



Languages, Automata, Regular Expressions & Scanners

BCS 307 – Compiler Construction

Agenda

- Basic concepts of formal languages and grammars
- Regular expressions
- Lexical specification of programming languages
- Using finite automata to recognize regular expressions
- Scanners and Tokens

Programming Language specifications

- Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
 - First done in 1959 with BNF (Backus-Naur Form) used to specify ALGOL 60 syntax

Grammar for a Tiny Language

- $\text{program} ::= \text{statement} \mid \text{program statement}$
- $\text{statement} ::= \text{assignStmt} \mid \text{ifStmt}$
- $\text{assignStmt} ::= \text{id} = \text{expr} ;$
- $\text{ifStmt} ::= \text{if} (\text{expr}) \text{statement}$
- $\text{expr} ::= \text{id} \mid \text{int} \mid \text{expr} + \text{expr}$
- $\text{id} ::= \text{a} \mid \text{b} \mid \text{c} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{n} \mid \text{x} \mid \text{y} \mid \text{z}$
- $\text{int} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Productions

- The rules of a grammar are called *productions*
- Rules contain:
 - Nonterminal symbols: grammar variables (*program, statement, id, etc.*)
 - Terminal symbols: concrete syntax that appears in programs (*a, b, c, 0, 1, if, (,), ...*)
- Nonterminal is: sequence of terminals and nonterminals
 - In a derivation, an instance of *nonterminal* can be replaced by the sequence of terminals and nonterminals on right of the production
- Often, there are two or more productions for one nonterminal
 - use any of them in different parts of derivation

Alternative Notations

- There are several notations for productions in use; all mean the same thing

$ifStmt ::= if (expr) statement$

$ifStmt \rightarrow if (expr) statement$

$\langle ifStmt \rangle ::= if (\langle expr \rangle) \langle statement \rangle$

Example Derivation

program ::= *statement* | *program*
statement
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
Id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

a = 1 ; if (a + 1) b = 2 ;

Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from a concrete, character-by-character grammar
- In practice this is never done

Parsing & Scanning

- In Production compilers the recognizer is split into two phases
 - Scanner: translates input characters to tokens
 - Also, report lexical errors like illegal characters and illegal symbols
 - Parser: reads token stream and reconstruct the derivation



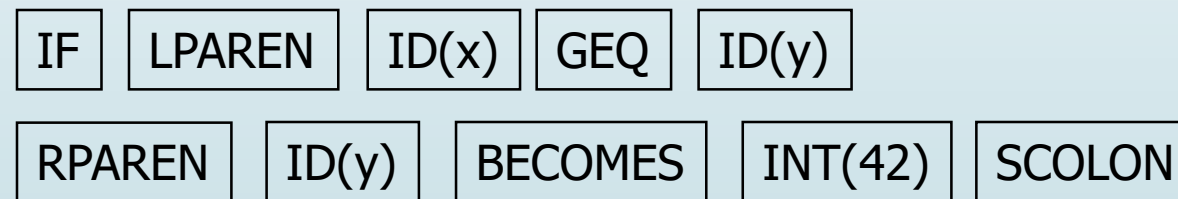
Characters vs Tokens

► Input text

// this statement does very little

if (x >= y) y = 42;

► Token Stream



Why Separate the Scanner and Parser?

- Simplicity & Separation of Concerns
 - Scanner hides details from parser (comments, whitespace, input files, etc.)
 - Parser is easier to build; has simpler input stream (tokens)
- Efficiency
 - Scanner can use simpler, faster design
 - (But still often consumes a surprising amount of the compiler's total execution time)

Tokens

- Idea: we want a distinct token kind (lexical class) for each distinct terminal symbol in the programming language
 - Examine the grammar to find these
- Some tokens may have attributes
 - Examples: integer constant token will have the actual integer (17, 42, ...) as an attribute; identifiers will have a string with the actual id

Typical Tokens in Programming Languages

- Operators & Punctuation
 - + - * / () { } [] ; : :: < <= == != ! ...
 - Each of these is normally a distinct lexical class
- Keywords
 - if while for goto return switch void ...
 - Each of these is also a distinct lexical class (*not* a string)
- Identifiers
 - A single ID lexical class, but parameterized by actual id
- Integer constants
 - A single INT lexical class, but parameterized by int value
- Other constants, etc.

Principle of Longest Match

- The scanner should pick the longest possible string to make up the next token if there is a choice
- Example

return maybe != iffy;

should be recognized as 5 tokens

RETURN

ID(maybe)

NEQ

ID(iffy)

SCOLON

i.e., != is one token, not two, “iffy” is an ID, not IF followed by ID(fy)

Formal Languages & Automata Theory (review)

- Alphabet: a finite set of symbols
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set of strings, often infinite
- Finite specifications of (possibly infinite) languages
 - Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
 - Grammar – a generator; a system for producing all strings in the language (and no other strings)
- A particular language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

Regular Expressions and FAs

- The lexical grammar (structure) of programming languages can be specified with regular expressions
- Tokens can be recognized by a deterministic finite automaton

Regular Expressions

- Defined over some alphabet Σ
 - For programming languages, alphabet is usually ASCII or Unicode
- If re is a regular expression, $L(re)$ is the language (set of strings) generated by re

Fundamental REs

re	$L(re)$	Notes
a	$\{ a \}$	Singleton set, for each a in Σ
ε	$\{ \varepsilon \}$	Empty string
\emptyset	$\{ \}$	Empty language

Operations on REs

re	$L(re)$	Notes
rs	$L(r)L(s)$	Concatenation
$r s$	$L(r) \cup L(s)$	Combination (union)
r^*	$L(r)^*$	0 or more occurrences (Kleene closure)

- Precedence: $*$ (highest), concatenation, $|$ (lowest)
- Parentheses can be used to group REs as needed

Abbreviations

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Typical examples:

Abbr.	Meaning	Notes
r^+	(rr^*)	1 or more occurrences
$r?$	$(r \mid \epsilon)$	0 or 1 occurrence
$[a-z]$	$(a b \dots z)$	1 character in given range
$[abxyz]$	$(a b x y z)$	1 of the given characters

Examples

<i>re</i>	Meaning
+	single + character
!	single ! character
=	single = character
!=	2 character sequence
<=	2 character sequence
xyzzzy	5 character sequence

Fill the meaning

<i>re</i>	Meaning
[abc]+	
[abc]*	
[0-9]+	
[1-9][0-9]*	
[a-zA-Z][a-zA-Z0-9_]*	

Abbreviations

- Many systems allow abbreviations to make writing and reading definitions or specifications easier

`name ::= re`

- Restriction: abbreviations may not be circular (recursive) either directly or indirectly (else would be non-regular)

Example

- Possible syntax for numeric constants

digit ::= [0-9]

digits ::= *digit*⁺

number ::= *digits* (*.* *digits*)?

([eE] (+ | -)? *digits*) ?

- How would you describe this set in English?
- What are some examples of legal constants (strings) generated by *number*?

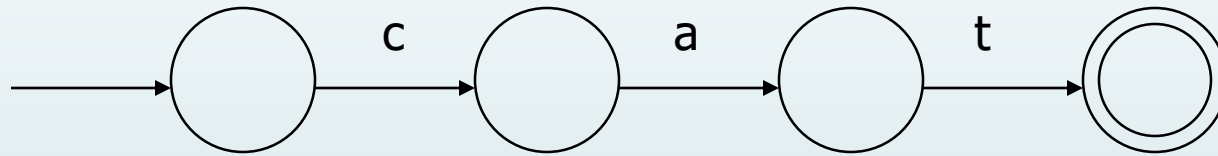
Recognizing REs

- ▶ Finite automata can recognize strings generated by regular expressions
- ▶ Not totally straightforward, but can be done systematically

Finite State Automaton

- A finite set of states
 - One marked as initial state
 - One or more marked as final states
 - States sometimes labeled or numbered
- A set of transitions from state to state
 - Each labeled with symbol from Σ , or ϵ
- Operate by reading input symbols (usually characters)
 - Transition can be taken if labeled with current symbol
 - ϵ -transition can be taken at any time
- Accept when final state reached & no more input
 - Scanner uses a FSA as a subroutine – accept longest match from current location each time called
 - Reject if no transition possible, or no more input and not in final state (DFA)

Example: FSA for “cat”



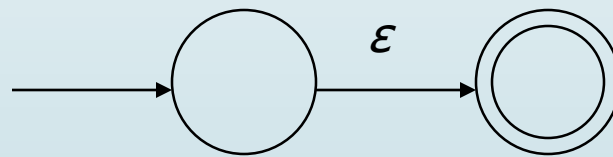
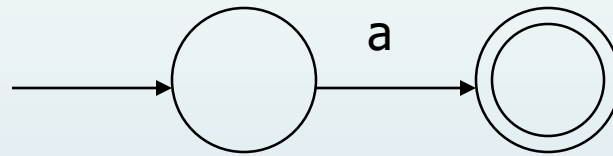
DFA vs NFA

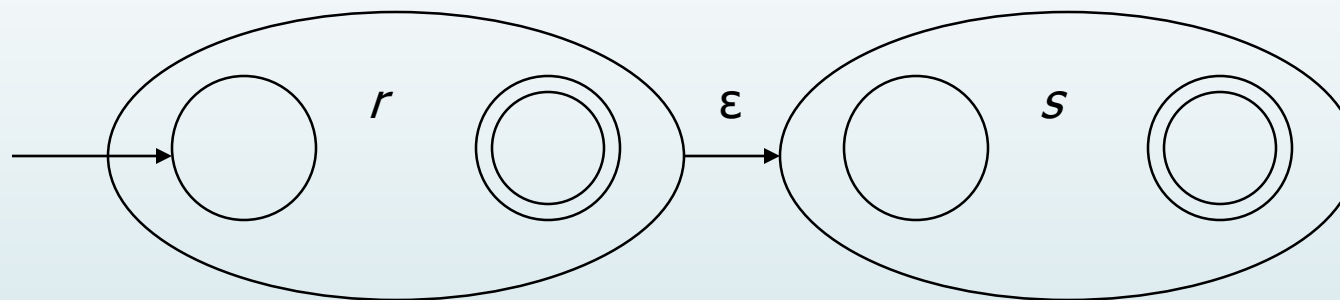
- Deterministic Finite Automata (DFA)
 - No choice of which transition to take under any condition
 - In particular, no ϵ transitions (arcs)
- Non-deterministic Finite Automata (NFA)
 - Choice of transition in at least one case
 - Accept if some way to reach final state on given input
 - Reject if no possible way to final state
 - i.e., may need to guess or backtrack

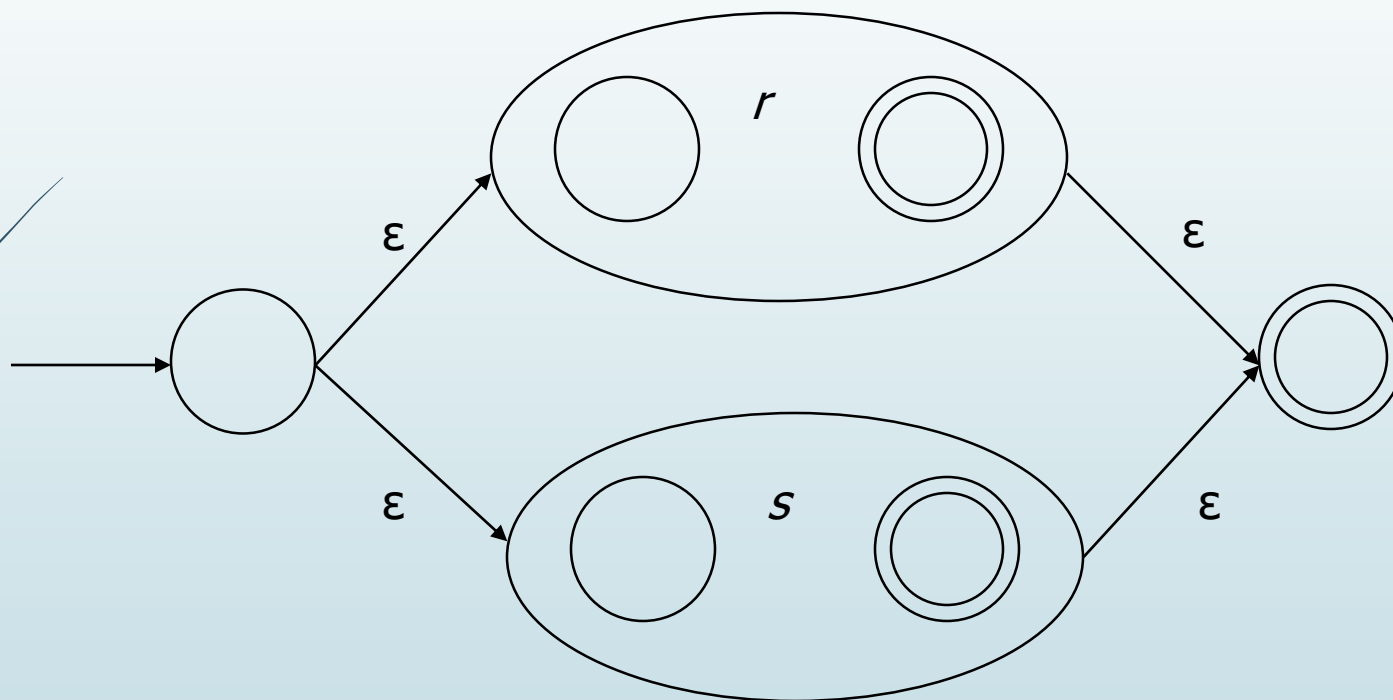
FAs in Scanners

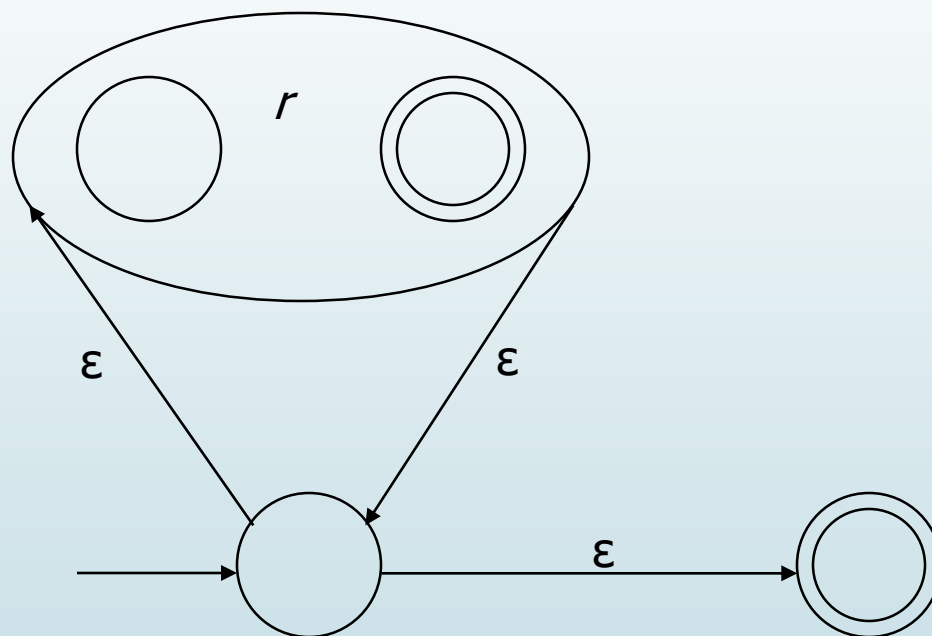
- DFA preferred for speed (no backtracking)
- Conversion from regular expressions to NFA is easy
- There is a well-defined procedure for converting a NFA to an equivalent DFA

From RE to NFA: base cases





$r \mid s$ 



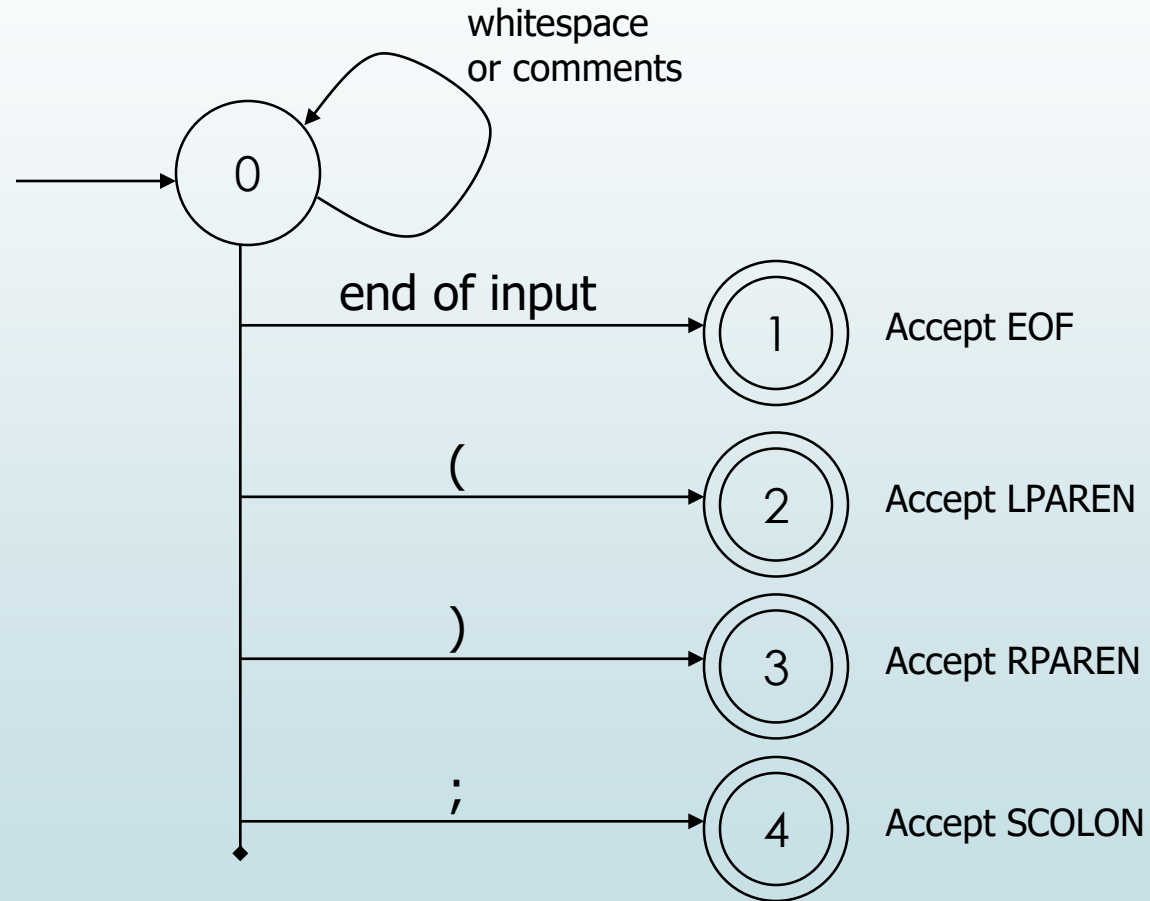
From NFA to DFA

- Subset construction
 - Construct a DFA from the NFA, where each DFA state represents a *set* of NFA states
- Key idea
 - The state of the DFA after reading some input is the set of *all* states the NFA could have reached after reading the same input
- Algorithm: example of a fixed-point computation
- If NFA has n states, DFA has at most 2^n states
 - \Rightarrow DFA is finite, can construct in finite # steps
- Resulting DFA may have more states than needed
 - See books for construction and minimization details

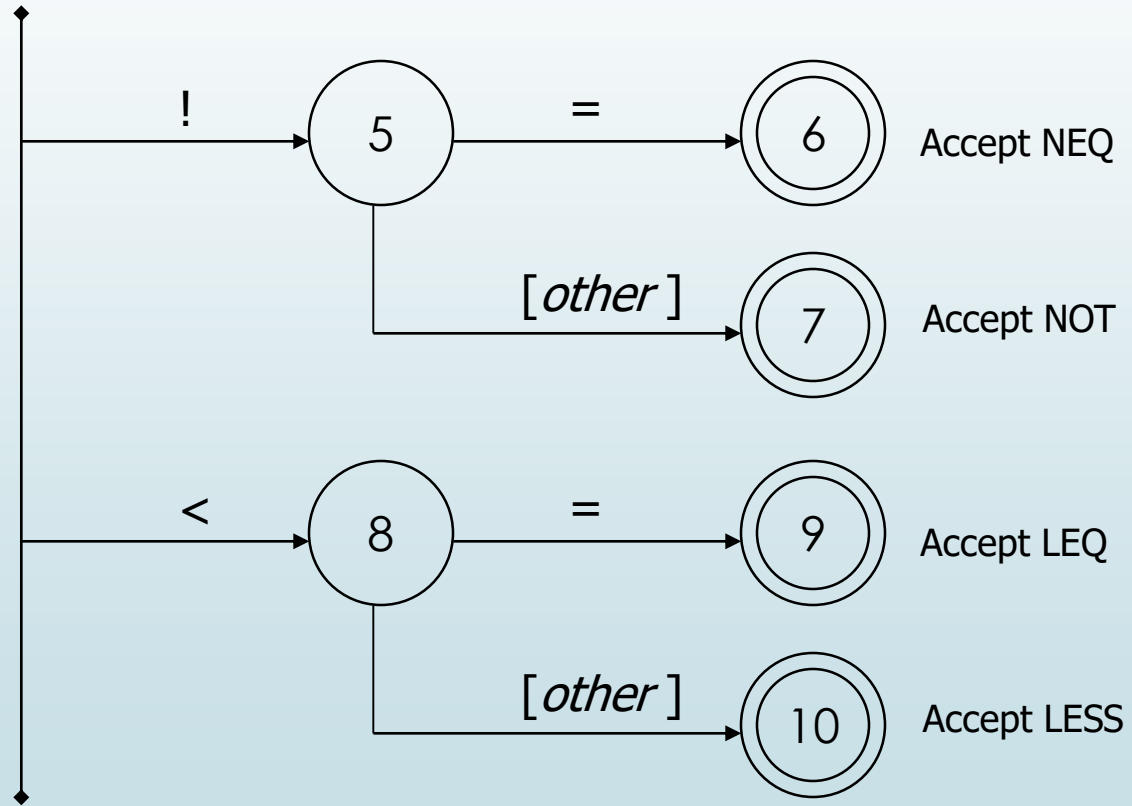
Example: DFA for hand-written scanner

- Idea: show a hand-written DFA for some typical programming language constructs
 - Then use to construct hand-written scanner
- Setting: Scanner is called whenever the parser needs a new token
 - Scanner stores current position in input
 - Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token

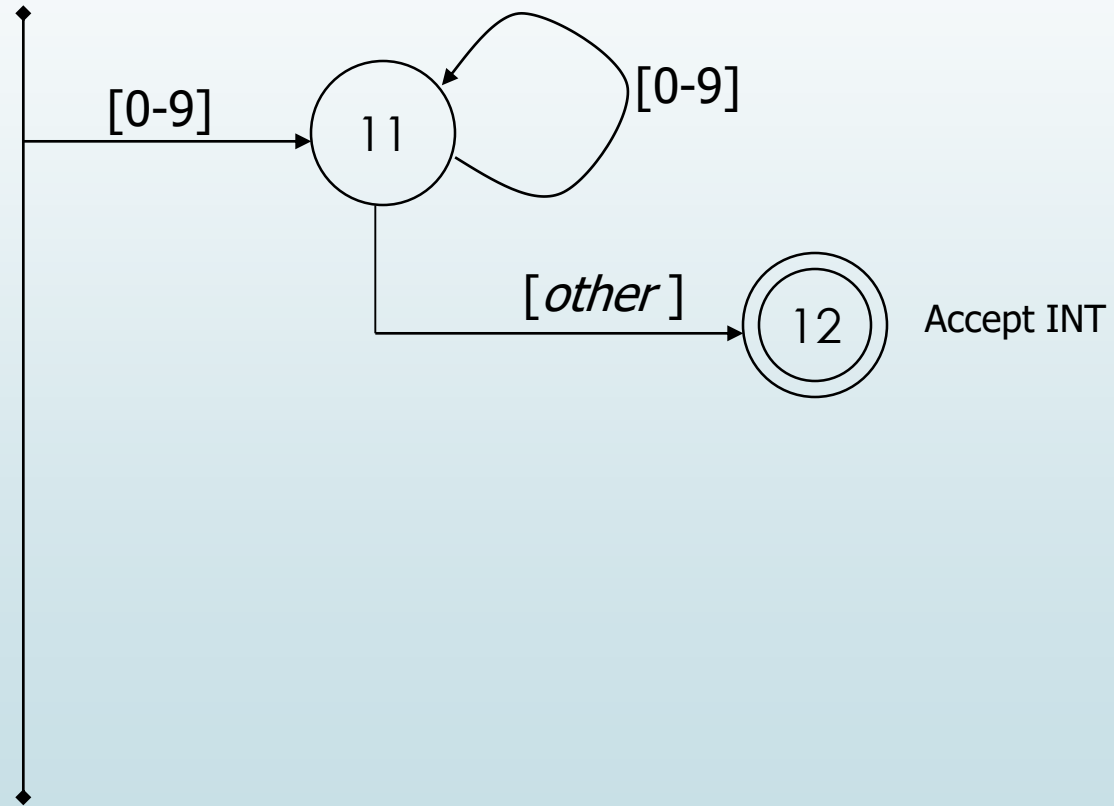
Scanner DFA Example (1)



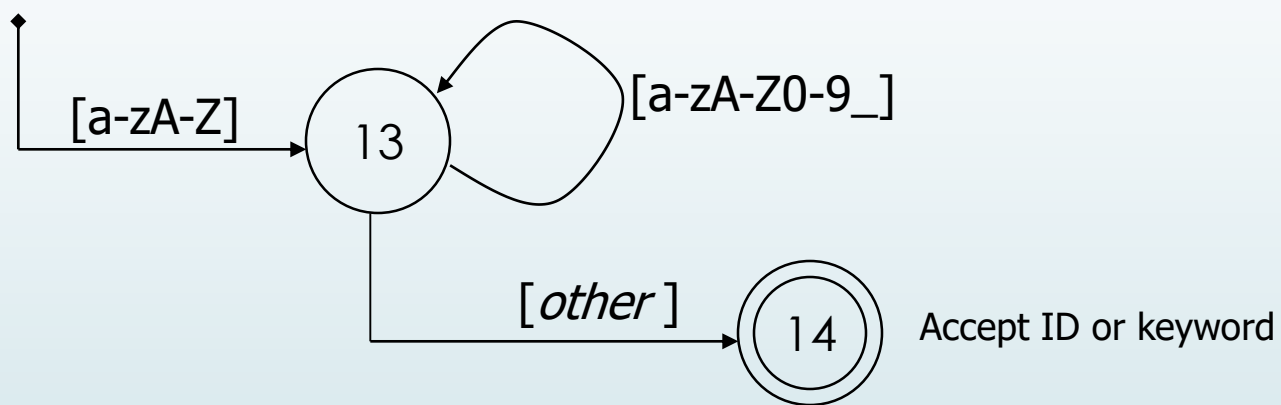
Scanner DFA Example (2)



Scanner DFA Example (3)



Scanner DFA Example (4)



- Strategies for handling identifiers vs keywords
 - Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
 - Machine-generated scanner: generate DFA with appropriate transitions to recognize keywords
 - Lots 'o states, but efficient (no extra lookup step)

Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure

```
public class Token {  
    public int kind;          // token's lexical class  
    public int intVal; // integer value if class = INT  
    public String id;        // actual identifier if class = ID  
    // lexical classes  
    public static final int EOF = 0;    // "end of file" token  
    public static final int ID  = 1;    // identifier, not keyword  
    public static final int INT = 2;    // integer  
    public static final int LPAREN = 4;  
    public static final int SCOLN  = 5;  
    public static final int WHILE  = 6;  
    // etc. etc. etc. ...  
}
```

better: use
enums if you
have them

Simple Scanner Example

```
// global state and methods
```

```
static char nextch;    // next unprocessed input character
```

```
// advance to next input char  
void getch() { ... }
```

```
// skip whitespace and comments  
void skipWhitespace() { ... }
```

Scanner getToken() method

```
// return next input token
public Token getToken() {
    Token result;

    skipWhiteSpace();

    if (no more input) {
        result = new Token(Token.EOF); return result;
    }

    switch(nextch) {
        case '(': result = new Token(Token.LPAREN); getch(); return result;
        case ')': result = new Token(Token.RPAREN); getch(); return result;
        case ';': result = new Token(Token.SCOLON); getch(); return result;

        // etc. ...
    }
}
```

getToken() (2)

```
case '!': // ! or !=
    getch();
    if (nextch == '=') {
        result = new Token(Token.NEQ); getch(); return result;
    } else {
        result = new Token(Token.NOT); return result;
    }

case '<': // < or <=
    getch();
    if (nextch == '=') {
        result = new Token(Token.LEQ); getch(); return result;
    } else {
        result = new Token(Token.LESS); return result;
    }
// etc. ...
```

getToken() (3)

```
case '0': case '1': case '2': case '3': case '4':  
case '5': case '6': case '7': case '8': case '9':  
    // integer constant  
    String num = nextch;  
    getch();  
    while (nextch is a digit) {  
        num = num + nextch; getch();  
    }  
    result = new Token(Token.INT, Integer(num).intValue());  
    return result;  
...
```

getToken() (4)

```
case 'a': ... case 'z':  
case 'A': ... case 'Z': // id or keyword  
    string s = nextch; getch();  
    while (nextch is a letter, digit, or underscore) {  
        s = s + nextch; getch();  
    }  
    if (s is a keyword) {  
        result = new Token(keywordTable.getKind(s));  
    } else {  
        result = new Token(Token.ID, s);  
    }  
    return result;
```

Project Notes

- Implement the token recognition code in C#

Next on Agenda

- ▀ Parsing
 - ▀ LR parsing