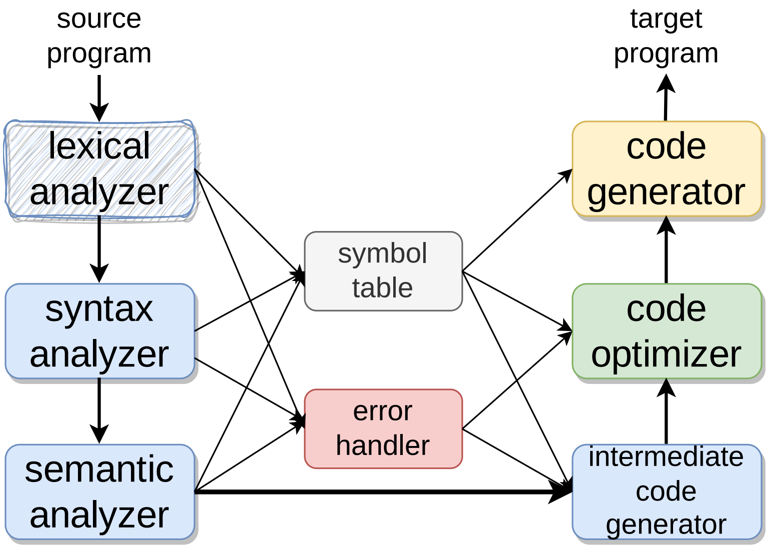
CS 335: Lexical Analysis

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

An Overview of Compilation



Overview of Lexical Analysis

First stage of a three-part front end to help understand the source program

- Processes every character in the input program, so good performance is important
- If a word is valid, then it is assigned to a syntactic category

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

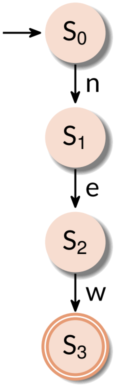
Output: A sequence of tokens along with attributes corresponding to different syntactic categories that are forwarded to the parser for syntax analysis

Functionality:

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

Recognizing Word "new"

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



Formalism for Scanners

Regular expressions, DFAs, and NFAs

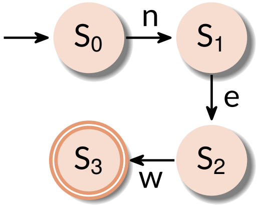
Definitions

- 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 quintuple $(S, \Sigma, \delta, s_0, S_F)$ where
  - 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`"



$$S = (s_0, s_1, s_2, s_3)$$

$$\Sigma = \{n, e, w\}$$

$$\delta = \{s_0 \xrightarrow{n} s_1, s_1 \xrightarrow{e} s_2, s_2 \xrightarrow{w} s_3\}$$

$$s_0 = s_0$$

$$S_F = \{s_3\}$$

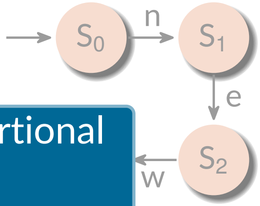
Finite State Automaton

- A **finite state automaton** (FSA) is a quintuple $(S, \Sigma, \delta, s_0, S_F)$ where
  - 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,
  - $\delta(s, c)$ re
    input c,
  - $s_0 \in S$ is
  - $S_F \subseteq S$ is

String is recognized in time proportional to the length of the input

- 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"



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

Implementing an FSA

$$F = (S, \Sigma, \delta, s_0, S_F)$$
$$S = (s_0, s_1, s_2, s_e)$$
$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$\delta = \{s_0 \xrightarrow{0} s_1, s_0 \xrightarrow{1-9} s_2, s_2 \xrightarrow{0-9} s_2, s_1 \xrightarrow{0-9} s_e\}$$
$$s_0 = s_0$$
$$S_F = \{s_1, s_2\}$$

```
ch = getNextChar( )
state = s0
// se is the error state
while (ch != EOF and state != se)
 state = δ(state, ch)
 ch = getNextChar()
if (state ∈ SF)
 report success
else
 report failure
```

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)

- NFA is an FSA that (i) allows transitions on the empty string ϵ and (ii) can have states that have multiple transitions on the same input character
- Ways to simulate an NFA
  - (i) **Always make the correct** nondeterministic choice to follow transitions that lead to accepting state(s) for the input string, if such transitions exist
  - (ii) 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 Regular Expressions (RE)
- The language described by an RE r is called a regular language (denoted by $L(R)$)
- 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, $L(rs) = RS = \{xy \mid x \in R \text{ and } y \in S\}$

Closure r^* is a 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 0s and 1s

$r = (0+1)^*$

$L = \{w \in \{0,1\}^*$ where w has two or three consecutive 1s, but the first and the second are not consecutive

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

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

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

Unsigned real numbers with exponents

$r = (0|[1\ldots9][0\ldots9]^*)\ (.[0\ldots9]^*|\epsilon)\ E\ (+|-|\epsilon)\ (0|[1\ldots9][0\ldots9]^*)$

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

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

More on 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
$r(s\|t) = rs\|rt$; $(s\|t)r = sr\|st$	Concatenation distributes over |
$r\epsilon = \epsilon r = r$	ϵ is the identity of concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$(r^*)^* = r^*$	* is idempotent

Regular Definitions

- Regular Definition is a sequence of definitions of the form

$$
\begin{aligned}
d_1 &\longrightarrow r_1 \\
d_2 &\longrightarrow r_2 \\
\ldots \\
d_n &\longrightarrow r_n
\end{aligned}
$$

where

- each d_i is a new symbol (i.e., name) not already in Σ,
- each r_i is a RE over the symbols $\Sigma \cup \{d_1, d_2, \ldots d_{i-1}\}$

Regular definition for unsigned numbers (e.g., 5280, 0.01234, 6.336E4, or 1.89E-4)

digit	=	0|1|2|3|4|5|6|7|8|9
digits	=	$digit\ digit^*$
opt_frac	=	. *digits* | ϵ
opt_exp	=	(E(+|-|ϵ) *digits*)|ϵ
unsigned_num	=	*digits opt_frac opt_exp*

Extensions of Regular Expressions

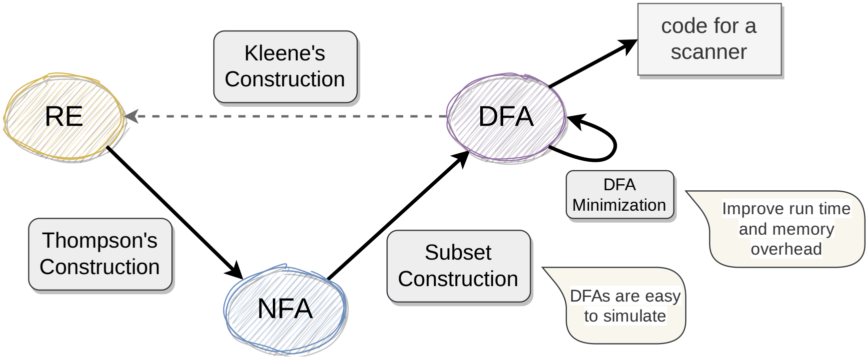
- "." is any character other than "\n"
- $[xyz]$ is $x|y|z$
- $[abg{-}pT{-}Y]$ is any character $a, b, g, \ldots p, T, \ldots, Y$
- $[\hat{}G{-}Q]$ is not any one of $G, H, \ldots Q$
- $r+$ is one or more occurrences of r
- $r?$ is zero or one r

Regular definition for unsigned numbers (e.g., 5280, 0.01234, 6.336E4, or 1.89E-4)

digit	=	[0-9]
digits	=	*digit*+
unsigned_num	=	$digits\ (.digits)?\ (E[+-]?digits)?$

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
- ...



NFA to DFA: Subset Construction

NFA

$(N, \Sigma, \delta_N, n_0, N_A)$

DFA

$(D, \Sigma, \delta_D, d_0, D_A)$

Subset Construction

```
q0 = ε-closure({s0})
Q = q0
Worklist = {q0}
while (Worklist ≠ ϕ) 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

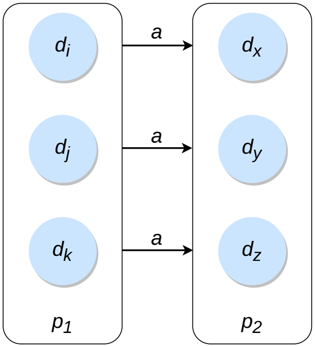
```
for each state n ∈ N do
  E(n) = {n}
Worklist = N
while (Worklist ≠ ϕ) do
  remove n from Worklist
  t = {n} ∪ ⋃_{n →ε p ∈ δN} E(p)
  if t ≠ E(n)
    E(n) = t
    Worklist = Worklist ∪ {m | m →ε n ∈ δN}
```

DFA to Minimal DFA: Hopcroft's Algorithm

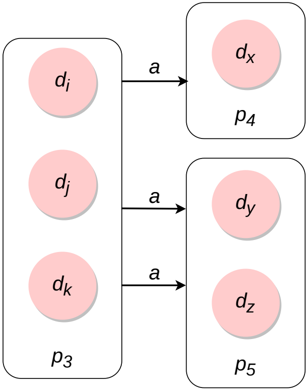
A DFA from Subset construction can have a large number of states

+ Does not increase the time needed to scan a string
– Increases the space requirement of the scanner in memory
  - Frequent accesses to main memory will slow the scanner
  - A smaller scanner has better chances of fitting in the processor cache

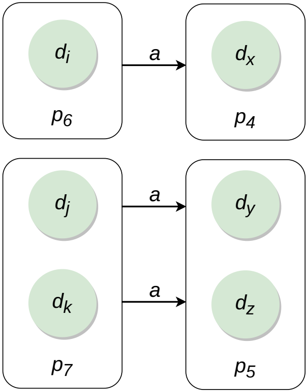
Identifying Behavioral Equivalence



a does not split p_1, equivalent behavior



a splits p_3, different behavior for the states



Partitions after splitting on a

DFA to Minimal DFA: Hopcroft's Algorithm

Minimization

```
T = {D_A, {D − D_A}}
P = ϕ
while (P ≠ T) do
  P = T
  T = ϕ
  for each set p ∈ P do
    T = T ∪ Split(p)
```

Split (S)

```
for each c ∈ Σ do
  if c splits S into s_1 and s_2
    return {s_1, s_2}
return S
```

Realizing Scanners

Tokens

Definition

Tokens are a string of characters that logically belong together in a syntactic category

- Example of tokens in programming languages: Keywords, operators, identifiers (names), constants, literal strings, and punctuation symbols (parentheses, brackets, commas, semicolons, and colons)

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

- Sentences consist of a string of tokens (e.g., float, identifier, assign, minus, intnum, and semicolon)
- Tokens are treated as terminal symbols of the grammar specifying the source language
- Tokens may have optional attributes

Patterns and Lexemes

Pattern is the rule describing the set of strings for which the same token is produced

Lexeme is the sequence of characters matched by a pattern to form the corresponding token

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

- Patterns are float, letter(letter|digit|_)*, =, -, digit+, and ;
- Lexemes are “float”, “abs_zero”, “=”, “-”, “273”, and “;”

Attributes of Tokens

Definition

An **attribute** of a token is a value that the scanner extracts from the corresponding lexeme and supplies to the syntax analyzer

Example attributes for tokens

- identifier: the lexeme of the token, or a pointer into the symbol table data structure where the lexeme is stored
- intnum: the value of the integer (similarly for floatnum)
- Type of the identifier and the location where first found

The exact set of attributes is dependent on the compiler designer

Role of a Lexical Analyzer

- Identify tokens and corresponding lexemes
- Construct constants
  - 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 is not a keyword
- Discard whatever does not contribute to parsing (e.g., white spaces (blanks, tabs, newlines) and comments)

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

Specifying and Recognizing Patterns and Tokens

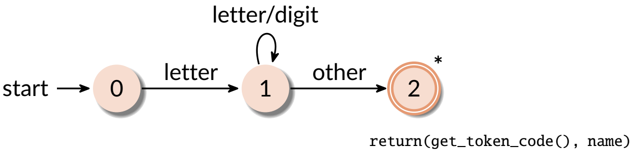
- Patterns are denoted with REs and recognized with FSAs
- Regular definitions are popular for specifying tokens
- Transition diagrams, a variant of FSAs, are used to implement regular definitions and to recognize tokens
  - Usually used to model LA before translating them to executable programs

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)

Example of Transition Diagram for Identifiers and Reserved Words

$$letter = [a - zA - Z]$$
$$digit = [0 - 9]$$
$$identifier = letter\,(letter|digit)^*$$



- * indicates a retraction state
- `get_token_code()` searches the symbol 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

Sample Specification

Grammar Specification

$$
\begin{aligned}
\textit{stmt} &\rightarrow \textbf{if } \textit{expr } \textbf{then } \textit{stmt} \\
&\mid \textbf{if } \textit{expr } \textbf{then } \textit{stmt } \textbf{else } \textit{stmt} \\
&\mid \epsilon \\
\textit{expr} &\rightarrow \textit{term } \textbf{relop } \textit{term} \\
&\mid \textit{term} \\
\textit{term} &\rightarrow \textbf{id} \\
&\mid \textbf{number}
\end{aligned}
$$

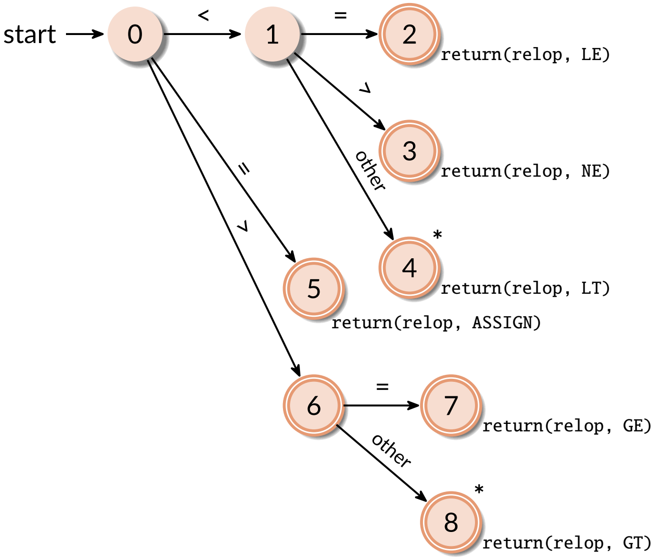
Lexical Specification

$$
\begin{aligned}
\textit{digit} &\rightarrow [0-9] \\
\textit{digits} &\rightarrow \textit{digit}^{+} \\
\textit{number} &\rightarrow \textit{digits}(.\textit{digits})?(E[+-]?\textit{digits})? \\
\textit{letter} &\rightarrow [A-Za-z] \\
\textit{id} &\rightarrow \textit{letter}(\textit{letter}|\textit{digit})^{*} \\
\textit{if} &\rightarrow \textbf{if} \\
\textit{then} &\rightarrow \textbf{then} \\
\textit{else} &\rightarrow \textbf{else} \\
\textit{relop} &\rightarrow < \mid > \mid \leq \mid \geq \mid = \mid <> \\
\textit{ws} &\rightarrow (\text{blank}|\text{tab}|\text{newline})^{+}
\end{aligned}
$$

Tokens, Lexemes, and Attributes for the Sample Specification

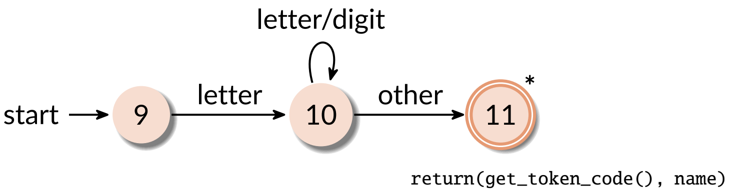
Lexemes	Token Name	Attribute Value
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to symbol table entry
Any *number*	**number**	Pointer to symbol table entry
$<$	**relop**	LT
$\leq$	**relop**	LE
$=$	**relop**	ASSIGN
$>$	**relop**	GT
$\geq$	**relop**	GE
$<>$	**relop**	NE

Transition Diagram for **relop**

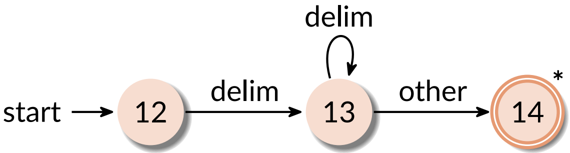


Transition Diagrams for IDs and Keywords

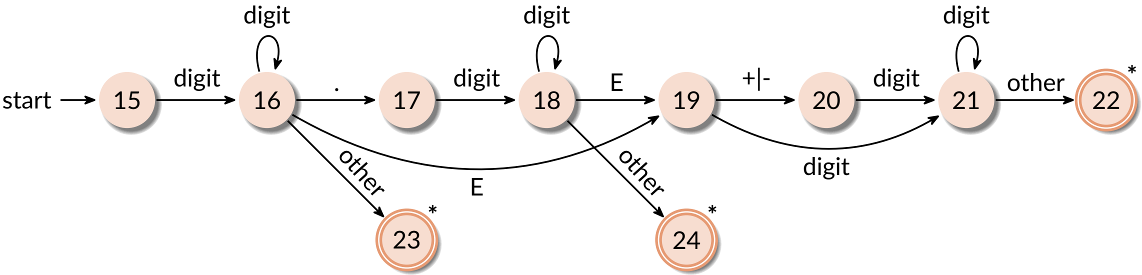
IDs and Keywords



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
  - 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 utilize 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

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

- PL/I declaration: `DECLARE(`$\texttt{arg}_1$`,`$\texttt{arg}_2$`,...,`$\texttt{arg}_n$`)`
- Cannot tell whether `DECLARE` is a keyword with variable definitions or is a procedure with arguments until after “`)`”

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
```

- PL/I declaration: `DECLARE(arg`$_1$`,arg`$_2$`,...,arg`$_n$`)`
- 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

Consider C++ syntax

- Template syntax: `Foo<Bar>`
- Stream syntax: `cin>>var;`
- Nested templates: `Foo<Bar<Bazz>>`

Can these challenges be resolved by lexical analyzers alone?

No, in some cases parser needs to help

Challenges in Lexical Analysis

- Consider a fixed-format language like Fortran
  - 80 columns per line
  - Columns 1-5 represent the statement number/label, column 6 denotes the continuation mark, columns 7-72 denote program statements, and columns 73-80 are ignored (used for other purposes)
  - Letter C in Column 1 means the current line is a comment
  - Some keywords are context-dependent in fixed-format Fortran,
  - Blanks are not always significant in Fortran and can appear amid identifiers
    - Variable "`counter`" is same as "`count er`"
    - Blanks are important only in literal strings
- In the statement `DO 10 I = 10.86`, `DO10I` is an identifier, and `DO` is not a keyword
- But in the statement `DO 10 I = 10,86`, `DO` is a keyword
- Reading from left to right, one cannot distinguish between the two until the "`,`" or "`.`" is reached
  - Requires lookahead 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 (e.g., “stress”)
- “are” is a verb, “art” is a noun, but “arz” is undefined

Grammars are rigorously specified to provide meaning

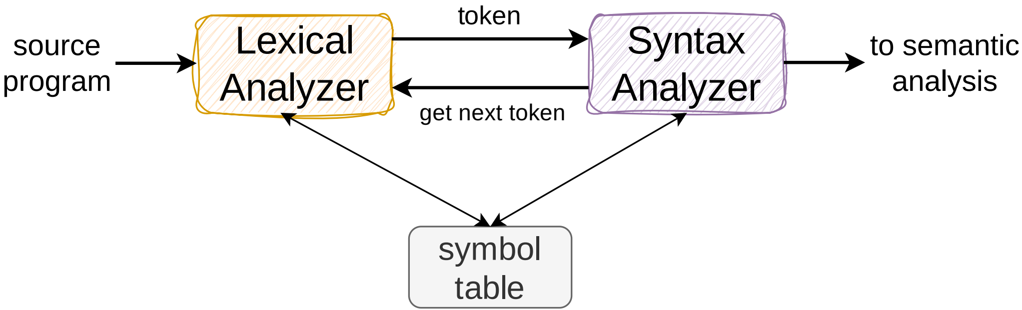
- Words in a programming language are always lexically specified
- Any string in `(1...9)(0...9)*` is a positive integer

Advantages with Lexical Analysis as a Separate Phase

+ **Simplifies** the compiler design: I/O issues are limited to only the lexical analyzer, leading to better **portability**
+ 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
+ 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 scanner 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
- Other recovery strategies are possible
  - 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
  - grep is an acronym for "global regular expression print"
- Search tools in editors
- Word-processing tools

Implementing Scanners

Implementing Scanners

1. Specify REs for each syntactic category in the programming language
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

High-Level Idea in a Scanner

- Read input characters one by one
- Look up the transition in the corresponding DFA based on the current state and the input character
- Switch to the new state
- Check for termination conditions, i.e., accept and error
- Repeat

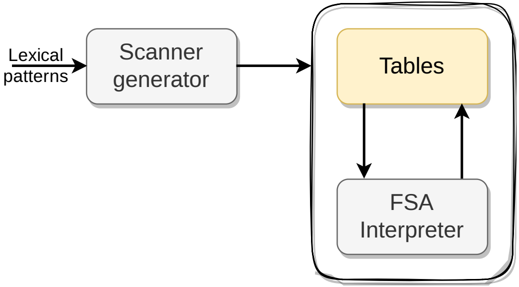
Speed is paramount for scanning

Processes every character from a possibly large input source program

Types of scanner implementations: table-driven, direct-coded, and hand-coded

Asymptotic complexity is the same but differs in run-time costs

Table-Driven Scanner



Consider a pattern specifying registers (e.g., r1 and r27)

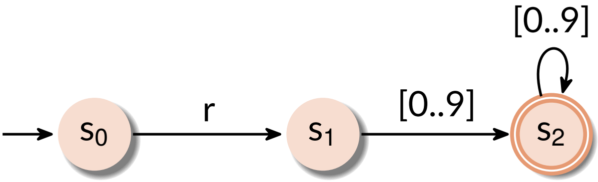


Table-Driven Scanner

```
state = s0; lexeme = "";
clear stack; push(bad);
// Model the DFA
while (state ≠ se)
  ch = getNextChar()
  lexeme = lexeme + ch
  if state ∈ sA
    clear stack
  push(state)
  token = lookup(PATTERN)
  state = δ(state, token
while (state ≠ sA and state ≠ bad)
  state = pop()
  truncate lexeme
  rollback()
if state ∈ sA
  return token
else
  return invalid
```

A single δ table mapping states in S to characters in Σ will be too large

r	**0,1,..,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 $ab|(ab)^{*}c$, and the 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 ∈ DFA
  for i = 1:len(input stream)
    Failed[state, i] = false
```

Address Excessive Rollbacks

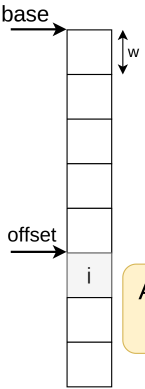
```
state = s0; lexeme = "";
clear stack; push(<bad, bad>);

while (state ≠ se)
  ch = getNextChar()
  lexeme = lexeme + ch
  inputPos = inputPos + 1
  if Failed[state, inputPos]
    break
  if state ∈ sA
    clear stack
  push(<state, inputPos>)
  token = lookup(PATTERN)
  state = δ(state, token)
```

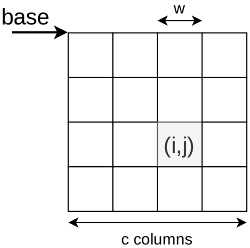
```
// Rollback
while (state ∉ sA and state ≠ bad)
  Failed[state,inputPos] = true
  <state,inputPos> = pop()
  truncate lexeme
  rollback()

if state ∈ sA
  return token
else
  return invalid
```

Overhead with Table Lookups



Address$_i$ = base + offset * w



Address$_{ij}$ = base + (i*c+j) * w

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

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

```
s2: ch = getNextChar()
    lexeme = lexeme + ch
    if state ∈ sA
      clear stack
    push(s2)
    if ‘0’ ≤ ch ≤ ‘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
  - 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

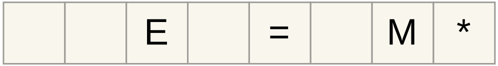
Reading Characters from Input

- 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

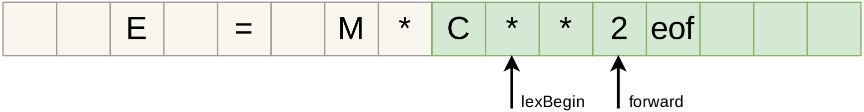
Optimizing Reads from the Buffer

E = M * C ** 2

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



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



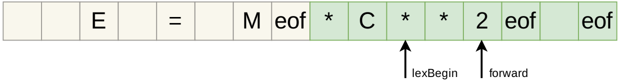
Advance Forward Pointer

- Read from the 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

```
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 the buffer to avoid two comparisons



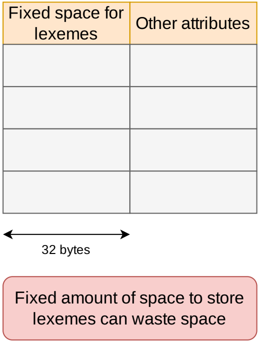

```
switch (*forward++) {
  case EOF:
    if (forward is at end of first buffer) {
      reload second buffer
      forward = beginning of second buffer
    } else if (forward is at end of second buffer) {
      reload first buffer
      forward = beginning of first buffer
    } else { // end of input
      break
    }
    ...
  // case for other characters
}
```

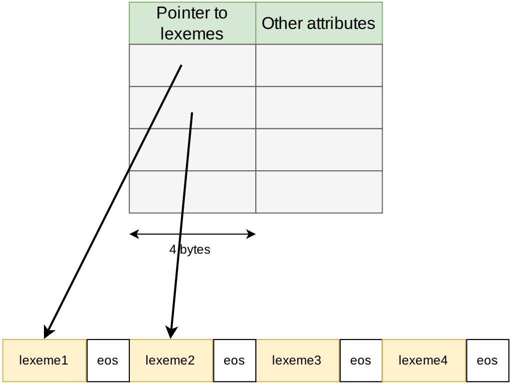
Symbol Table

Symbol Table

Symbol Table is a 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 0 if `s` is not found





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 the 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. Chapter 3, 2nd edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Chapter 2, 2nd edition, Morgan Kaufmann.