\D	[^0-9]
\w	[a-zA-Z0-9_]
\W	[^a-zA-Z0-9_]

Describe the language:

1. $0(0|1)^{+}0$
2. $((\varepsilon|0)1^{*})^{*}$
3. $0^{*}10^{*}10^{*}10^{*}$
4. $(00|11)^{*}$

Write regular expression for:

1. All strings of lowercase letters, where letters appear in ascending order.

2. All strings of letters containing vowels in order.

3. Strings of digits (0, 1, 2) with no consecutively repeated digit.

Finite State Automata (FSA) --- DFA, NFA

BEHIND THE SCENE OF REGULAR EXPRESSIONS

READING:
TEXTBOOK AND HANDOUT (SCOTT)

Known algorithms

1. Regular expression → NFA
2. NFA → DFA

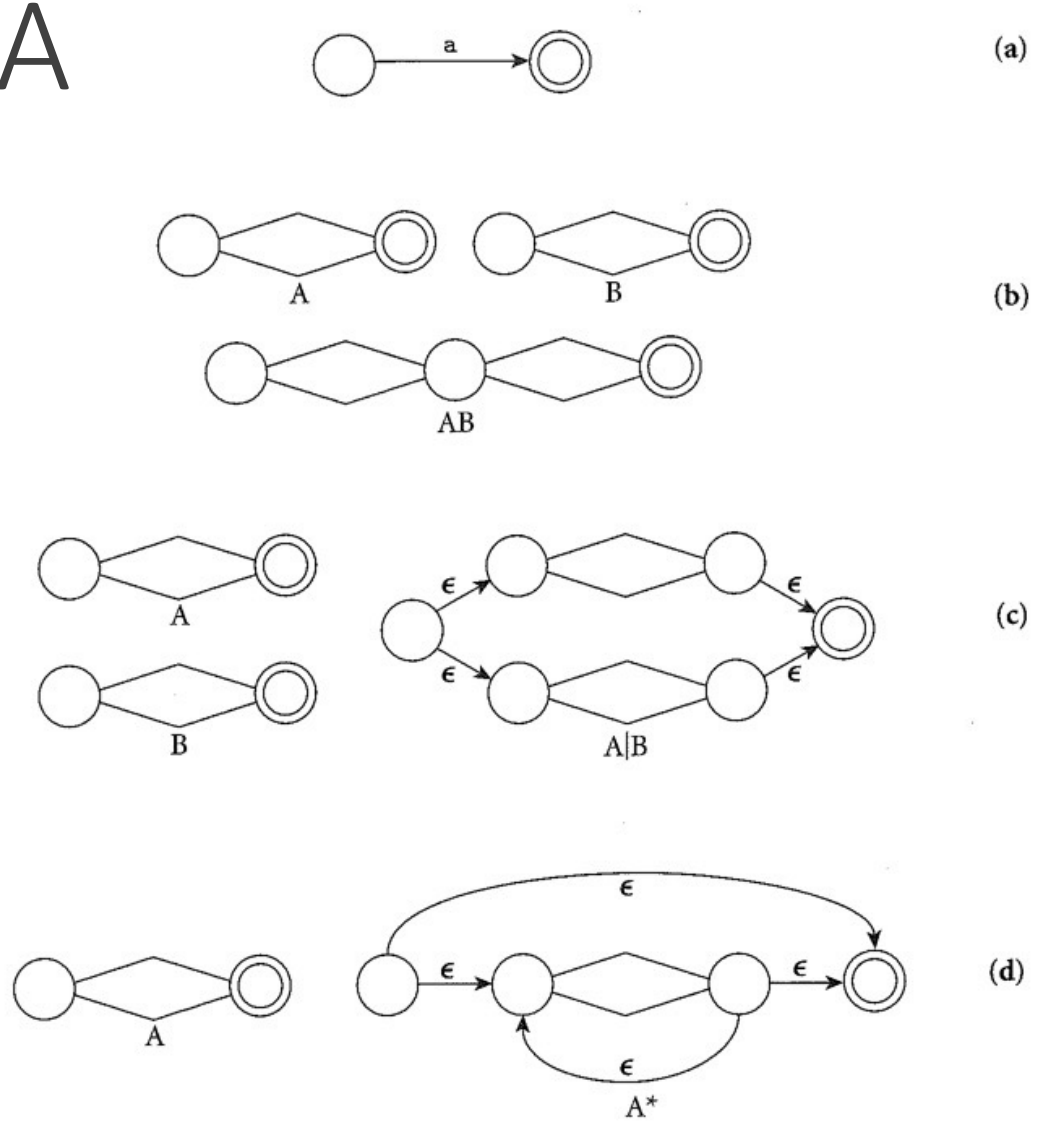
Language designer → implementation (parsing)

3. DFA → regular expression

State elimination

Regex → NFA → DFA → Regex
All 3 are equivalent!

Regex → NFA



Scott, Programming Languages (2000)

Resources

Tucker-Noonan book chapter

Scott's book chapter (handout on Canvas)

Video lecture on NFA -> DFA: https://youtu.be/qfVTXwuxEOY