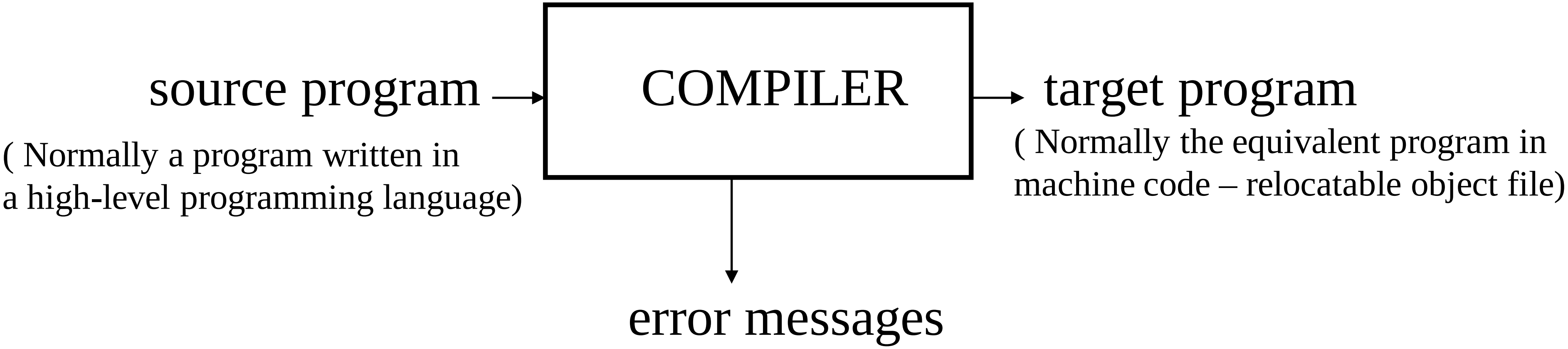
Course Outline

- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation

COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



source program

(Normally a program written in
a high-level programming language)

COMPILER

target program

(Normally the equivalent program in
machine code – relocatable object file)

error messages

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Phases of A Compiler

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

- Each phase transforms the source program from one representation into another representation.

- They communicate with error handlers.

- They communicate with the symbol table.

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

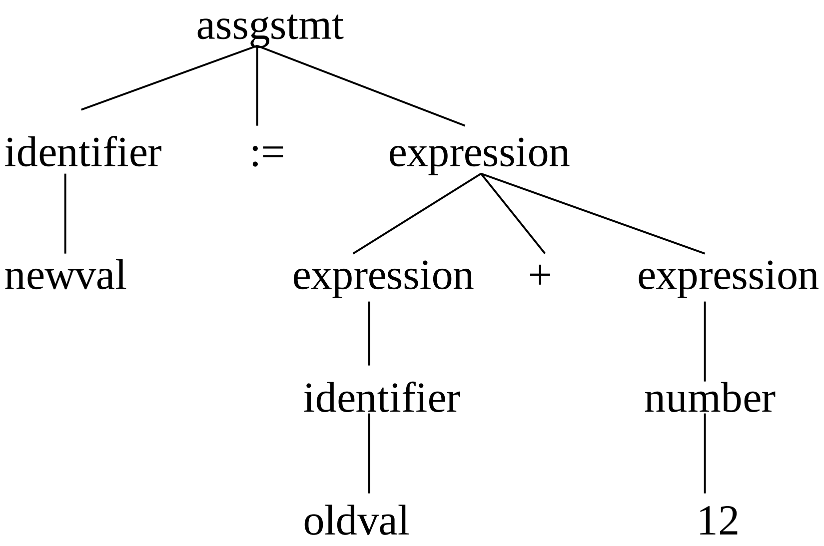
Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

assgstmt -> identifier := expression
expression -> identifier
expression -> number
expression -> expression + expression

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  newval := oldval + 12

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
- Ex:

newval := oldval * fact + 1

↓

id1 := id2 * id3 + 1

↓

```
MULT    id2,id3,temp1                Intermediates Codes (Quadraples)
ADD     temp1,#1,temp2
MOV     temp2,,id1
```

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT    id2,id3,temp1
ADD     temp1,#1,id1
```

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

- Ex:

(assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE      id2,R1
MULT      id3,R1
ADD       #1,R1
MOVE      R1,id1
```

Compiler course

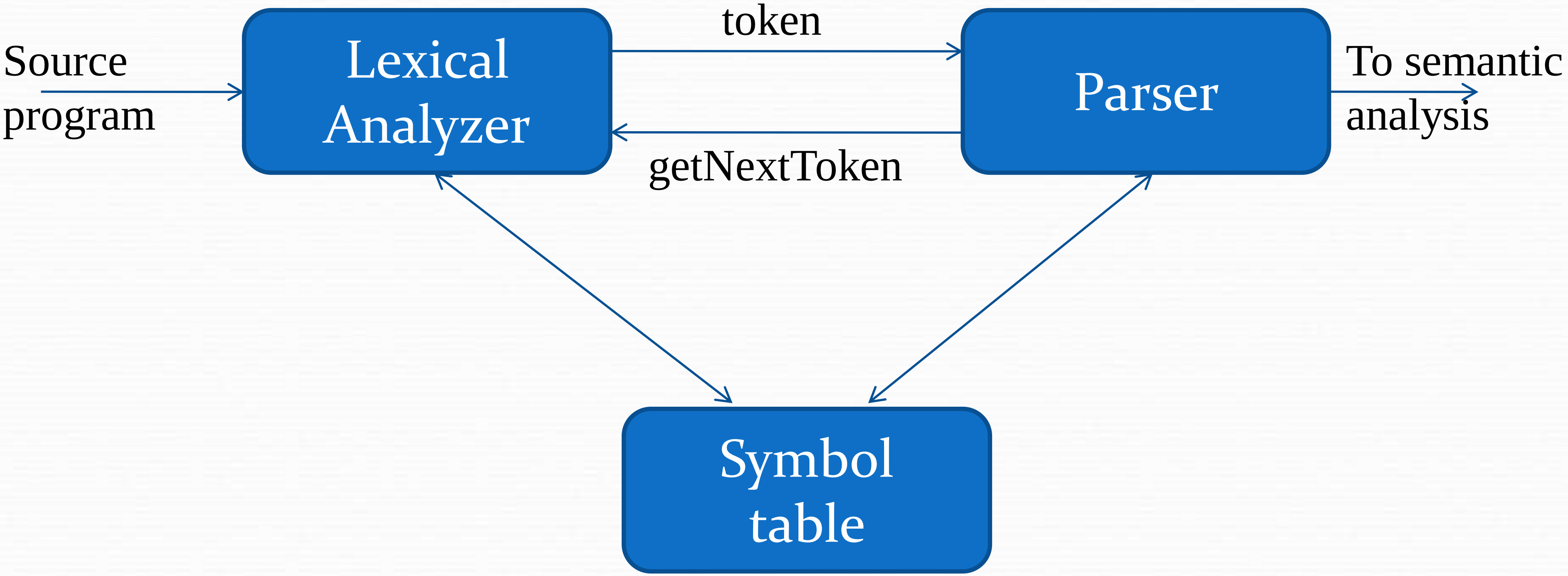
Chapter 3

Lexical Analysis

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

The role of lexical analyzer



Source program

Lexical Analyzer

token

getNextToken

Parser

To semantic analysis

Symbol table

Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ sorrounded by “	“core dumped”

printf(“total = %d\n”, score);

Attributes for tokens

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

Lexical errors

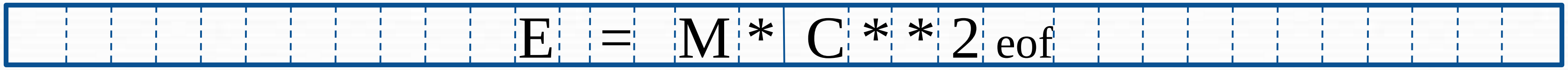
- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely



Sentinels

```
Switch (*forward++) {
    case eof:
        if (forward is at end of first buffer) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if {forward is at end of second buffer) {
            reload first buffer;\
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    cases for the other characters;
}
```

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L9r))*
- (r) is a regular expression denting L(r)

Regular definitions

d1 -> r1

d2 -> r2

…

dn -> rn

- Example:

  letter_ -> A | B | … | Z | a | b | … | Z | _
  digit -> 0 | 1 | … | 9
  id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter|digit)*

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ε
expr -> term **relop** term
| term
term -> **id**
| **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

  digit -> [0-9]

  Digits -> digit+

  number -> digit(.digits)? (E[+-]? Digit)?

  letter -> [A-Za-z_]

  id -> letter (letter|digit)*

  If -> if

  Then -> then

  Else -> else

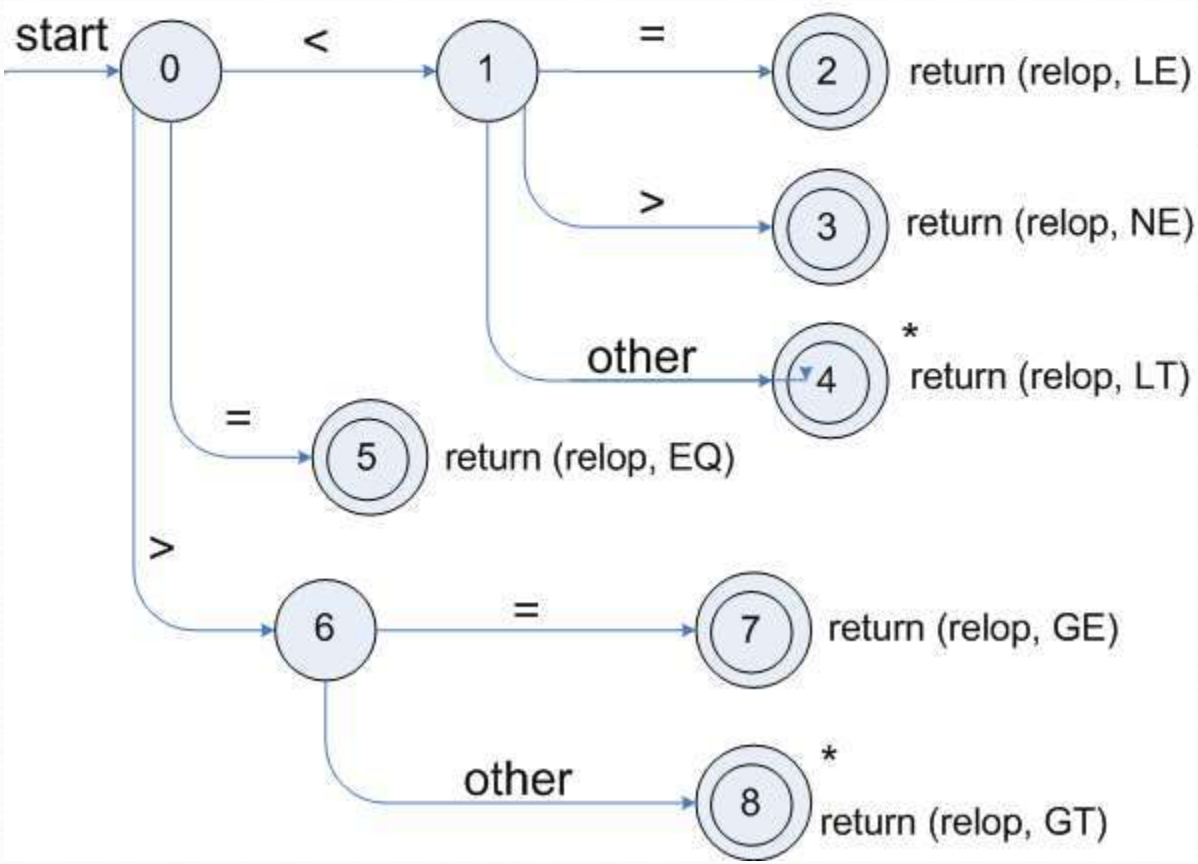
  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

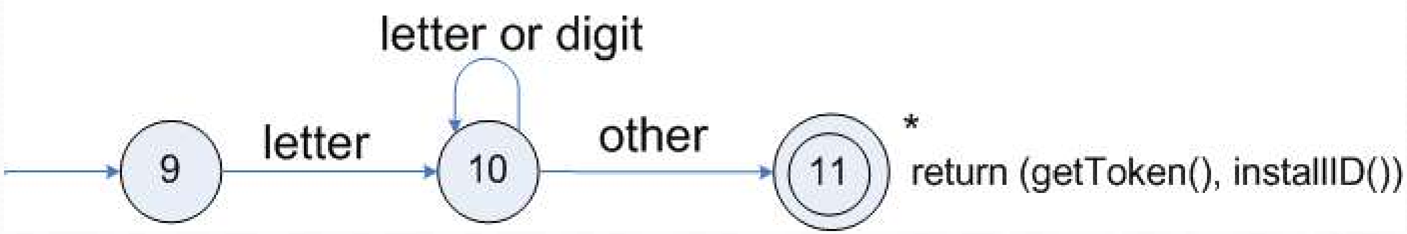
Transition diagrams

- Transition diagram for relop



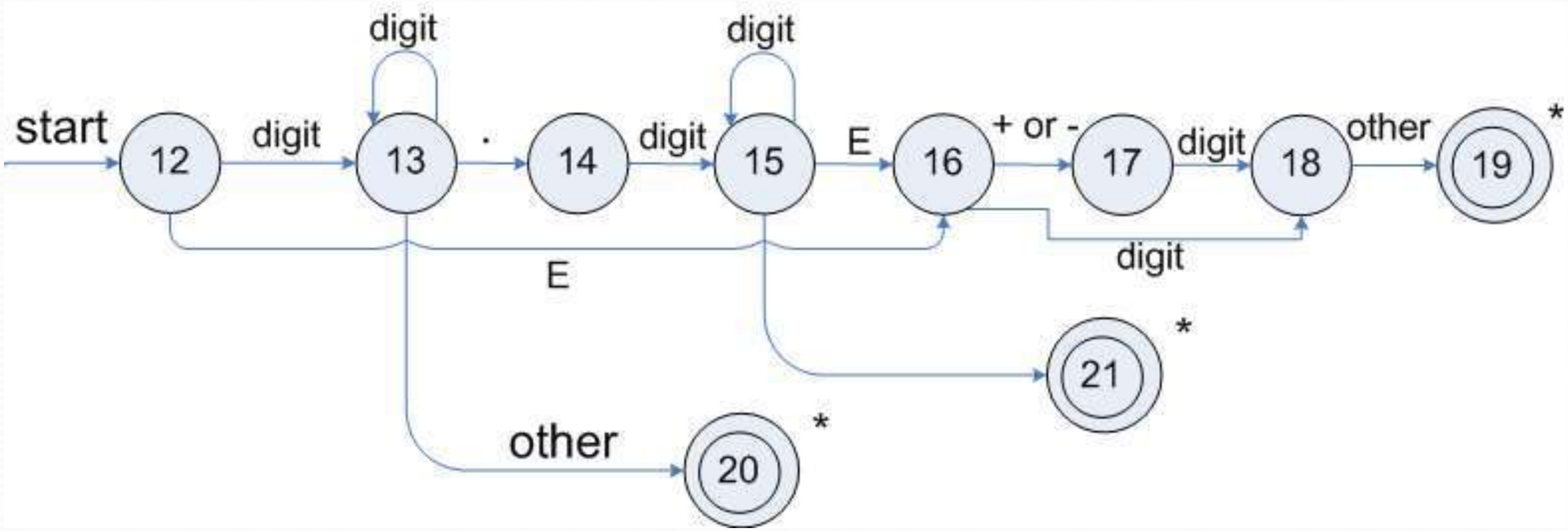
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



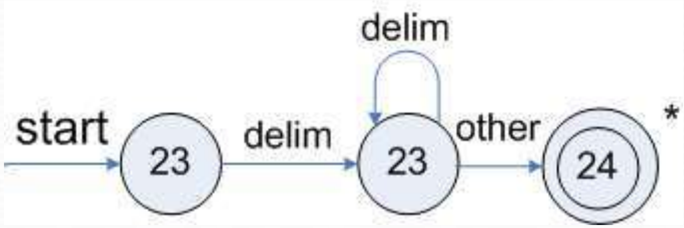
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

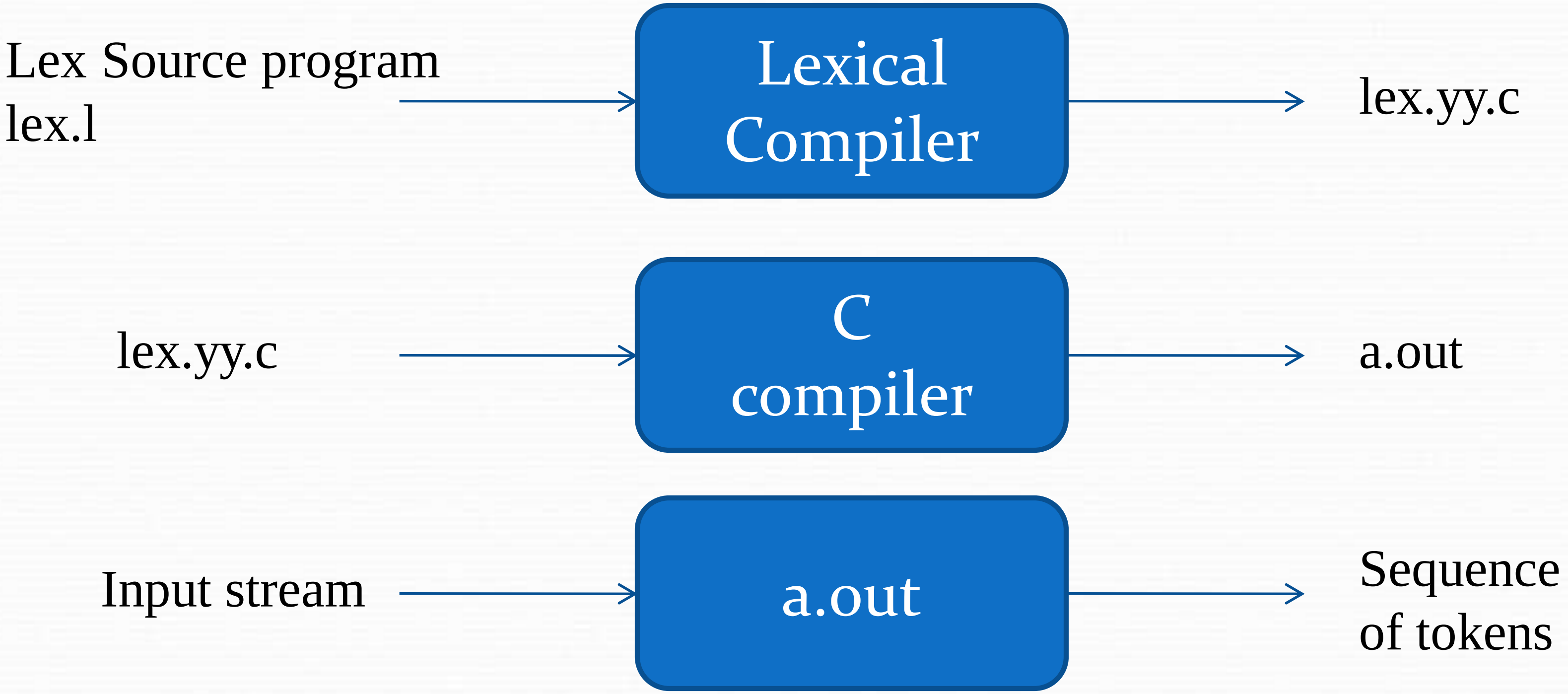
- Transition diagram for whitespace



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) {            /* repeat character processing until a
                                   return or failure occurs  */
    switch(state) {
             case 0: c= nextchar();
                           if (c == ‘<‘) state = 1;
                           else if (c == ‘=‘) state = 5;
                           else if (c == ‘>’) state = 6;
                           else fail();  /* lexeme is not a relop */
                           break;
             case 1: …
             …
             case 8: retract();
                          retToken.attribute = GT;
                          return(retToken);
    }
```

Lexical Analyzer Generator - Lex



Lex Source program
lex.l

Lexical Compiler

lex.yy.c

lex.yy.c

C compiler

a.out

Input stream

a.out

Sequence of tokens

Structure of Lex programs

declarations
%%
translation rules → Pattern {Action}
%%
auxiliary functions

Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
...
```

```
Int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \rightarrow^{a} s_2$$

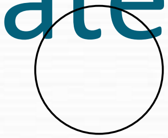
- Is read

In state s_1 on input “a” go to state s_2

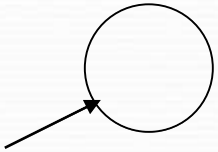
- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

Finite Automata State Graphs

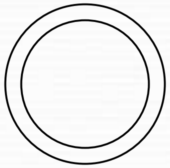
- A state



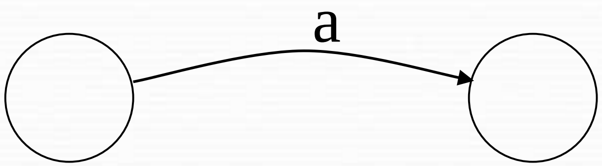
- The start state



- An accepting state

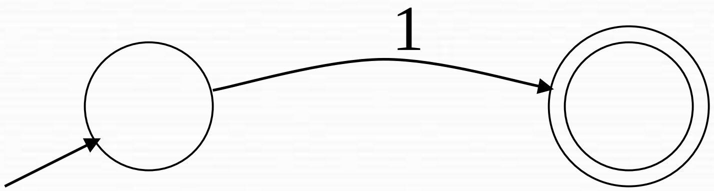


- A transition



A Simple Example

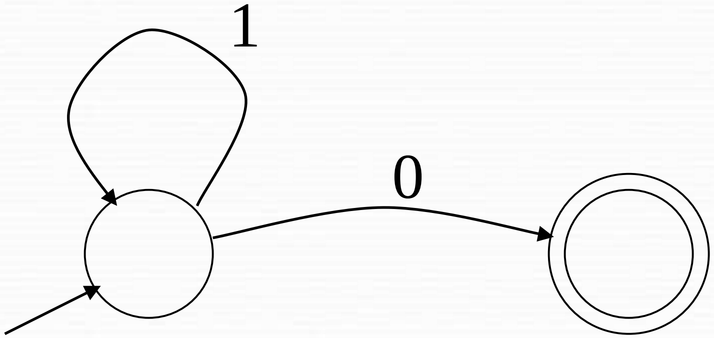
- A finite automaton that accepts only “1”



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

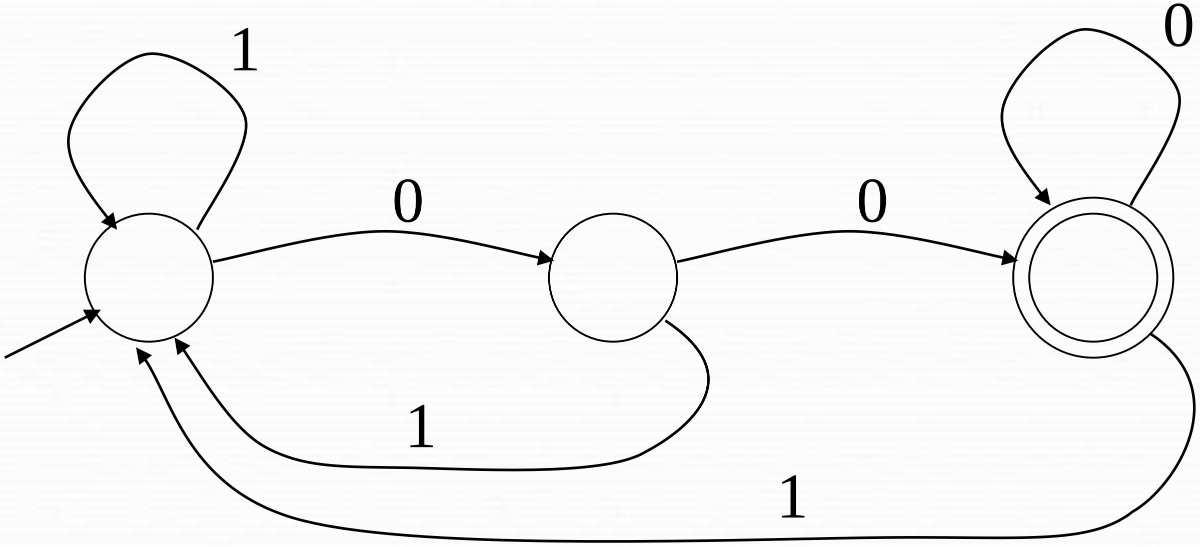
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted but "110…" is not

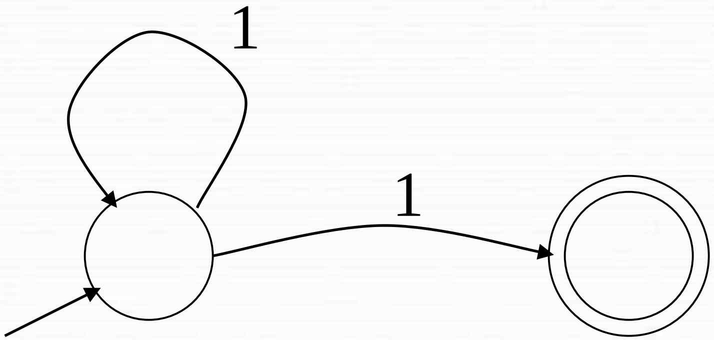
And Another Example

- Alphabet {0,1}
- What language does this recognize?



And Another Example

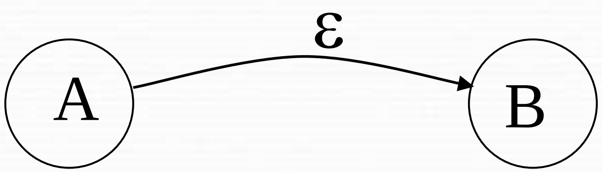
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

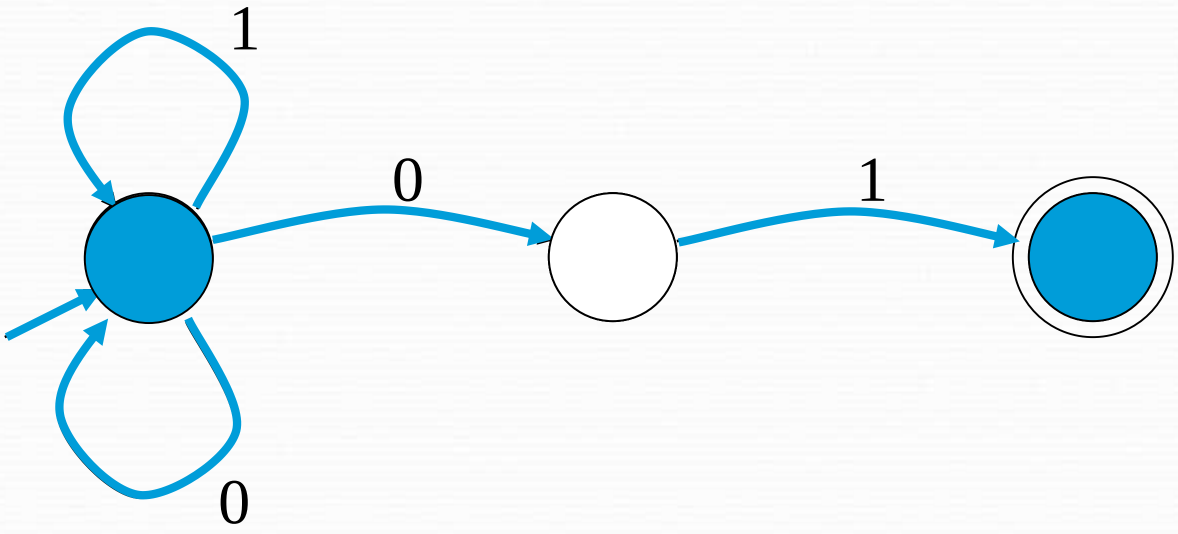
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it can get in a final state

NFA vs. DFA (1)

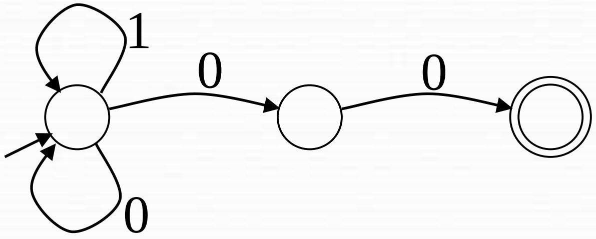
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider

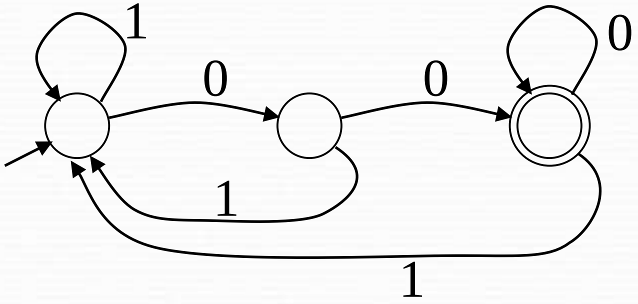
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



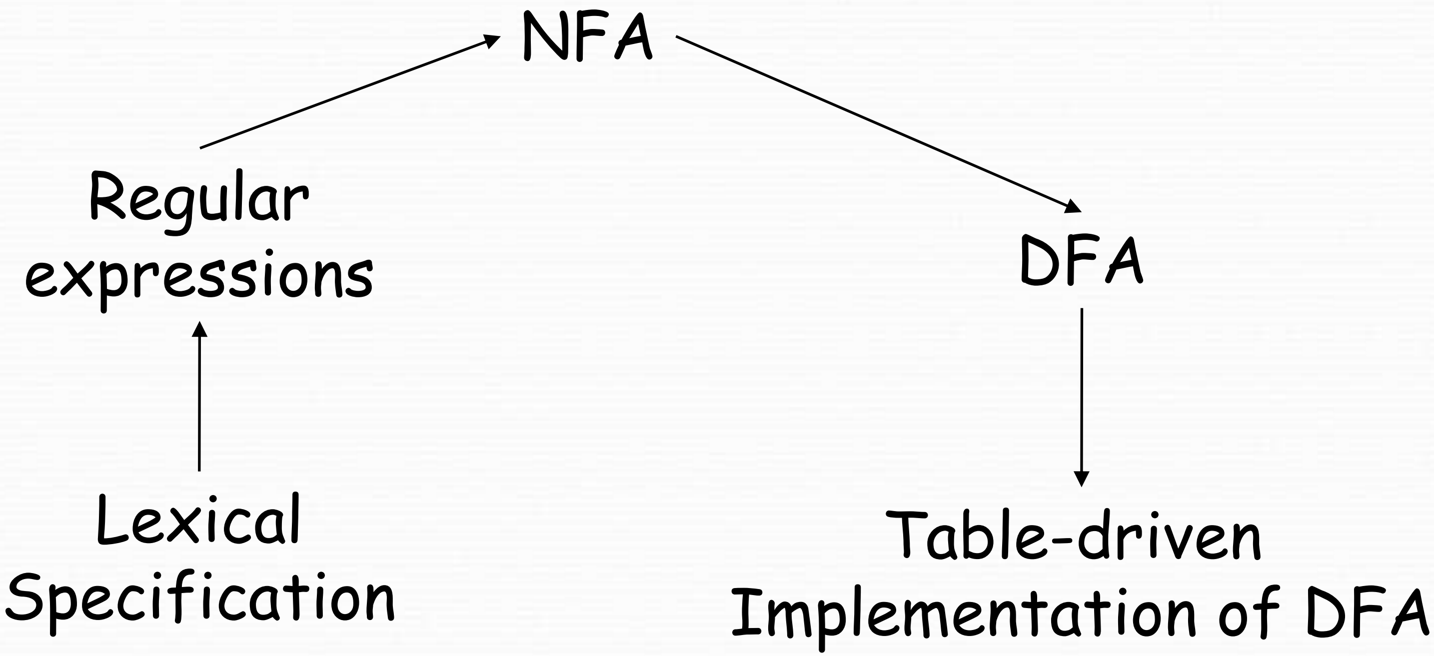
DFA



- DFA can be exponentially larger than NFA

Regular Expressions to Finite Automata

- High-level sketch



NFA

Regular expressions

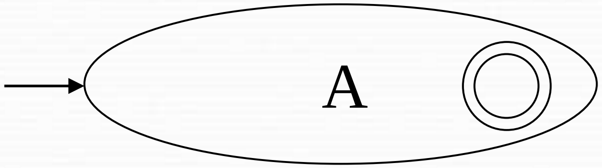
DFA

Lexical Specification

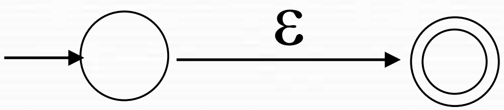
Table-driven Implementation of DFA

Regular Expressions to NFA (1)

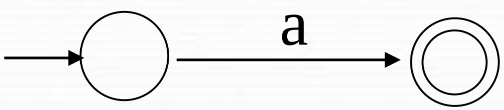
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε

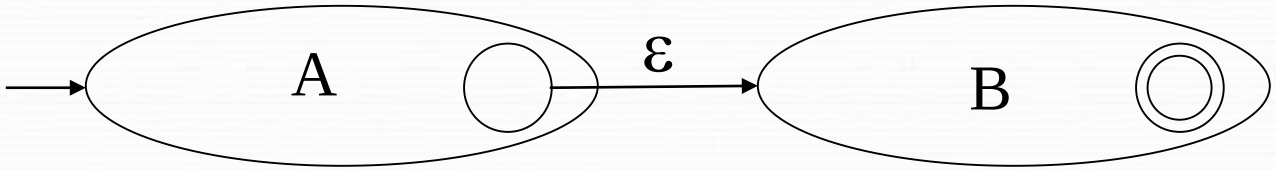


- For input a

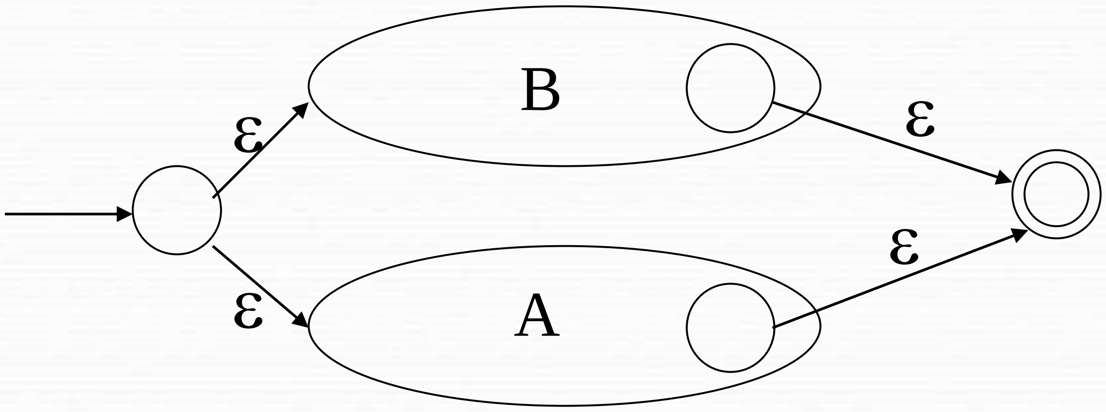


Regular Expressions to NFA (2)

- For AB

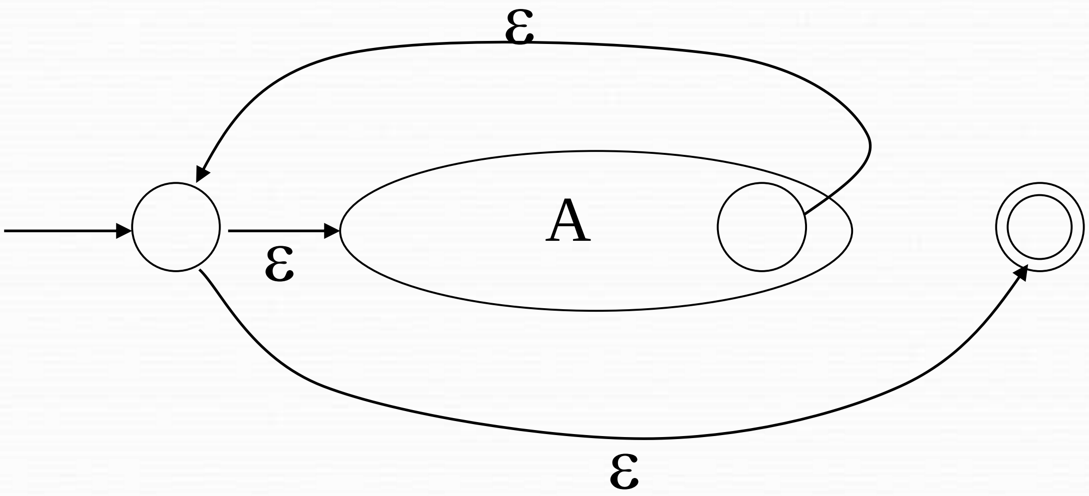


- **For A | B**



Regular Expressions to NFA (3)

- For A*

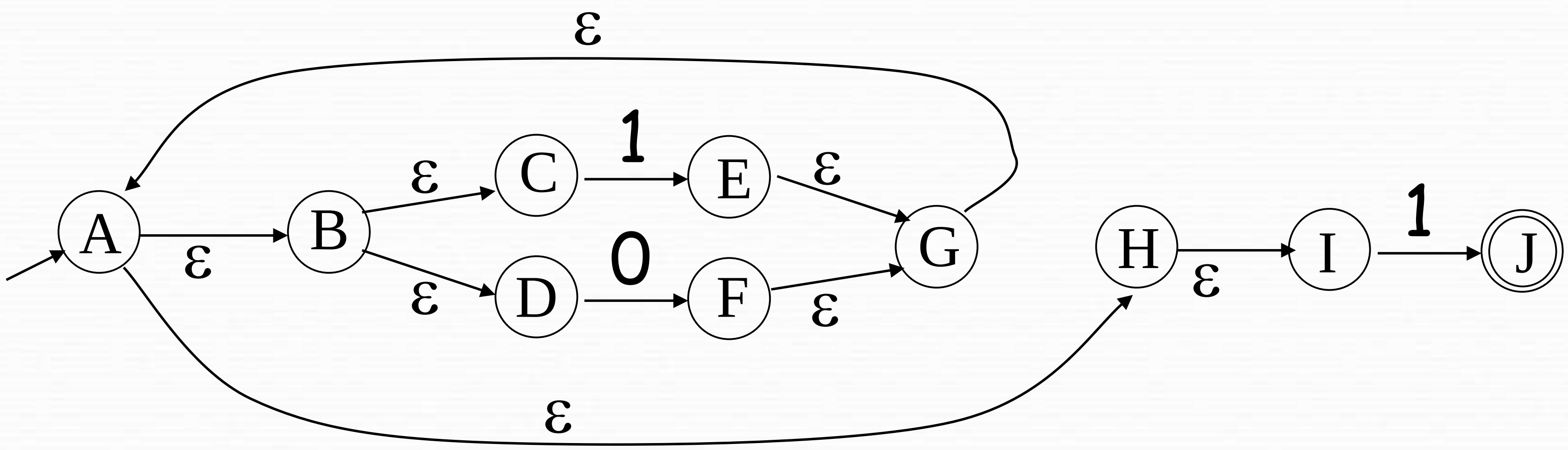


Example of RegExp -> NFA conversion

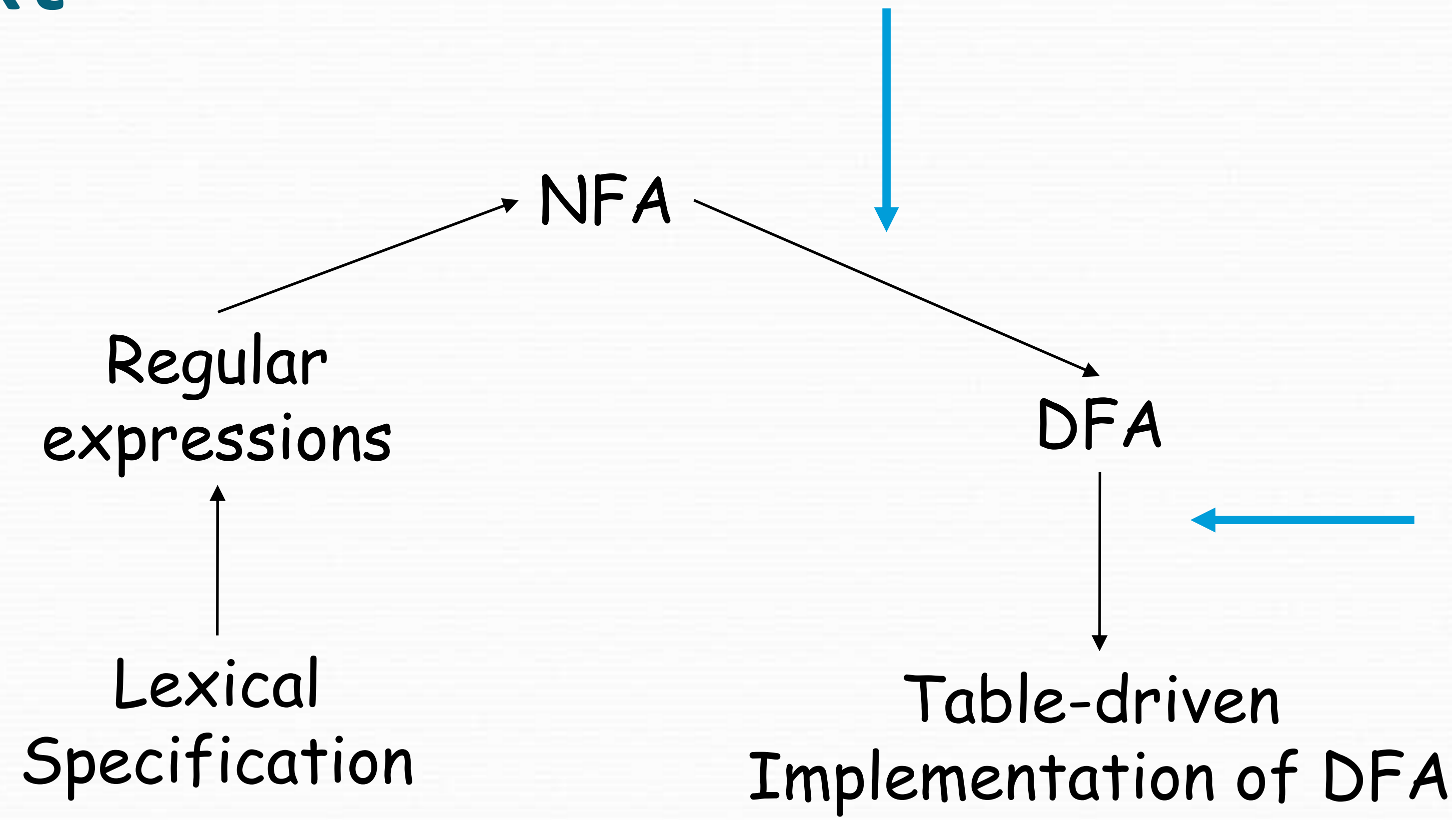
- Consider the regular expression

  $(1 \mid 0)^*1$

- The NFA is



Next



NFA

Regular expressions

DFA

Lexical Specification

Table-driven Implementation of DFA

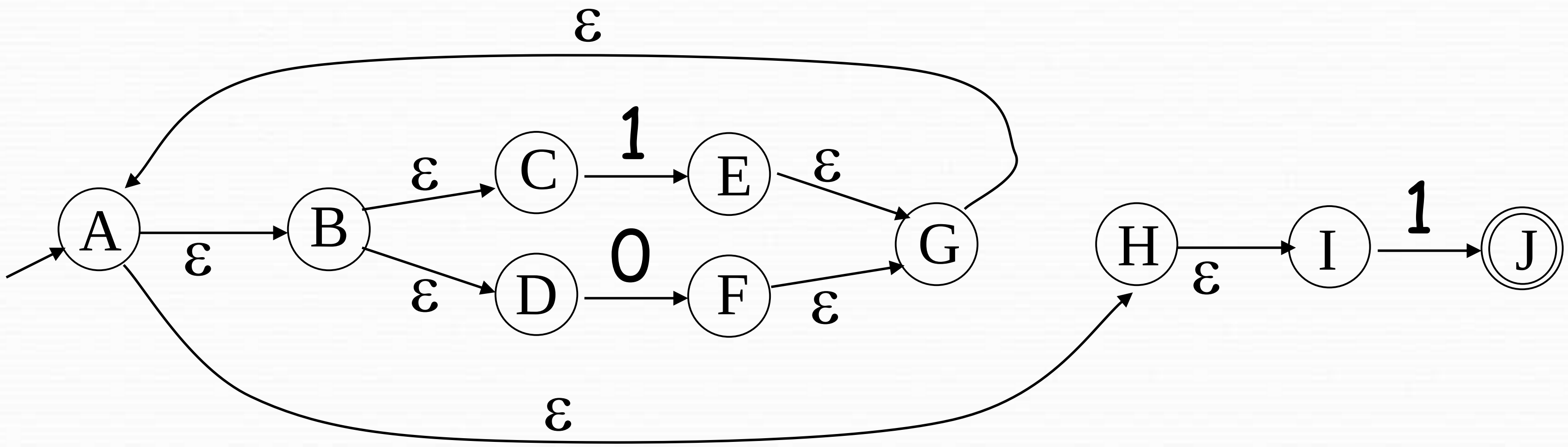
NFA to DFA. The Trick

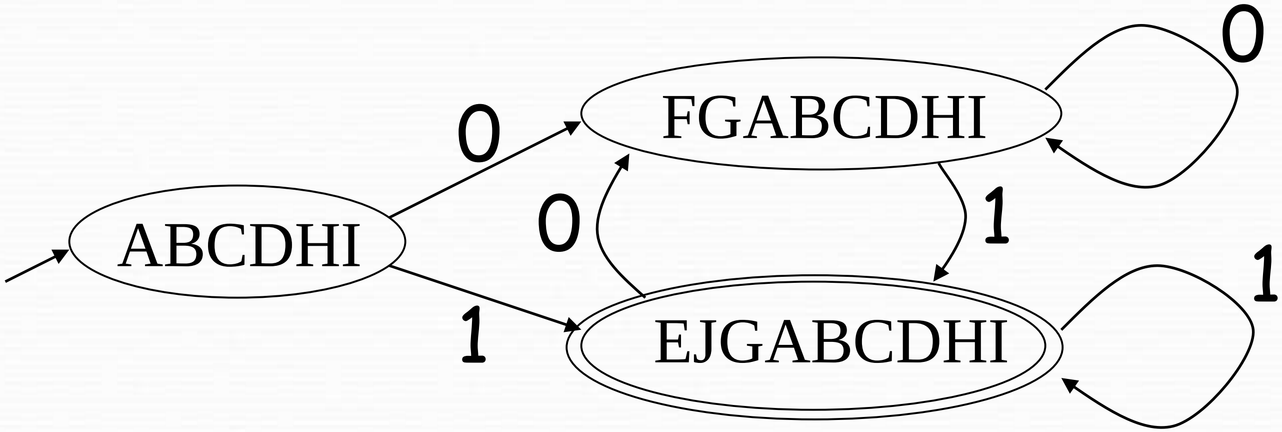
- Simulate the NFA
- Each state of resulting DFA

  = a non-empty subset of states of the NFA
- Start state

  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \to^{a} S'$ to DFA iff
  - S’ is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

NFA -> DFA Example





NFA to DFA. Remark

- An NFA may be in many states at any time

- How many different states ?

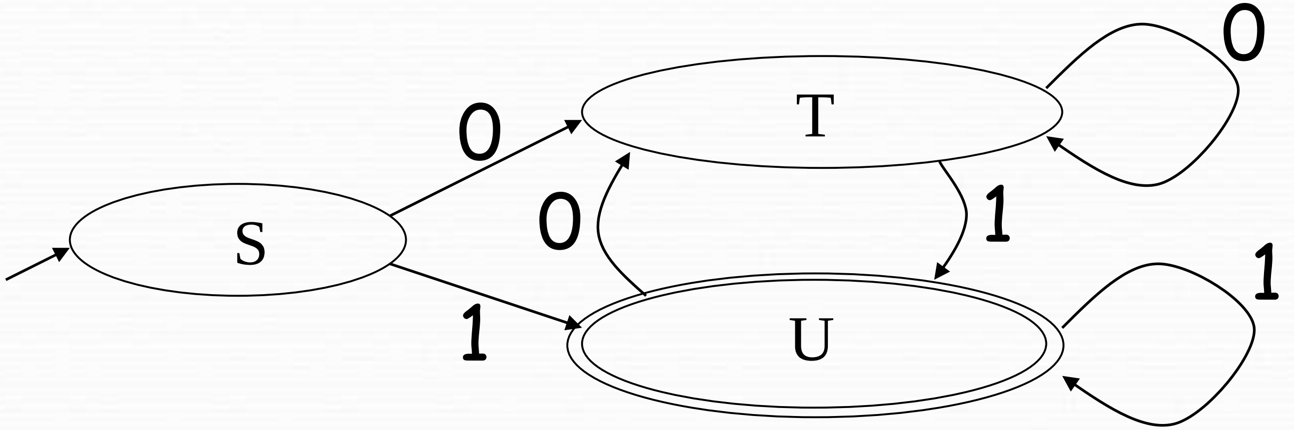
- If there are N states, the NFA must be in some subset of those N states

- How many non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many

Implementation

- A DFA can be implemented by a 2D table T
    - One dimension is "states"
    - Other dimension is "input symbols"
    - For every transition $S_i \rightarrow^{a} S_k$ define T[i,a] = k
- DFA "execution"
    - If in state S_i and input a, read T[i,a] = k and skip to state S_k
    - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex or jflex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

Readings

- Chapter 3 of the book

Context Free Grammars

- One or more non terminal symbols
  - Lexically distinguished, e.g. upper case
- Terminal symbols are actual characters in the language
  - Or they can be tokens in practice
- One non-terminal is the distinguished start symbol.

Grammar Rules

- Non-terminal ::= sequence
  - Where sequence can be non-terminals or terminals
- At least some rules must have ONLY terminals on the right side

Example of Grammar

- S ::= (S)
- S ::= <S>
- S ::= (empty)
- This is the language D2, the language of two kinds of balanced parens
  - E.g. ((<<>>))
- Well not quite D2, since that should allow things like (())<>

Example, continued

- So add the rule
  - S ::= SS
- And that is indeed D2
- But this is ambiguous
  - ()<>() can be parsed two ways
  - ()<> is an S and () is an S
  - () is an S and <>() is an S
- Nothing wrong with ambiguous grammars

BNF (Backus Naur/Normal Form)

- Properly attributed to Sanskrit scholars
- An extension of CFG with
  - Optional constructs in []
  - Sequences {} = 0 or more
  - Alternation |
- All these are just short hands

BNF Shorthands

- IF ::= if EXPR then STM [else STM] fi
  - IF ::= if EXPR then STM fi
  - IF ::= if EXPR then STM else STM fi
- STM ::= IF | WHILE
  - STM ::= IF
  - STM ::= WHILE
- STMSEQ ::= STM {;STM}
  - STMSEQ ::= STM
  - STMSEQ ::= STM ; STMSEQ

Programming Language Syntax

- Expressed as a CFG where the grammar is closely related to the semantics
- For example
    - EXPR ::= PRIMARY {OP | PRIMARY}
    - OP ::= + | *
- Not good, better is
    - EXPR ::= TERM | EXPR + TERM
    - TERM ::= PRIMARY | TERM * PRIMARY
- This implies associativity and precedence

PL Syntax Continued

- No point in using BNF for tokens, since no semantics involved
  - ID ::= LETTER | LETTER ID
- Is actively confusing since the BC of ABC is not an identifier, and anyway there is no tree structure here
- Better to regard ID as a terminal symbol. In other words grammar is a grammar of tokens, not characters

Grammars and Trees

- A Grammar with a starting symbol naturally indicates a tree representation of the program
- Non terminal on left is root of tree node
- Right hand side are descendents
- Leaves read left to right are the terminals that give the tokens of the program

The Parsing Problem

- Given a grammar of tokens
- And a sequence of tokens
- Construct the corresponding parse tree
- Giving good error messages

General Parsing

- Not known to be easier than matrix multiplication
  - Cubic, or more properly n**2.71.. (whatever that unlikely constant is)
  - In practice almost always linear
  - In any case not a significant amount of time
  - Hardest part by far is to give good messages

Two Basic Approaches

- Table driven parsers
  - Given a grammar, run a program that generates a set of tables for an automaton
  - Use the standard automaton with these tables to generate the trees.
  - Grammar must be in appropriate form (not always so easy)
  - Error detection is tricky to automate

The Other Approach

- Hand Parser
    - Write a program that calls the scanner and assembles the tree
    - Most natural way of doing this is called recursive descent.
    - Which is a fancy way of saying scan out what you are looking for ☺

Recursive Descent in Action

- Each rule generates a procedure to scan out the procedure.
  - This procedure simply scans out its right hand side in sequence
- For example
  - IF ::= if EXPR then STM fi;
  - Scan “if”, call EXPR, scan “then”, call STM, scan “fi” done.

Recursive Descent in Action

- For an alternation we have to figure out which way to go (how to do that, more later, could backtrack, but that’s exponential)
- For optional stuff, figure out if item is present and scan if it is
- For a {repeated} construct program a loop which scans as long as item is present

Left Recursion ☹

- Left recursion is a problem
  - STMSEQ ::= STMSEQ STM | STM
- If you go down the left path, you are quickly stuck in an infinite recursive loop, so that will not do.
- Change to a loop
  - STMSEQ ::= STM {STM}

Ambiguous Alternation ☹

- If two alternatives
  - A ::= B | C
- Then which way to go
  - If set of initial tokens possible for B (called First(B)) is different from set of initial tokens of C, then we can tell
  - For example
    - STM ::= IFSTM | WHILESTM
    - If next token “if” then IFSTM, else if next token is “while then WHILESTM

Really Ambiguous Cases ☹

- Suppose FIRST sets are not disjoint
  - IFSTM ::= IF_SIMPLE | IF_ELSE
  - IF_SIMPLE ::= if EXPR then STM fi
  - IF_ELSE ::= if EXPR then STM else STM fi
- Factor left side
  - IFSTM ::= IFCOMMON IFTAIL
  - IFCOMMON ::= if EXPR then STM
  - IFTAIL ::= fi | else STM fi
- Last alternation is now distinguished

Recursive Descent, Errors

- If you don’t find what you are looking for, you know exactly what you are looking for so you can usually give a useful message
- IFSTM ::= if EXPR then STM fi;
  - Parse if a > b then b := g ;
  - Missing FI!

Recursive Descent, Last Word

- Don’t need much formalism here
- You know what you are looking for
- So scan it in sequence
- Called recursive just because rules can be recursive, so naturally maps to recursive language
- Really not hard at all, and not something that requires a lot of special knowledge

Table Driven Techniques

- There are parser generators that can be used as black boxes, e.g. bison
- But you really need to know how they work
- And that we will look at next time