

## CS421 Lecture 5 - Lexical Analysis

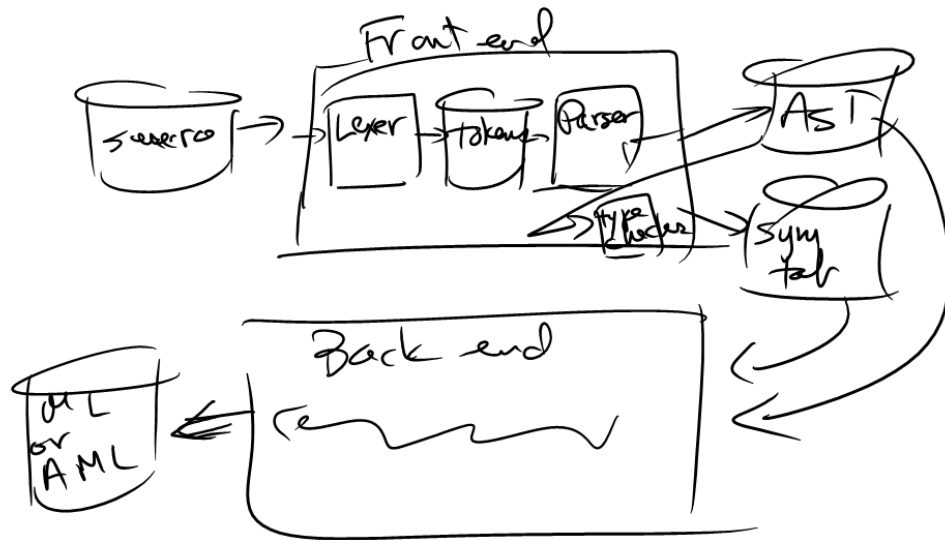
---

- ▶ Today's class
  - ▶ Compiler overview
  - ▶ Lexing
  - ▶ Finite-State Machine as Lexer

## Compiler Outline

---

- ▶ Front-End
  - ▶ Takes Input Source Code
  - ▶ Returns Abstract Syntax Tree + symbol table
- ▶ Back-End
  - ▶ Takes Abstract Syntax Tree + symbol table
  - ▶ Returns machine executable binary code, or virtual machine code, or just interprets program



## Front-end structure

---

- ▶ Lexer (aka scanner, tokenizer)
  - ▶ Transforms program to list of tokens
  - ▶ Produces name table (usually hash table)
- ▶ Parser
  - ▶ Transforms list of tokens to AST
- ▶ Symbol table construction
  - ▶ Fills in name table with information about names in program - type, location, etc.

## Manual and automatic methods

---

- ▶ We will study how to write lexers and parsers. For each, we will give a manual technique and an automatic one:
- ▶ Lexing:
  - ▶ Manual: Finite-state machines
  - ▶ Automatic: Regular expressions - `ocamllex`
- ▶ Parsing
  - ▶ Manual: Top-down (recursive descent) parsing
  - ▶ Automatic: Bottom-up (LR(1)) - `ocamlyacc`

CS 421 - lecture 5 - page 4

## Lexer

---

- ▶ Divide input into “tokens”
- ▶ Tokens are smallest units that are useful for parsing. E.g. parser needs to know if “while” keyword appears; doesn’t need to know that it is made up of characters w, h, etc.
- ▶ Why? Efficiency
  - ▶ Simpler to specify grammatical structure, and implement parser, in terms of tokens

CS 421 - lecture 5 - page 5

# Lexer Input & Output

## ► Lexer Input

### ► Character stream in the form of

- Input Stream, or
- String

## ► Lexer Output

- Stream of tokens, or
- List of tokens

# Tokens

type token =

```
EOF | BOOLEAN | BREAK | CASE | CHAR | CLASS | CONST | CONTINUE  
| DO | DOUBLE | ELSE | EXTENDS | FINAL | FINALLY | FLOAT | FOR  
| DEFAULT | IMPLEMENTS | IMPORT | INT | NEW | IF | PUBLIC  
| SWITCH | RETURN | VOID | STATIC | WHILE | THIS  
| NULL_LITERAL | LPAREN | RPAREN | LBRACE | RBRACE | LBRACK | RBRACK  
| SEMICOLON | COMMA | DOT | EQ | GT | LT | NOT | COMP  
| QUESTION | COLON | EQEQ | LTEQ | GTEQ | NOTEQ | ANDAND | OROR  
| PLUSPLUS | MINUSMINUS | PLUS | MINUS | MULT | DIV | AND  
| OR | XOR | MOD | LSHIFT | RSHIFT | URSHIFT | PLUSEQ | MINUSEQ |  
  MULTEQ  
| DIVEQ | ANDEQ | OREQ | XOREQ | MODEQ | LSHIFTEQ | RSHIFTEQ  
| URSHIFTEQ  
| BOOLEAN_LITERAL of bool  
| INTEGER_LITERAL of int  
| FLOAT_LITERAL of float  
| IDENTIFIER of string - abc7  
| STRING_LITERAL of string - " - - - "
```

## Example

---

- ▶ Input

“class MP1 { public static void main ( .....”

- ▶ Output – list of tokens

[CLASS; IDENTIFIER “MP1”; LBRACE; PUBLIC; STATIC; VOID;  
IDENTIFIER “main”; LPAREN; ..... ]

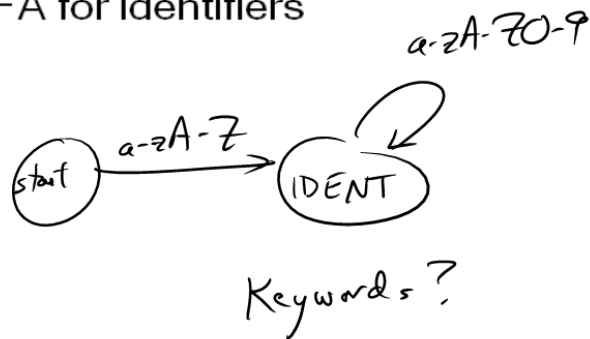
## Lexing with DFA

---

- ▶ Words recognized by finite state automaton
- ▶ Deterministic Finite Automaton (DFA)
  - ▶ A directed graph whose *vertices* are labeled from a set  $\text{Tokens} \cup \{\text{Error}, \text{Discard}\}$  and whose *edges* are labeled with sets of characters. Also, if the outgoing edges from vertex  $v$  are  $\{e_1, \dots, e_n\}$ , then the sets labelling  $e_1, \dots, e_n$  are disjoint. Also, one vertex is specified as the *start* vertex.

## Example 1

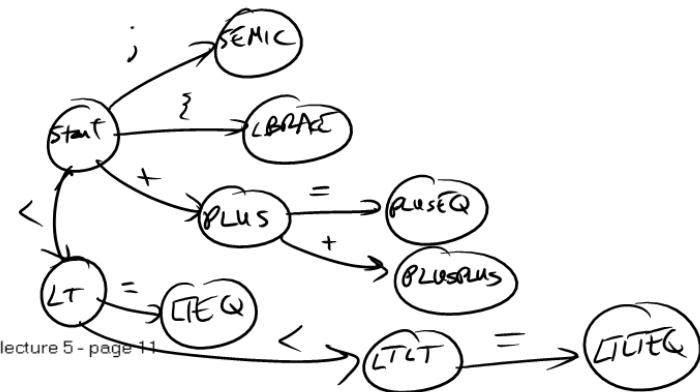
### ► DFA for identifiers



## Example 2

### ► DFA for Operators

; { + += < <= << <=<



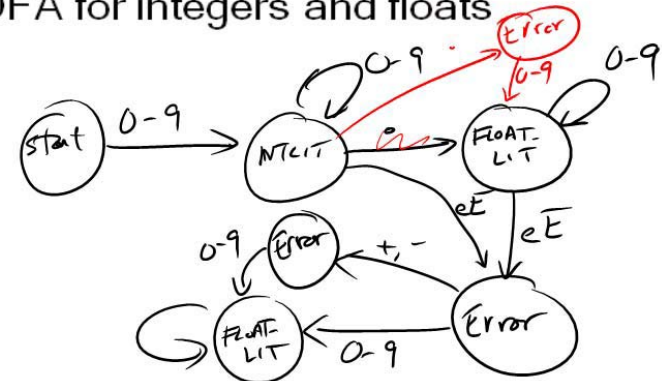
### Example 3

- ▶ DFA for integer constants



Example 4  $\underbrace{-3e7}_{\text{negative}}$   $\underbrace{3.7e+1}_{\text{positive}}$

- ▶ DFA for integers and floats



Whitespace



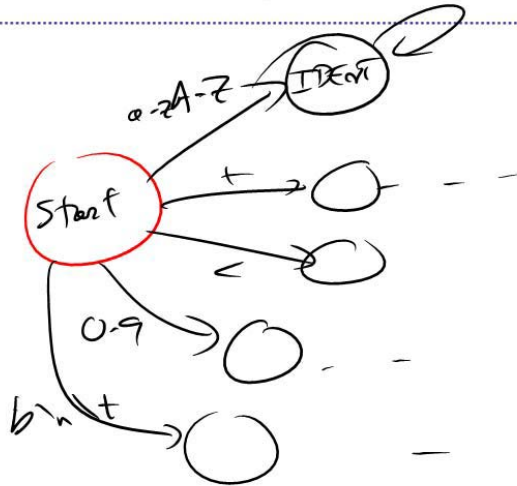
## Completing the DFA

---

- ▶ Need to create a single DFA for all tokens – recall that all outgoing edges must have disjoint label sets.
- ▶ For keyword:
  - ▶ Use DFA for identifiers, but look in table when token is complete to check if it is a keyword.

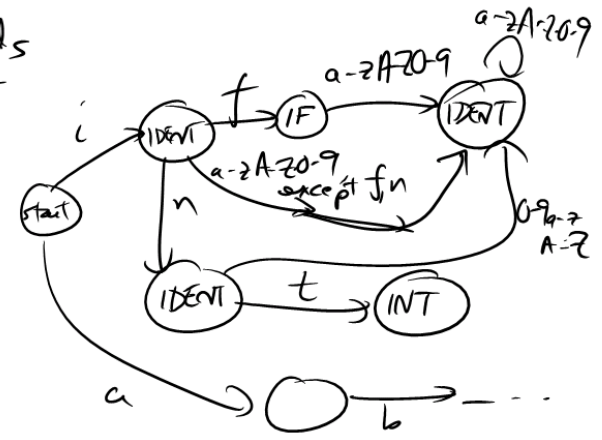


# Completing the DFA



CS 421 - lecture 5 - page 15

Handling keywords  
Using DFA only:



"Cheating":  
 Use IDENT DFA,  
 check for keywords

## Implementing lexing with a DFA

- ▶ Define a transition function. Give each state a number.
  - ▶ transition: state  $\times$  character  $\rightarrow$  state  $\cup$   $\{-1\}$
- ▶ Action
  - ▶ state  $\rightarrow$  token  $\cup$  {discard, error}
- ▶ Assume start state = 0

## Implementing lexing with a DFA

Function to get a single token:

```
(state × string) getNextToken() {  
    s = 0; tokenchars = "";  
    while (true) {  
        c = peek at next char  
        if (move(s,c) == -1)  
            return (s, tokenchars)  
        move c from input to tokenchars  
        s = move(s,c)  
    }  
}
```

S, C) Declare transition function:

```
int move[numstate][numchar] = {
```

|   |   |   |
|---|---|---|
|   | { | } |
| { | { | } |
| { | { | } |

## Implementing lexing with a DFA

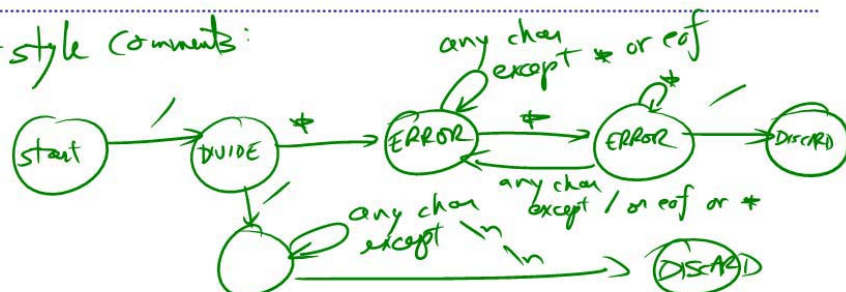
```
token list gettokens() {  
    tokenlis = []  
    while (true) {  
        c = peek at next char  
        if (c == eofchar) {  
            tokenlis = tokenlis @ [EOF]  
            break  
        }  
        (s, tokenchars) = getnexttoken()  
        perform action of state s (which may look at tokenchars)  
    }  
    return tokenlis  
}
```

## Typical lexer actions

- ▶ Recall that a state's action is a token, "error", or "discard":
  - ▶ Error: Represents an erroneous input; abort.
  - ▶ LTLT: - Put in token list
  - ▶ IDENT: - Put in token list, put identifier in "name table"  
Check for keyword, emit kw
  - ▶ ~~COMMENT~~ DISCARD: do nothing

## More DFAs

C-style comments:



## More DFAs

OCaml-style comment:

(+ -- (\* --- +) -- \*)

Impossible: Pumping lemma

