

CS 335: Lexical Analysis

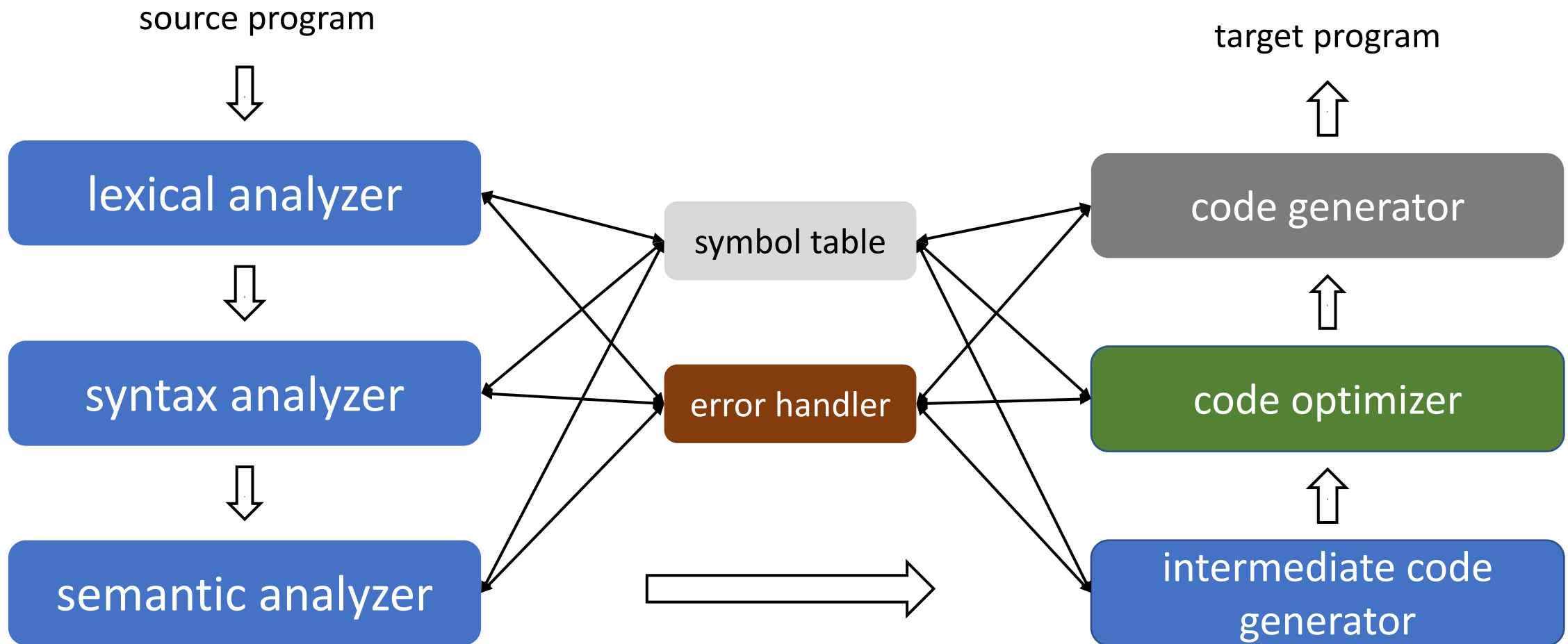
Swarnendu Biswas

Semester 2022-2023-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

An Overview of Compilation



Overview of Lexical Analysis

- First stage of a three-part frontend to help understand the source program
 - Processes every character in the input program
 - If a word is valid, then it is assigned to a syntactic category
 - This is similar to identifying the part of speech of an English word

Compilers are engineered objects.

noun verb adjective noun punctuation

Description of Lexical Analysis

- **Input:**

- A high level language (e.g., C++ and Java) program in the form of a sequence of ASCII characters

- **Output:**

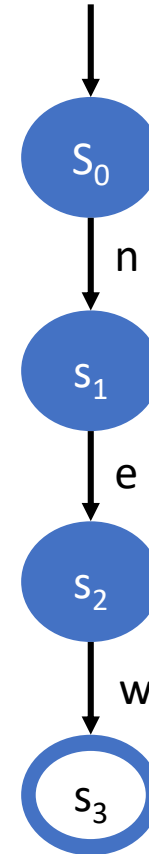
- A sequence of tokens along with attributes corresponding to different syntactic categories that is forwarded to the parser for syntax analysis

- **Functionality:**

- Strips off blanks, tabs, newlines, and comments from the source program
- Keeps track of line numbers and associates error messages from various parts of a compiler with line numbers
- Performs some preprocessor functions in languages like C

Recognizing Word “new”

```
c = getNextChar();
if (c == 'n')
    c = getNextChar();
    if (c == 'e')
        c = getNextChar();
        if (c == 'w')
            report success;
        else
            // Other logic
    else
        // Other logic
else
    // Other logic
```



Formalism for Scanners

Regular expressions, DFAs, and NFAs

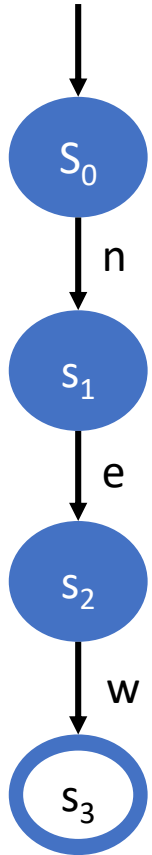
Definitions

- An **alphabet** is a finite set of symbols
 - Typical symbols are letters, digits, and punctuations
 - ASCII and UNICODE are examples of alphabets
- A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet
- A **language** is any countable set of strings over a fixed alphabet

Finite State Automaton

- A finite state automaton (FSA) is a five-tuple or quintuple $(S, \Sigma, \delta, s_0, S_F)$
 - S is a finite set of states
 - Σ is the alphabet or character set, is the union of all edge labels in the FSA, and is finite
 - $\delta(s, c)$ represents the transition from state s on input c
 - $s_0 \in S$ is the designated start state
 - $S_F \subseteq S$ is the set of final states
- A FSA accepts a string x if and only if
 - i. FSA starts in s_0
 - ii. Executes transitions for the sequence of characters in x
 - iii. Final state is an accepting state $\in S_F$ after x has been consumed

FSA for recognizing “new”



- FSA = $(S, \Sigma, \delta, s_0, S_F)$
 - $S = (s_0, s_1, s_2, s_3)$
 - $\Sigma = \{n, e, w\}$
 - $\delta = \{s_0 \xrightarrow{n} s_1, s_1 \xrightarrow{e} s_2, s_2 \xrightarrow{w} s_3\}$
 - $s_0 = s_0$
 - $S_F = \{s_3\}$

String is recognized in time proportional to the **input**

FSA for Unsigned Integers

```
char = getNextChar( )
state = s0
while (char ≠ EOF and state ≠ se)
    state = δ(state, char)
    char = getNextChar()

if (state ∈ SF)
    report success
else
    report failure
```

s_e is the error state

- FSA = (S, Σ, δ, s₀, S_F)
 - S = (s₀, s₁, s₂, s_e)
 - Σ = {0,1,2,3,4,5,6,7,8,9}
 - δ = {s₀⁰ → s₁, s₀¹⁻⁹ → s₂,
s₁⁰⁻⁹ → s₂, s₂ → s₂, s₁ → s_e}
 - s₀ = s₀
 - S_F = {s₁, s₂}

Dealing with Erroneous Situations

- FSA is in state s , the next input character is c , and $\delta(s, c)$ is not defined
- FSA processes the complete input and is still not in the final state
 - Input string is a proper prefix for some word accepted by the FSA

Nondeterministic Finite Automaton

- NFA is a FSA that allows transitions on the empty string ϵ and can have states that have multiple transitions on the same input character
- Simulating an NFA
 - Always make the correct nondeterministic choice to follow transitions that lead to accepting state(s) for the input string, if such transitions exist
 - Try all nondeterministic choices in parallel to search the space of all possible configurations
- Simulating a DFA is more efficient than an NFA

Regular Expressions

- The set of words accepted by an FSA F is called its language $L(F)$
- For any FSA F , we can also describe $L(F)$ using a notation called a Regular Expressions (RE)
- The language described by a RE r is called a regular language (denoted by $L(r)$)

Regular Expressions

- ϵ is a RE, $L(\epsilon) = \{\epsilon\}$
- Let Σ be an alphabet. For each $a \in \Sigma$, a is a RE, and $L(a) = \{a\}$.
- Let r and s be REs denoting the languages R and S , respectively
 - **Alternation** (or **union**): $(r|s)$ is a RE, $L(r|s) = R|S = \{x \mid x \in R \text{ or } x \in S\} = L(r) \cup L(s)$
 - **Concatenation**: (rs) is a RE, $L(rs) = R.S = \{xy \mid x \in R \wedge y \in S\}$
 - **Closure**: (r^*) is an RE, $L(r^*) = R^* = \bigcup_{i=0}^{\infty} R^i$
 - L^* is called the Kleene closure or closure of L

Examples of Regular Expressions

$L =$ set of all strings of 0's and 1's

$$r = (0 + 1)^*$$

$L = \{w \in \{0,1\}^* \mid w \text{ has two or three occurrences of } 1, \text{ the first and second are not consecutive}\}$

$$r = 0^*10^*010^*(10^* + \epsilon)$$

$L = \{w \mid w \in \{a,b\}^* \wedge w \text{ ends with } a\}$

$$r = (a + b)^*a$$

Examples of Regular Expressions

Unsigned real numbers with exponents

$$r = (0|[1 \dots 9][0 \dots 9]^*)(.[0 \dots 9]^*|\epsilon)E(+|-|\epsilon)(0|[1 \dots 9][0 \dots 9]^*)$$

$L = \{w \in \{0,1\}^* \mid w \text{ has no pair of consecutive zeros}\}$

$$r = (1 + 01)^*(0 + \epsilon)$$

Regular Expressions

- We can reduce the use of parentheses by introducing precedence and associativity rules
 - Binary operators, closure, concatenation, and alternation are left associative
- Precedence rule is

parentheses > closure > concatenation > alternation

Algebraic Rules for REs

Rule	Description
$r s = s r$	is commutative
$r (s t) = (r s) t$	is associative
$r(st) = (rs)t$	Concatenation is commutative
$r(s t) = rs rt; (s t)r = sr tr$	Concatenation distributes over
$\epsilon r = r\epsilon = r$	ϵ is the identity of concatenation
$r^* = (r \epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Regular Definitions

- Let r_i be a regular expression and d_i be a distinct name
- Regular Definition is a sequence of definitions of the form

$$\begin{aligned}d_1 &\rightarrow r_1 \\d_2 &\rightarrow r_2 \\&\dots \\d_n &\rightarrow r_n\end{aligned}$$

- Each r_i is a regular expression over the symbols $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$
- Each d_i is a new symbol not in Σ

Example of Regular Definitions

- Unsigned numbers (e.g., 5280, 0.01234, 6.336E4, or 1.89E-4)

<i>digit</i>	=	0 1 2 3 4 5 6 7 8 9
<i>digits</i>	=	<i>digit digit*</i>
<i>optfrac</i>	=	<i>. digits ε</i>
<i>optexp</i>	=	<i>(E(+ - ε) digits) ε</i>
<i>unsignednum</i>	=	<i>digits optfrac optexp</i>

Extensions of Regular Expressions

`“.”` is any character other than `“\n”`

`[xyz]` is `x|y|z`

`[abg-pT-Y]` is any character `a, b, g, ..., p, T, ..., Y`

`[^G-Q]` is not any one of `G, H, ..., Q`

`r+` is one or more `r`'s

`r?` is zero or one `r`

Example of Regular Definitions

- Unsigned numbers

- Example: 5280, 0.01234, 6.336E4, or 1.89E-4

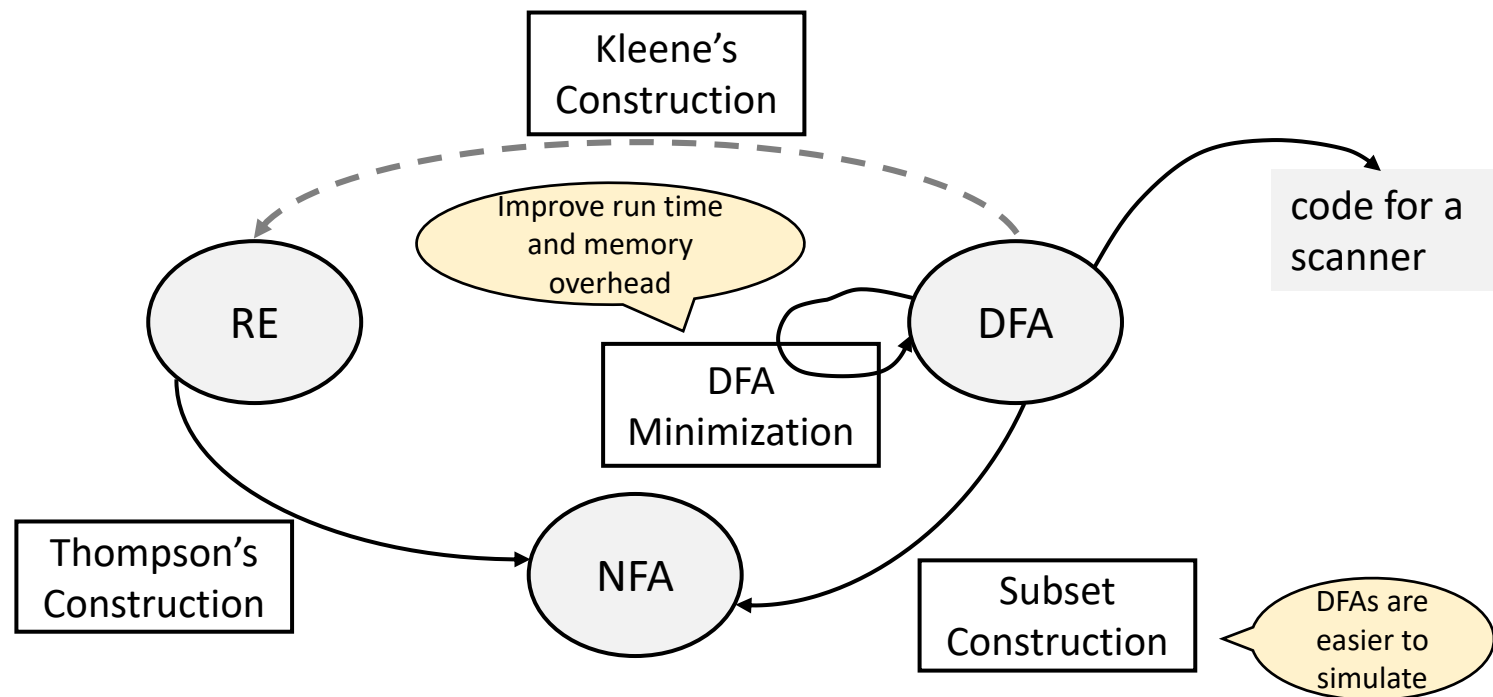
<i>digit</i>	=	0 1 2 3 4 5 6 7 8 9
<i>digits</i>	=	<i>digit digit</i> *
<i>optfrac</i>	=	. <i>digits</i> ϵ
<i>optexp</i>	=	(<i>E</i> (+ - ϵ) <i>digits</i>) ϵ
<i>unsignednum</i>	=	<i>digits optfrac optexp</i>

<i>digits</i>	=	[0-9]
<i>digits</i>	=	<i>digit</i> ⁺
<i>unsignednum</i>	=	<i>digits</i> (. <i>digits</i>)? (<i>E</i> [+-]? <i>digits</i>)?

Simpler to write

Equivalence of RE and FSA

- There exists an NFA with ϵ -transitions that accepts $L(r)$, where r is a RE
- If L is accepted by a DFA, then L is generated by a RE
- ...



$$\text{NFA} = (N, \Sigma, \delta_N, n_0, N_A)$$

$$\text{DFA} = (D, \Sigma, \delta_D, d_0, D_A)$$

NFA to DFA: Subset Construction

Subset Construction

```
q0 = ε-closure({s0})
Q = q0
WorkList = {q0}
while (WorkList ≠ ∅) do
  remove q from WorkList
  for each character c ∈ Σ do
    t = ε-closure(δ(q, c))
    T[q, c] = t
    if t ∉ Q then
      add t to Q and to WorkList
```

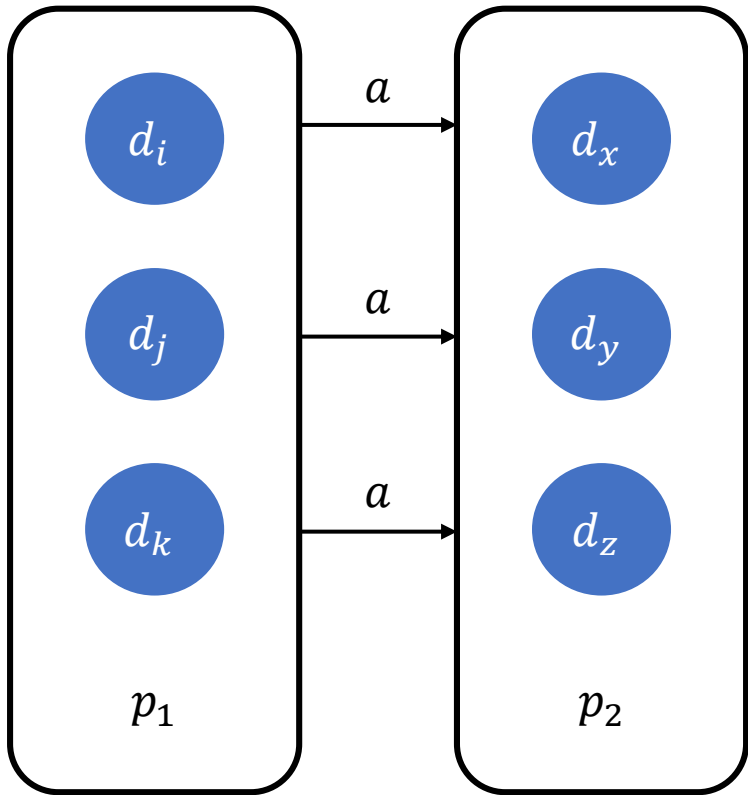
ε-closure

```
for each state n ∈ N do
  E(n) = {n}
WorkList = N
while (WorkList ≠ ∅) do
  remove n from WorkList
  t = {n} ∪ ∪n→p ∈ δNε E(p)
  if t ≠ E(n)
    E(n) = t
  WorkList = WorkList ∪ {m | m ε→ n ∈ δN}
```

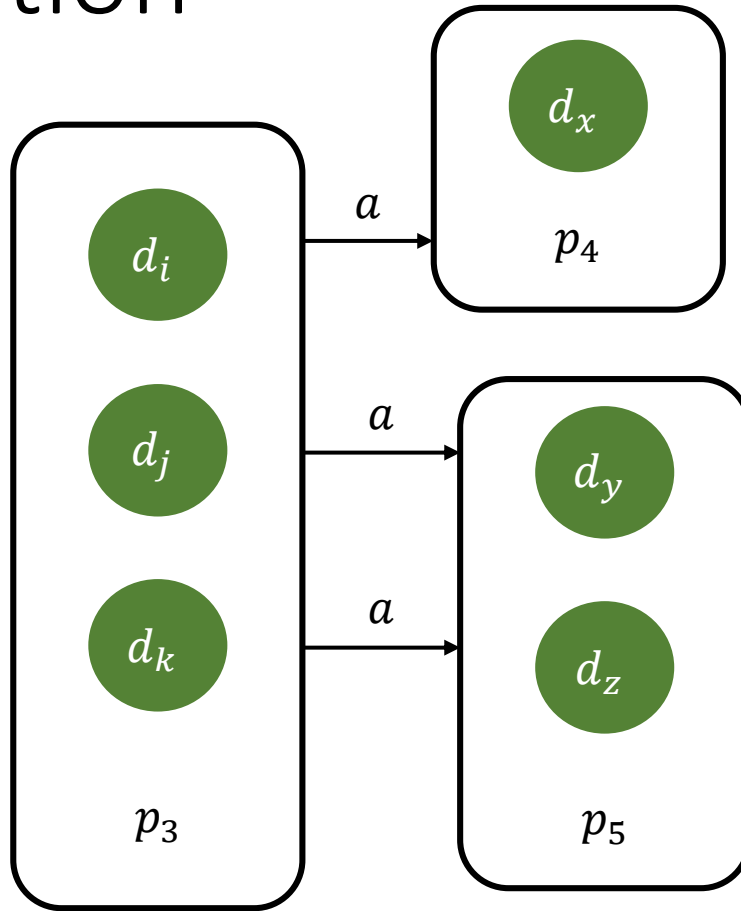

DFA to Minimal DFA: Hopcroft's Algorithm

- A DFA from Subset construction can have a large number of states
 - Does not increase the time needed to scan a string
- Increases the space requirement of the scanner in memory
 - Speed of accesses to main memory may turn out to be the bottleneck
 - Smaller scanner has better chances of fitting in the processor cache

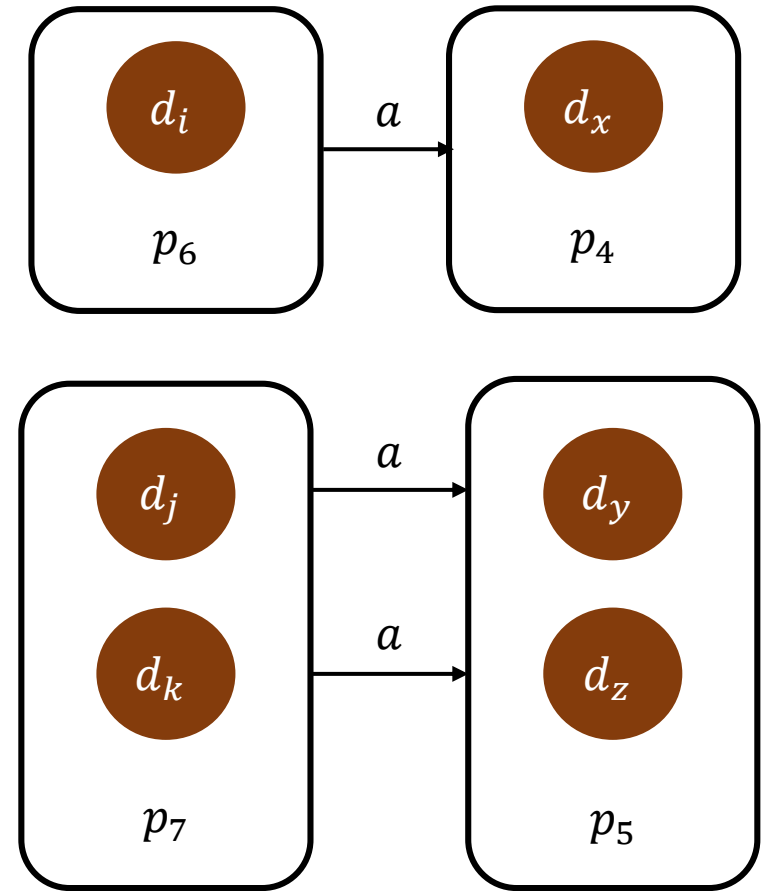
Splitting a Partition



a does not split p_1



a splits p_3



Partitions after splitting on a

DFA to Minimal DFA: Hopcroft's Algorithm

Minimization

```
 $T = \{D_A, \{D - D_A\}\}$   
 $P = \phi$   
while( $P \neq T$ ) do  
   $P = T$   
   $T = \phi$   
  for each set  $p \in P$  do  
     $T = T \cup \text{Split}(p)$ 
```

Split(S)

```
for each  $c \in \Sigma$  do  
  if  $c$  splits  $S$  into  $s_1$  and  $s_2$   
    return  $\{s_1, s_2\}$   
return  $S$ 
```

Realizing Scanners

Tokens

f	l	o	a	t		a	b	s	_	z	e	r	o		=		-	2	7	3	;	/	*	K	e	l	v	i	n	*	/
---	---	---	---	---	--	---	---	---	---	---	---	---	---	--	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- **Token**

- A string of characters which logically belong together in a syntactic category
- Sentences consist of a string of tokens (e.g., **float, identifier, assign, minus, intnum, semicolon**)
- Tokens are treated as terminal symbols of the grammar specifying the source language
- May have optional attributes
- Example of tokens in programming languages: Keywords, operators, identifiers (names), constants, literal strings, punctuation symbols (parentheses, brackets, commas, semicolons, and colons)

Patterns and Lexemes

- **Pattern**

- The rule describing the set of strings for which the same token is produced
- The pattern is said to match each string in the set
- `float`, `letter(letter|digit|_)*`, `=`, `-`, `digit+`, `;`

- **Lexeme**

- The sequence of characters matched by a pattern to form the corresponding token
- `"float"`, `"abs_zero"`, `"="`, `"-"`, `"273"`, `";"`

Attributes of Tokens

- An attribute of a token is a value that the scanner extracts from the corresponding lexeme and supplies to the syntax analyzer
- Examples attributes for tokens
 - identifier: the lexeme of the token, or a pointer into the symbol table where the lexeme is stored by the LA
 - intnum: the value of the integer (similarly for floatnum, etc.)
 - Type of the identifier, location where first found
 - The exact set of attributes are dependent on the compiler designer

Role of a Lexical Analyzer

- Identify tokens and corresponding lexemes
- Construct constants: for example, convert a number to token `intnum` and pass the value as its attribute
 - 31 becomes `<intnum, 31>`
- Recognize keyword and identifiers
 - `counter = counter + increment` becomes `id = id + id`
 - Check that `id` here is not a keyword
- Discard whatever does not contribute to parsing
 - White spaces (blanks, tabs, newlines) and comments

Specifying and Recognizing Patterns and Tokens

- Patterns are denoted with REs, and recognized with FSAs
- Regular definitions, a mechanism based on regular expressions, are popular for specification of tokens
- Transition diagrams, a variant of FSAs, are used to implement regular definitions and to recognize tokens
 - Usually used to model LA before translating them to executable programs

Transition Diagrams

- Transition diagrams (TDs) are generalized DFAs with the following differences
 - Edges may be labelled by a symbol, a set of symbols, or a regular definition
 - Few accepting states may be indicated as retracting states
 - Indicates that the lexeme does not include the symbol that transitions to the accepting state
 - Each accepting state has an action attached to it
 - Action is executed when the state is reached (e.g., return a token and its attribute value)

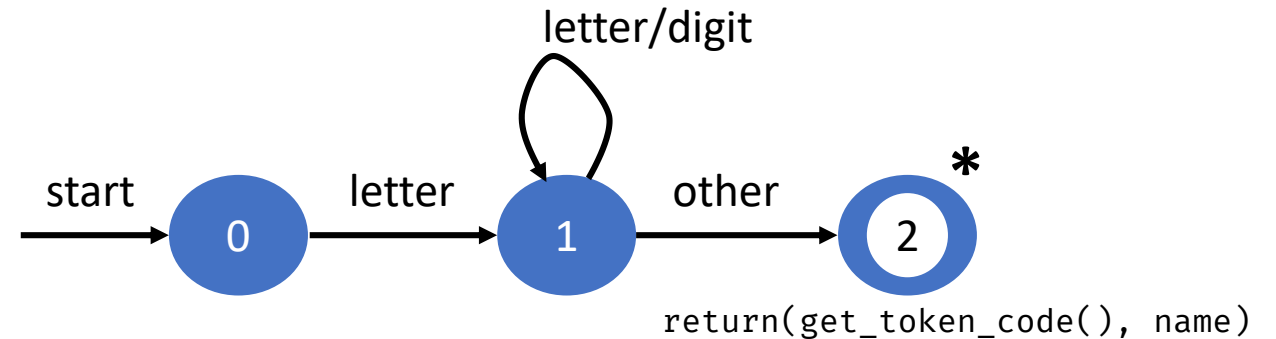
Examples of Transition Diagrams

Identifiers and reserved words

letter = [a-zA-Z]

digit = [0-9]

identifier = *letter*(*letter*|*digit*)*



- * indicates a retraction state
- `get_token_code()` searches a table to check if the name is a reserved word and returns its integer code if so
- Otherwise, it returns the integer code of the IDENTIFIER token, with name containing the string of characters forming the token
 - Name is not relevant for reserved words

A Sample Specification

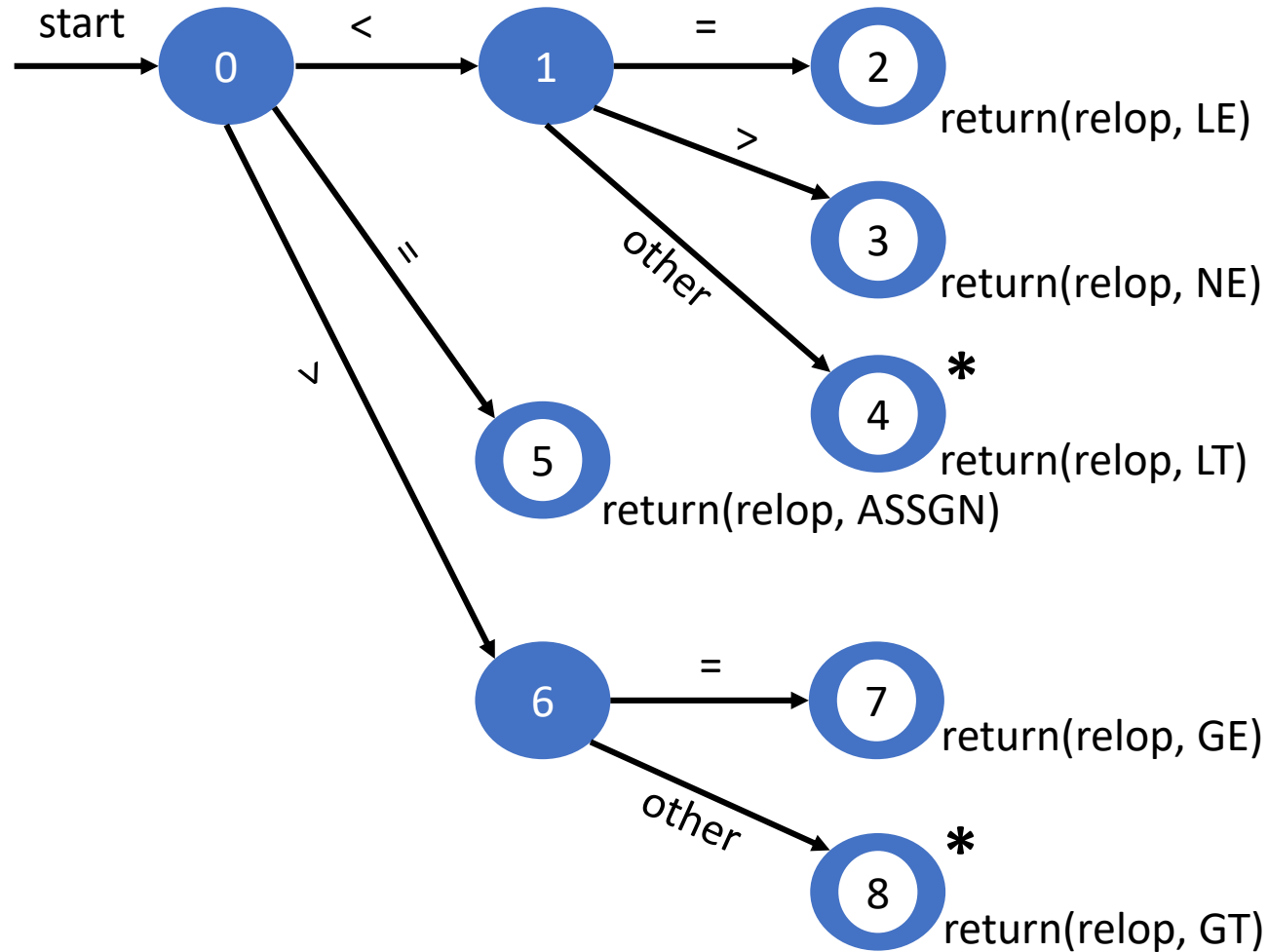
```
stmt → if expr then stmt  
      | if expr then stmt else stmt  
      |  $\epsilon$   
expr → term relop term  
      | term  
term → id  
      | number
```

```
digit → [0–9]  
digits → digit+  
number → digits (.digits)? (E[+–]? digits)?  
letter → [A–Za–z]  
id → letter (letter | digit)*  
if → if  
then → then  
else → else  
relop → < | > | <= | >= | = | <>  
ws → (blank | tab | newline)+
```

Tokens, Lexemes, and Attributes

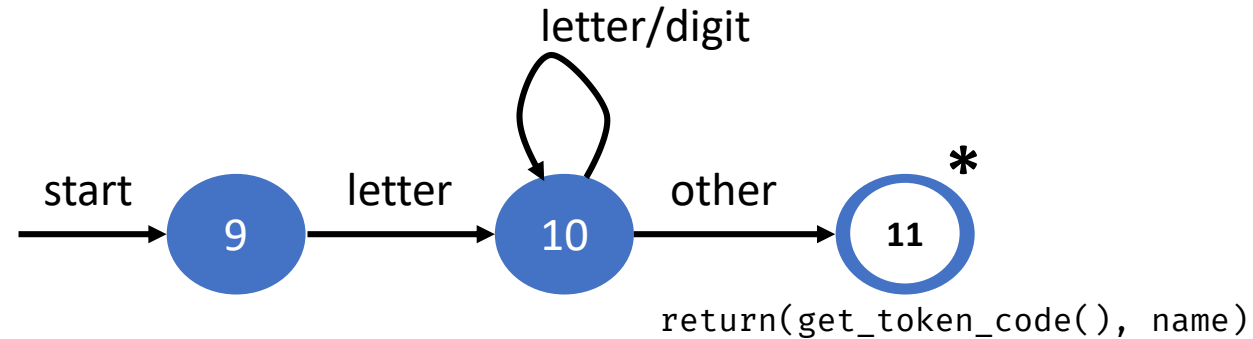
Lexemes	Token Name	Attribute Value
Any <i>ws</i>	--	--
<i>if</i>	if	--
<i>then</i>	then	--
<i>else</i>	else	--
Any <i>id</i>	id	Pointer to symbol table entry
Any <i>number</i>	number	Pointer to symbol table entry
<	relop	LT
<=	relop	LE
=	relop	ASSGN
<>	relop	NE
>	relop	GT
>=	relop	GE

Transition Diagram for **relop**

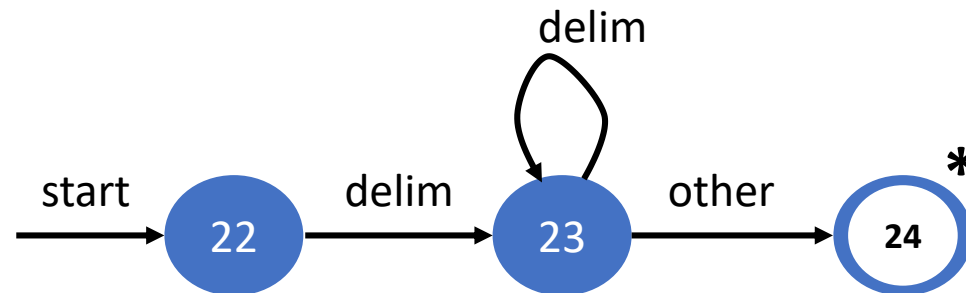


Transition Diagrams for IDs and Keywords

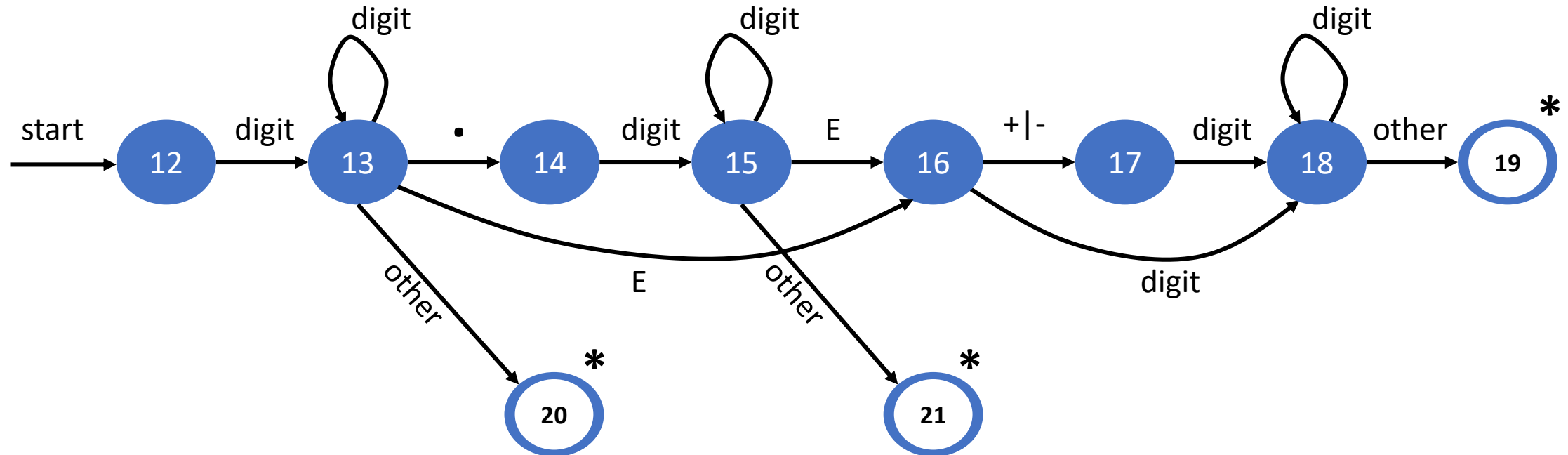
IDs and Keywords



Whitespace



Transition Diagram for Unsigned Numbers



Combining Transition Diagrams to form a Lexical Analyzer

- Different transition diagrams (TDs) must be combined appropriately to yield a scanner

How do we do this?

Combining Transition Diagrams to form a Lexical Analyzer

- Different transition diagrams (TDs) must be combined appropriately to yield a scanner
 - Try different transition diagrams one after another
 - For example, TDs for reserved words, constants, identifiers, and operators could be tried in that order
 - However, this does not use the “longest match” characteristic
 - thenext should be an identifier, and not reserved word then followed by identifier ext
- To find the longest match, all TDs must be tried and the longest match must be used

Challenges in Lexical Analysis

- Certain languages like PL/I do not have any reserved words
 - `while`, `do`, `if`, and `else` are reserved in C but not in PL/I
 - Makes it difficult for the scanner to distinguish between keywords and user-defined identifiers

```
if then then then = else else else = then
```

```
if if then then = then + 1
```

Challenges in Lexical Analysis

- Certain languages like PL/I do not have any reserved words
 - `while`, `do`, `if`, and `else` are reserved in C but not in PL/I
 - Makes it difficult for the scanner to distinguish between keywords and user-defined identifiers
- PL/I declarations
 - `DECLARE(arg1, arg2, arg3, ..., argn)`
 - Cannot tell whether `DECLARE` is a keyword with variable definitions or is a procedure with arguments until after “)”
- Requires arbitrary lookahead and very large buffers
 - Worse, the buffers may have to be reloaded in case of wrong inferences

Challenges in Lexical Analysis

```
fi (a == g(x)) ...
```

- Is `fi` a typo or a function call?
 - Remember, `fi` is a valid lexeme for IDENTIFIER
- Think of C++
 - Template syntax: `Foo<Bar>`
 - Stream syntax: `cin >> var;`
 - Nested templates: `Foo<Bar<Bazz>>`
- Can these problems be resolved by lexical analysers alone? No, in some cases parser needs to help.

Challenges in Lexical Analysis

- Consider a fixed-format language like Fortran
 - 80 columns per line
 - Column 1-5 for the statement number/label column
 - Column 6 for continuation mark
 - Column 7-72 for the program statements
 - Column 73-80 Ignored (used for other purposes)
 - Letter C in Column 1 meant the current line is a comment

Challenges in Lexical Analysis

- In fixed-format Fortran, some keywords are context-dependent
 - In the statement, `D0 10 I = 10.86`, `D010I` is an identifier, and `D0` is not a keyword
 - But in the statement, `D0 10 I = 10, 86`, `D0` is a keyword
 - Blanks are not significant in Fortran and can appear in the midst of identifiers
 - Variable “`count er`” is same as “`count er`”
 - In Fortran, blanks are important only in literal strings
 - Reading from left to right, one cannot distinguish between the two until the “`,`” or “`.`” is reached
 - Requires look ahead for resolution

Programming Languages vs Natural Languages

- Meaning of words in natural languages is often context-sensitive
 - An English word can be a noun or a verb (for e.g., “stress”)
 - “are” is a verb, “art” is a noun, and “arz” is undefined
- Grammars are rigorously specified to provide meaning
 - Words in a programming language are always lexically specified
 - Any string in $(1\dots9)(0\dots9)^*$ is a positive integer

Why separate tokens and lexemes?

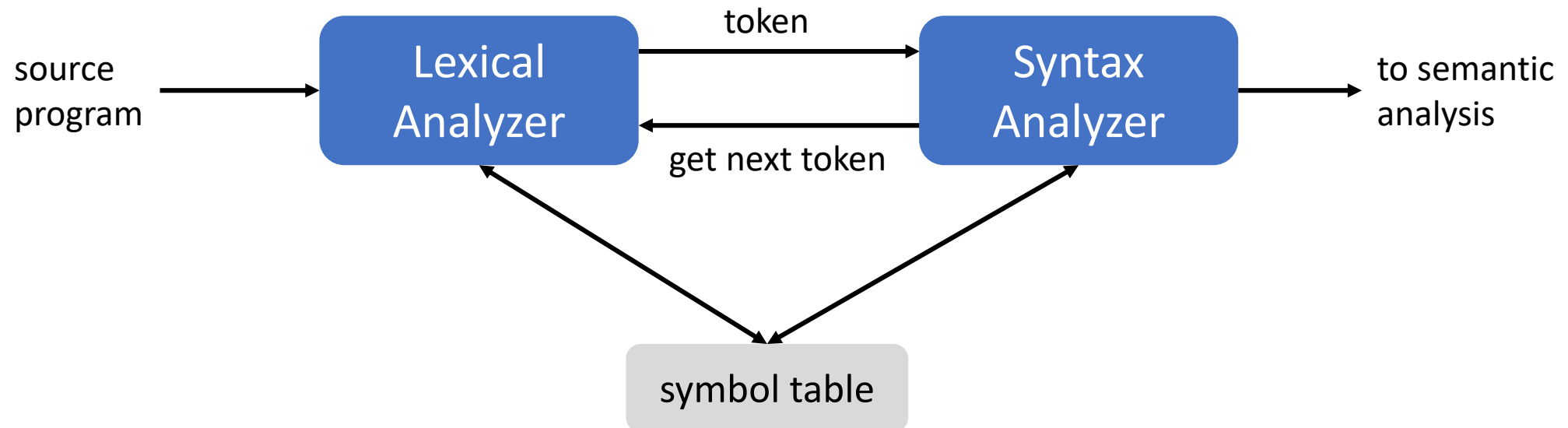
- Rules to govern the lexical structure of a programming language is called its **microsyntax**
- Separating syntax and microsyntax allows for a simpler parser
 - Parser only needs to deal with syntactic categories like IDENTIFIER

Lexical Analysis as a Separate Phase

1. Simplifies the compiler design: I/O issues are limited to only the lexical analyzer, leading to better portability
2. Allows designing a more compact and faster parser
 - Comments and whitespace need not be handled by the parser
 - No rules for numbers, names, and comments are needed in the parser
 - A parser is more complicated than a lexical analyzer and shrinking the grammar makes the parser more efficient
3. Scanners based on finite automata are more efficient to implement than stack-based pushdown automata used for parsing

Interfacing with Parser

- A unique integer representing the token is passed by LA to the parser



Error Handling in Lexical Analysis

- LA cannot catch any other errors except for simple errors such as illegal symbols
- In such cases, LA skips characters in the input until a well-formed token is found
 - This is called “panic mode” recovery
- We can think of other possible recovery strategies
 - Delete one character from the remaining input, or insert a missing character
 - Replace a character, or transpose two adjacent characters
 - Idea is to see if a single (or few) transformation(s) can repair the error

Other Uses of Lexical Analysis Concepts

- UNIX command line tools like grep, awk, and sed
- Search tools in editors
- Word-processing tools

Implementing Scanners

Implementing Scanners

1. Specify REs for each syntactic category in the PL
2. Construct an NFA for each RE
3. Join the NFAs with ϵ -transitions
4. Create the equivalent DFA
5. Minimize the DFA
6. Generate code to implement the DFA

Implementation Considerations

- Speed is paramount for scanning
 - Processes every character from a possibly large input source program
- Repeatedly read input characters and simulate the corresponding DFA
 - Types of scanner implementations: table-driven, direct-coded, and hand-coded
 - Asymptotic complexity is the same, differs in run-time costs

High-Level Idea in Implementing Scanners

Read input characters one by one

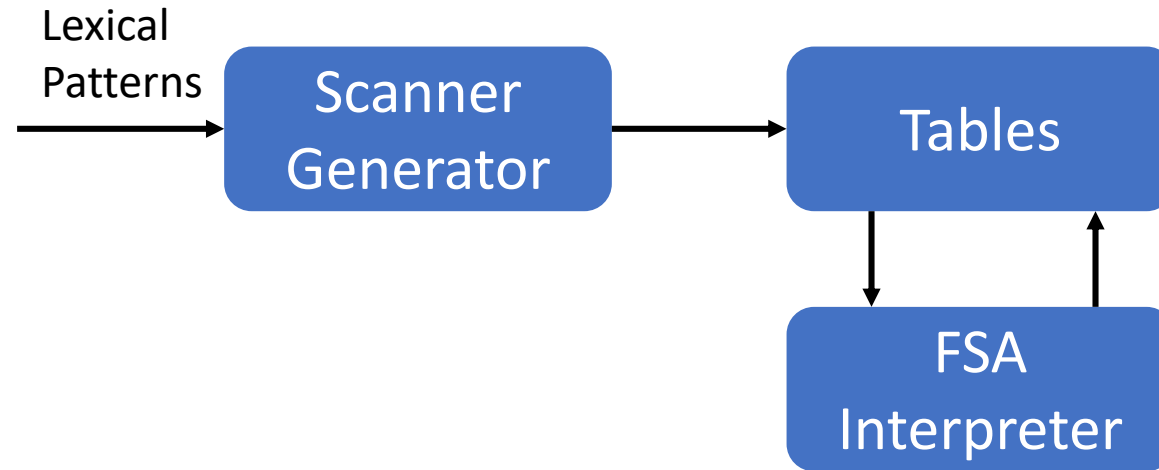
Look up the transition based on the current state and the input character

Switch to the new state

Check for termination conditions, i.e., accept and error

Repeat

Table-Driven Scanner



- Register specification
 - For example, r1 and r27

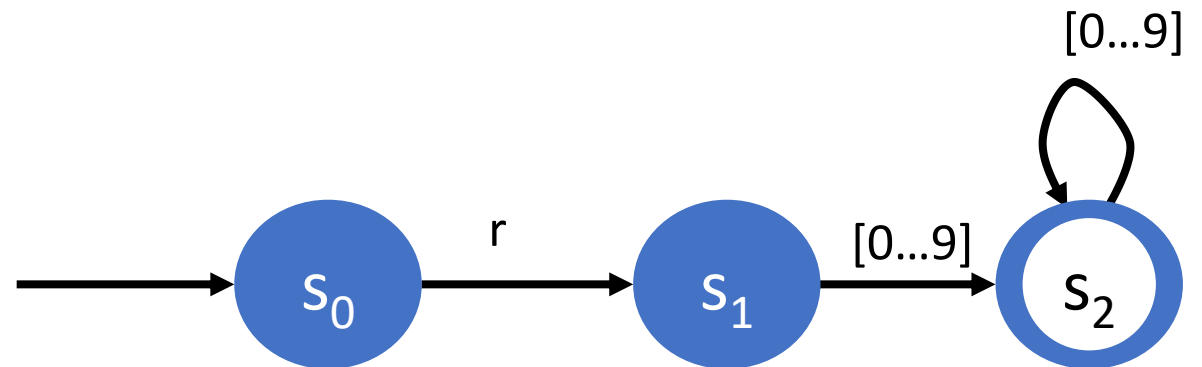


Table-Driven Scanner

```

state =  $s_0$ ; lexeme = "";
clear stack; push(bad);

// Model the DFA
while (state  $\neq s_e$ )
    char = getNextChar()
    lexeme = lexeme + char
    if state  $\in s_A$ 
        clear stack
        push(state)
    token = lookup(PATTERN)
    state =  $\delta$ (state, token)

```

}

involves two
table lookups

r	0, 1, 2, ..., 9	EOF	Other
Register	Digit	EOF	Other

δ	R	0,1,...,9	other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table-Driven Scanner

```

// Rollback
while (state  $\notin$   $s_A$  and state  $\neq$  bad)
    state = pop()
    truncate lexeme
    rollback()

if state  $\in$   $s_A$ 
    return token
else
    return invalid
    
```

r	0, 1, 2, ..., 9	EOF	Other
Register	Digit	EOF	Other

δ	R	0,1,...,9	other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Problem of Rollbacks

- A scanner's aim is to recognize the longest match but it can increase rollbacks
 - Consider the RE $ab \mid (ab)^*c$, and input *abababab*
- A scanner can avoid such pathological quadratic expense by remembering failed attempts
 - Such scanners are called **maximal munch** scanners

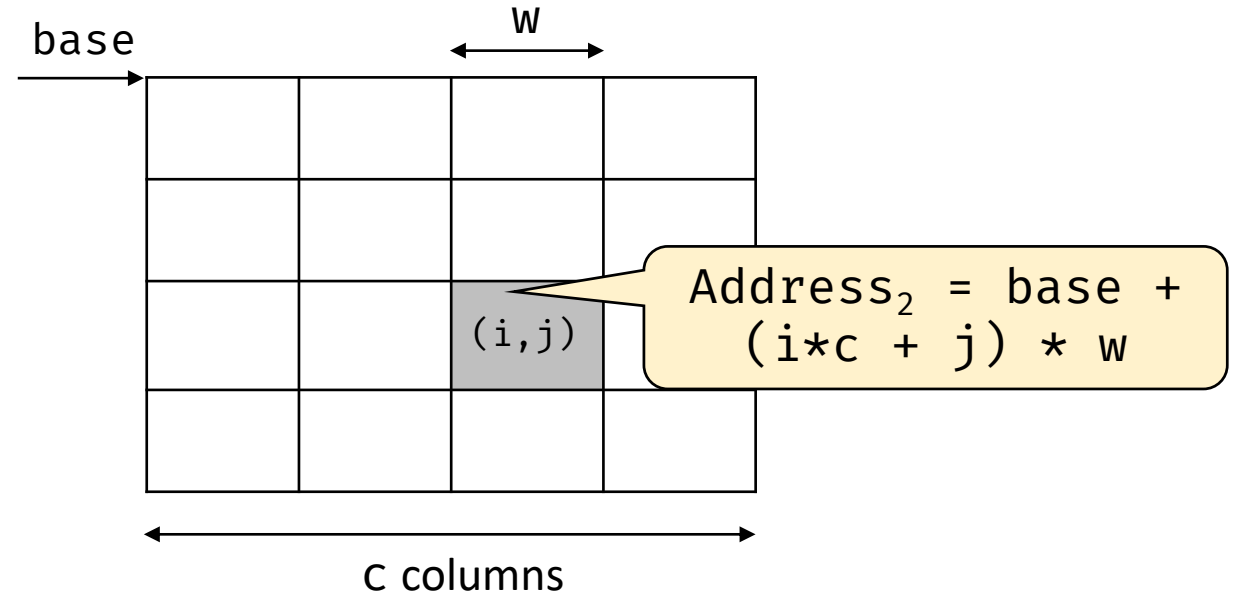
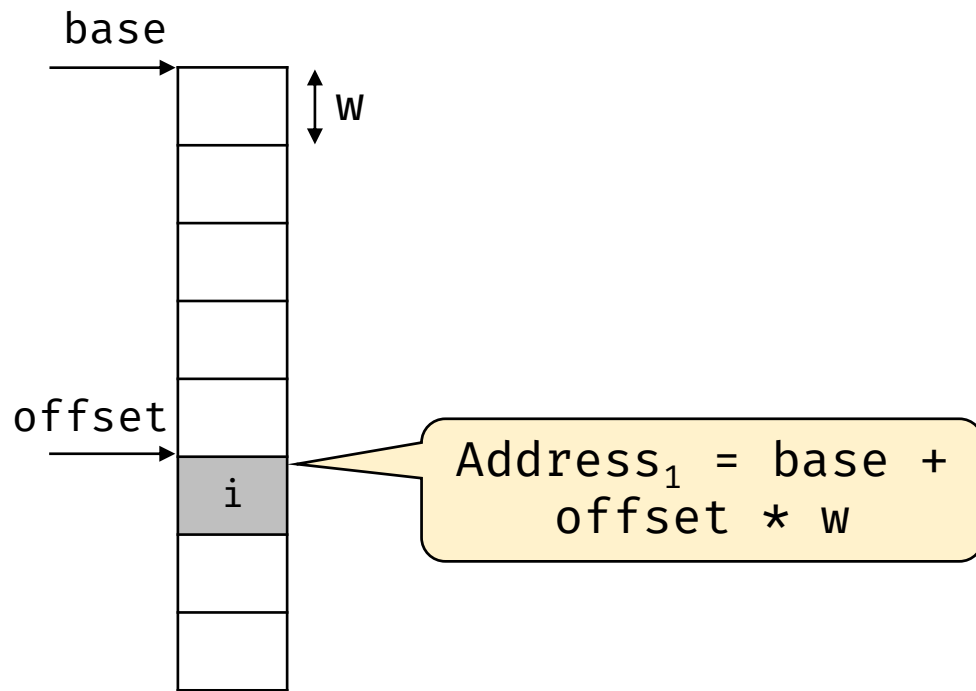
```
inputPos = 0
for each state  $s \in \text{DFA}$ 
  for  $i = 1:|\text{input stream}|$ 
    Failed[state, i] = false
```

Address Excessive Rollbacks

```
state =  $s_0$ ; lexeme = "";  
clear stack; push(<bad, bad>);  
  
while (state  $\neq s_e$ )  
    char = getNextChar()  
    lexeme = lexeme + char  
    inputPos = inputPos + 1  
    if Failed[state, inputPos]  
        break  
    if state  $\in s_A$   
        clear stack  
        push(<state, inputPos>)  
        token = lookup(PATTERN)  
        state =  $\delta$ (state, token)
```

```
// Rollback  
while (state  $\notin s_A$  and state  $\neq$  bad)  
    Failed[state, inputPos] = true  
    <state, inputPos> = pop()  
    truncate lexeme  
    rollback()  
  
if state  $\in s_A$   
    return token  
else  
    return invalid
```

Overhead with Table Lookups



The table-driven scanner performs two address computations and **two load operations** for each character that it processes

Direct-Coded Scanner

```
lexeme = ""; clear stack;  
push(bad); goto s0;
```

```
s0: char = getNextChar()  
    lexeme = char  
    if state ∈ sA  
        clear stack  
    push(s0)  
    if (char == 'r')  
        goto s1  
    else  
        goto se
```

```
s1: char = getNextChar()  
    lexeme = lexeme + char  
    if state ∈ sA  
        clear stack  
    push(s1)  
    if ('0' ≤ char ≤ '9')  
        goto s2  
    else  
        goto se
```


Direct-Coded Scanner

```
s2: char = getNextChar()
      lexeme = lexeme + char
      if state ∈ sA
        clear stack
      push(s2)
      if ('0' ≤ char ≤ '9')
        goto s2
      else
        goto se
```

```
se: while (state ∉ sA and
           state ≠ bad)
      state = pop()
      truncate lexeme
      rollback()
      if state ∈ sA
        return token
      else
        return invalid
```

Hand-Coded Scanner

- Many real-world compilers use hand-coded scanners for further efficiency
 - For e.g., gcc 4.0 uses hand-coded scanners in several of its front ends
- i. Fetching a character one-by-one from I/O is expensive; fetch a number of characters in one go and store in a buffer
- ii. Use double buffering to simplify lookahead and rollback

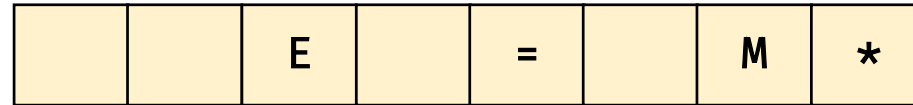
Reading Characters from Input

- A scanner reads the input character by character
 - Reading the input will be very inefficient if it requires a system call for every character read
- Input buffer
 - OS reads a block of data, supplies scanner the required amount, and stores the remaining portion in a buffer called buffer cache
 - In subsequent calls, actual I/O does not take place as long as the data is available in the buffer cache
 - Scanner uses its own buffer since requesting OS for single character is also costly due to context-switching overhead

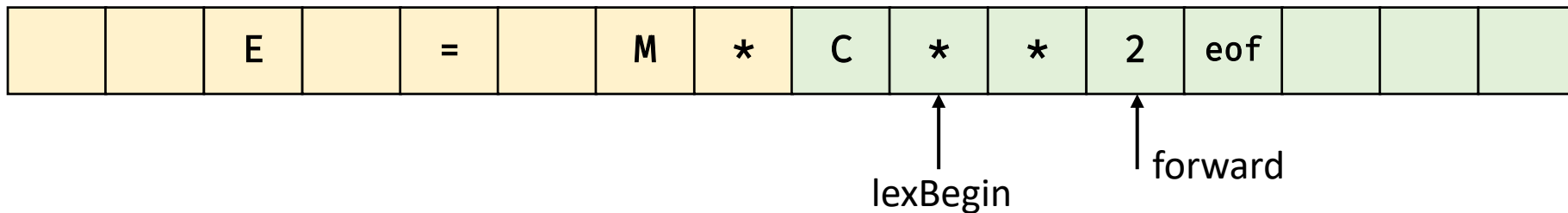
Optimizing Reads from the Buffer

- A buffer at its end may contain an initial portion of a lexeme

E = M*C**2

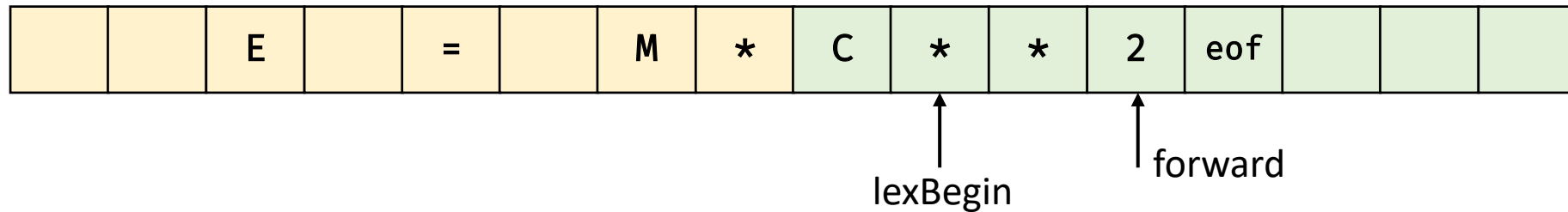


- It creates problem in refilling the buffer, so a two-buffer scheme is used where the two buffers are filled alternatively



Optimizing Reads from the Buffer

- Read from buffer
 - (1) Check for end of buffer, and (2) test the type of the input character
 - If end of buffer, then reload the other buffer

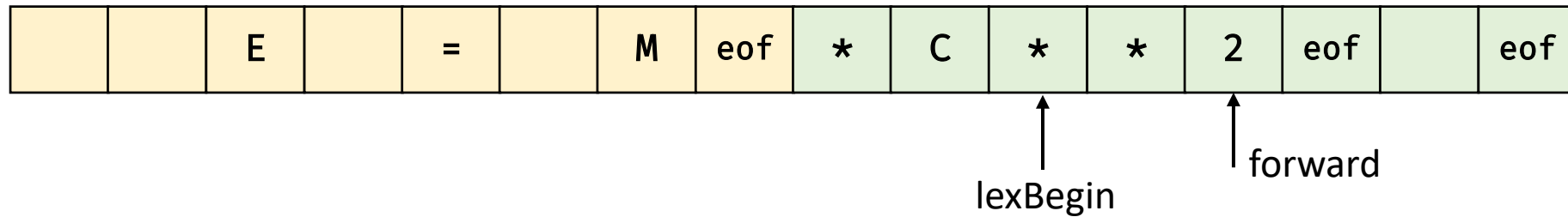


Advance Forward Pointer

```
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 {  
    forward++  
}
```

Optimizing Reads from the Buffer

- A sentinel character (say eof) is placed at the end of buffer to avoid two comparisons



Optimizing Reads from the Buffer

```
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 { // end of input
      break
    }
  ...
  // case for other characters
}
```


Symbol Table

- Data structure that stores information for subsequent phases
- Symbol table interface
 - `insert(s, t)`: save lexeme `s`, token `t`, and return pointer
 - `lookup(s)`: return index of entry for lexeme `s` or \emptyset if `s` is not found

Implementation of Symbol Table

Fixed space for lexemes	Other attributes

←-----→
32 bytes

Fixed amount of space to store lexemes might waste space

Pointer to lexemes	Other attributes

←-----→
4 bytes



Handling Keywords

- Two choices: use separate REs or compare lexemes for ID token
- Consider token DIV and MOD with lexemes `div` and `mod`
- Initialize symbol table with `insert("div", DIV)` and `insert("mod", MOD)` before beginning of scanning
 - Any subsequent insert fails and any subsequent lookup returns the keyword value
 - These lexemes can no longer be used as an identifier

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 3.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 2.