CS 421 Lecture 5: Lexical analysis

- Lecture outline
  - Lexical analysis (lexing)
  - Deterministic Finite Automaton as Lexer

Compiler outline

- Front-end
  - Takes input source code
  - Returns Abstract Syntax Tree and symbol table
- Back-end
  - Takes AST and symbol table
  - Returns machine-executable binary code, or virtual machine code, or just interprets the program

Front-end structure

- Lexer (a.k.a. scanner, tokenizer)
  - Transforms program into a 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: Deterministic Finite Automata (DFA)
  - Automatic: Regular expressions – ocamllex
- Parsing
  - Manual: Top-down (recursive descent) parsing
  - Automatic: Bottom-up (LR(1)) – ocamlyacc

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's made up of characters 'w', 'h', *etc.*
- Why? Efficiency
  - Simpler to specify grammatical structure, and implement a parser, in terms of tokens

Lexer input and 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
  | STRING_LITERAL of string
```

Example

- Input

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

- Output – list of tokens

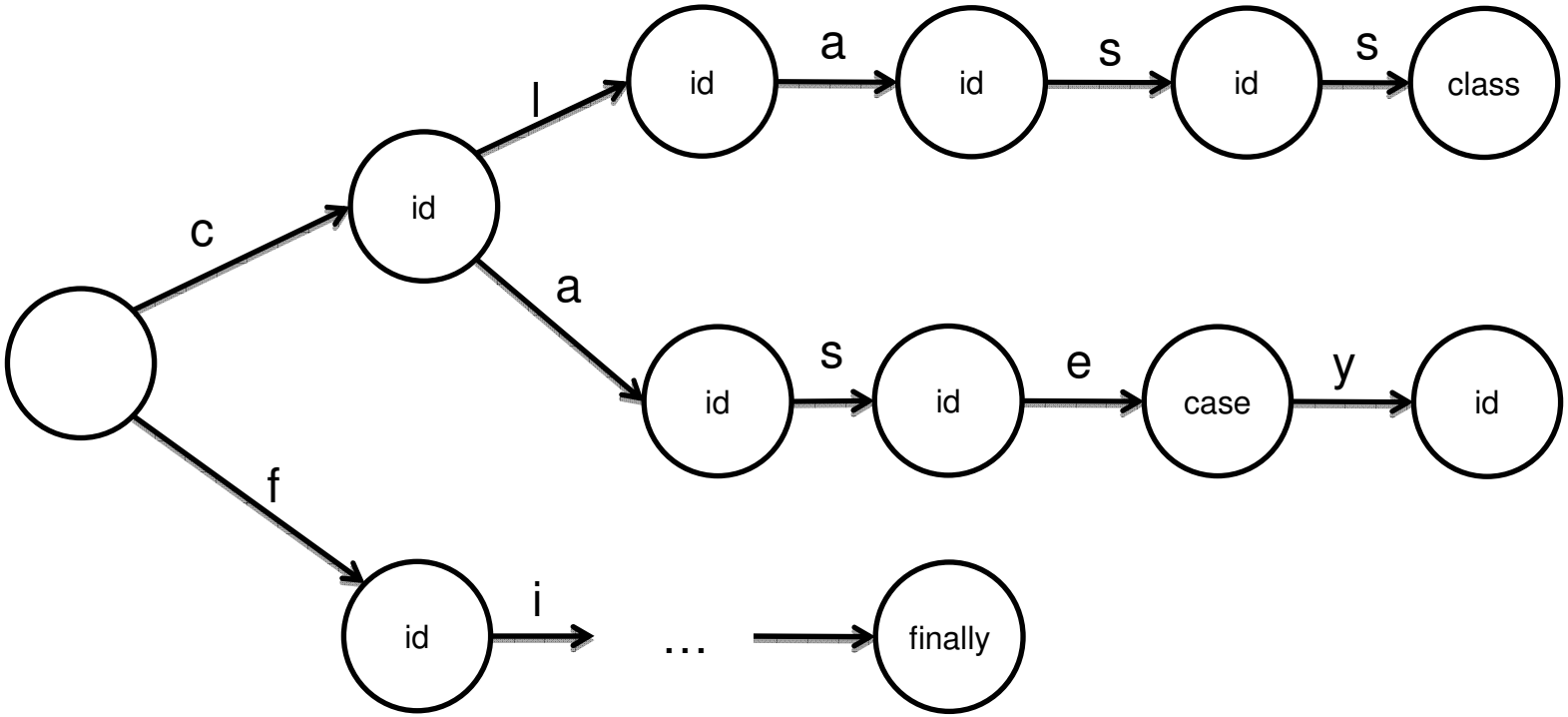
  [CLASS; IDENTIFIER “MP1”; LBRACE; PUBLIC; STATIC;
  VOID; IDENTIFIER “main”; LPAREN; … …]

Lexing with DFA

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

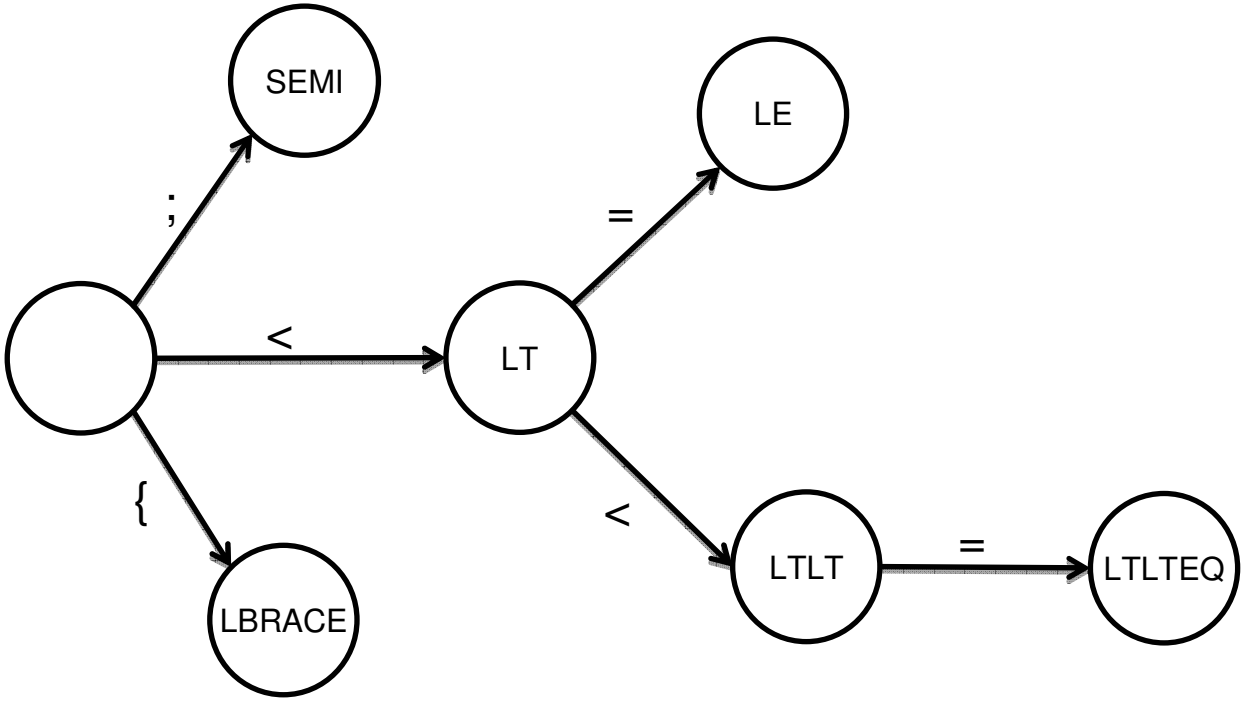
Example 1

- DFA for keywords
  - class case finally



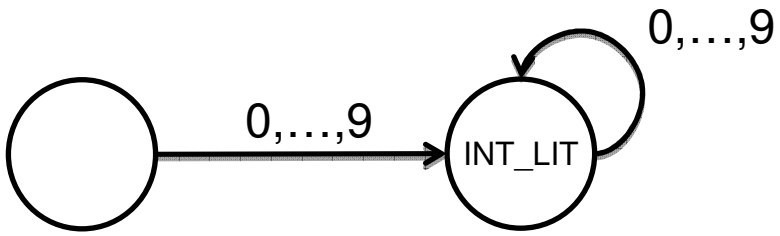
Example 2

- DFA for operators
  - ; { + += < <= << <<=



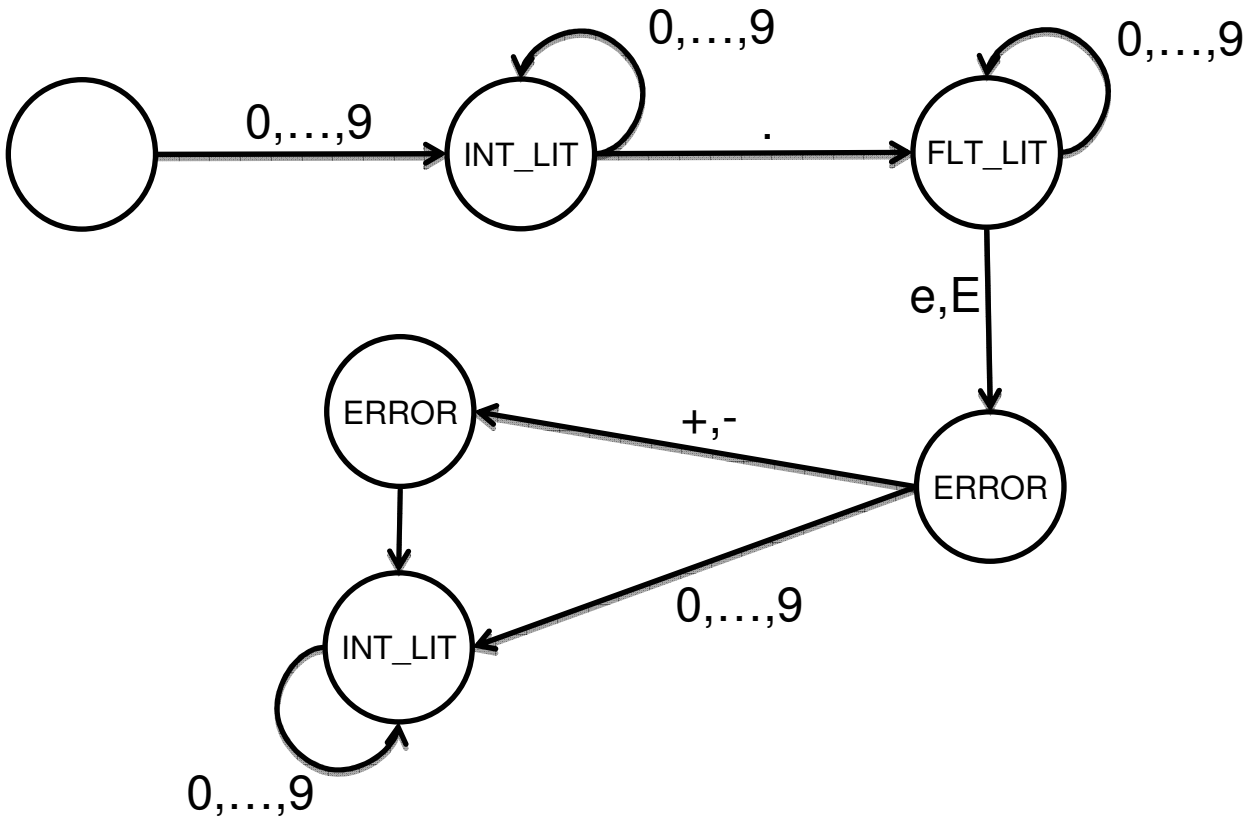
Example 3

- DFA for integer constants



Example 4

- DFA for integers and floats



Using DFA for lexing

- Need one DFA for all tokens
- Start at start state, follow DFA as long as possible
- Then, depending on label of final state:
  - Token: emit token
  - Error: abort
  - Discard: do nothing
- Repeat until input is consumed

Completing the DFA

- Need to create a single DFA for all tokens – recall that all outgoing edges must have disjoint label sets.
- For keywords and identifiers:
  - Instead of creating the DFA shown earlier, create a small DFA, and use action to distinguish keywords

Implementing lexing with a DFA

- Define a transition function. Give each state a number.
  - transition : state x character -> state U {-1}
- Label function
  - state -> token U {discard, error}
- Assume start state = 0

Implementing lexing with a DFA

- Function to get a single token:

```
(state x 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)
 }
```

Implementing lexing with a DFA

- Function to get a all tokens:

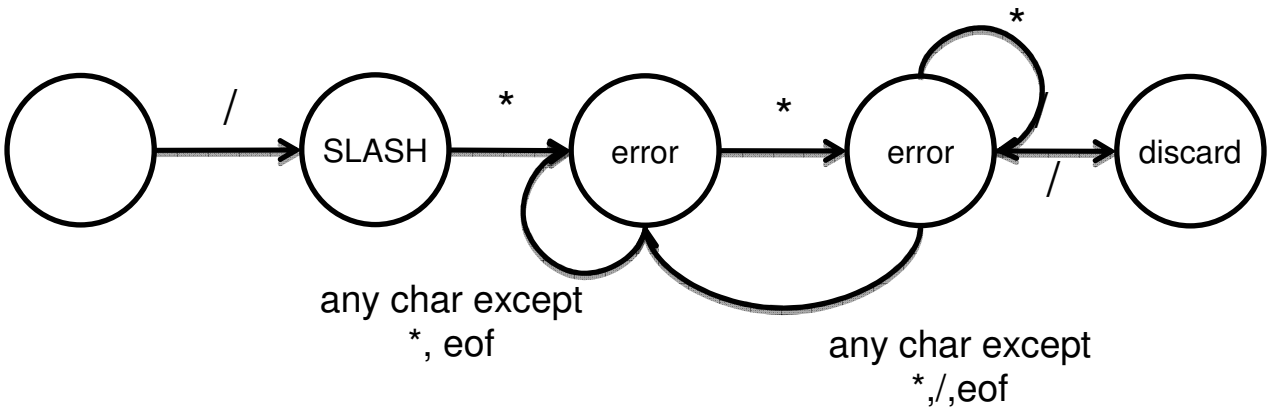
```
token list gettokens() {
    tokenlis = []
    while (true) {
       c = peek at next char
       if (c == eofchar) {
          tokenlis = tokenlis @ [EOF]
          break
       }
       (s, tokenchars) = getnexttoken()
        perform action based on s and tokenchars
     }
     return tokenlis
 }
```

Typical lexer actions

- Recall that a state's label is a token or error. Action depends on that label, *e.g.*:
  - Error: represents erroneous input; abort.
  - LTLT:
  - IDENT:
  - COMMENT:

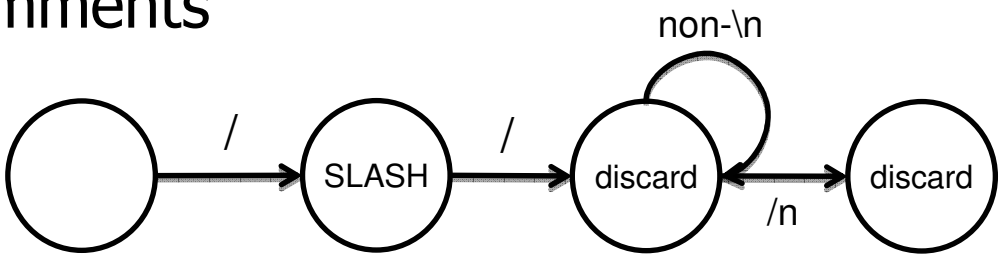
More DFAs

- C-style comments

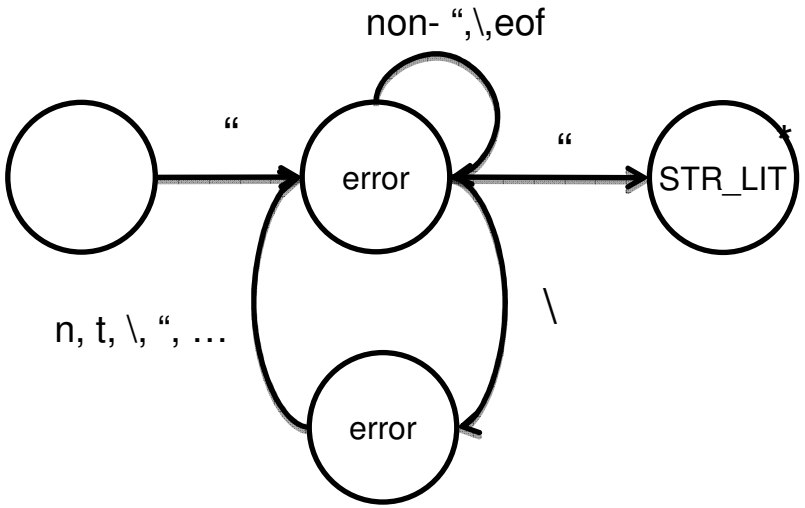


More DFAs

- C++ comments

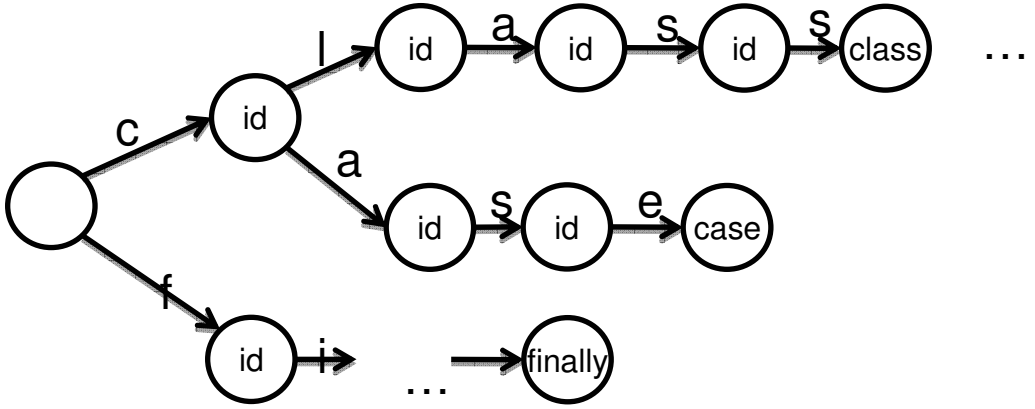


- Strings

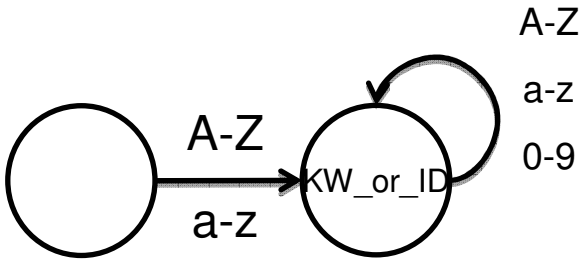


Example 1

- DFA for keywords and identifiers
- Bad way:



- Good way:



Action: see if chars are keyword