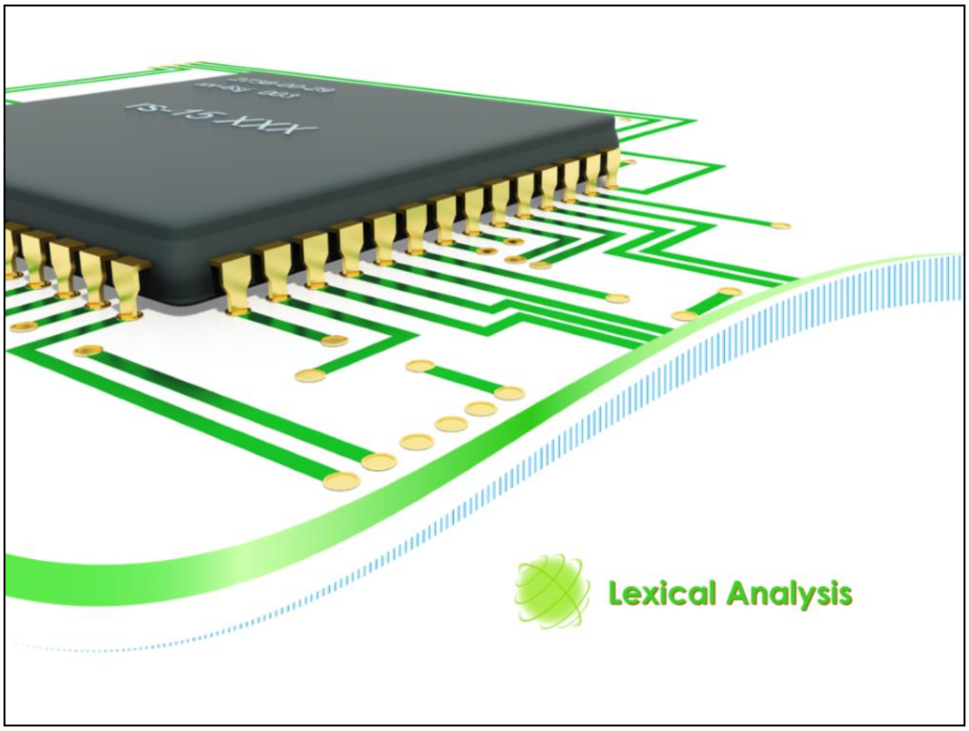
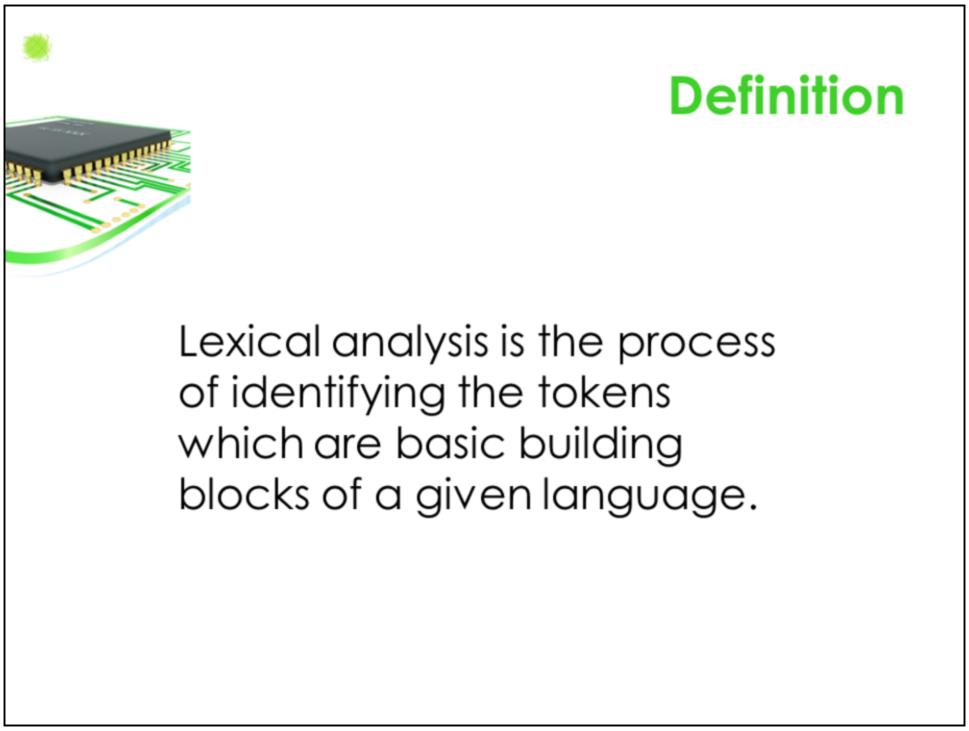


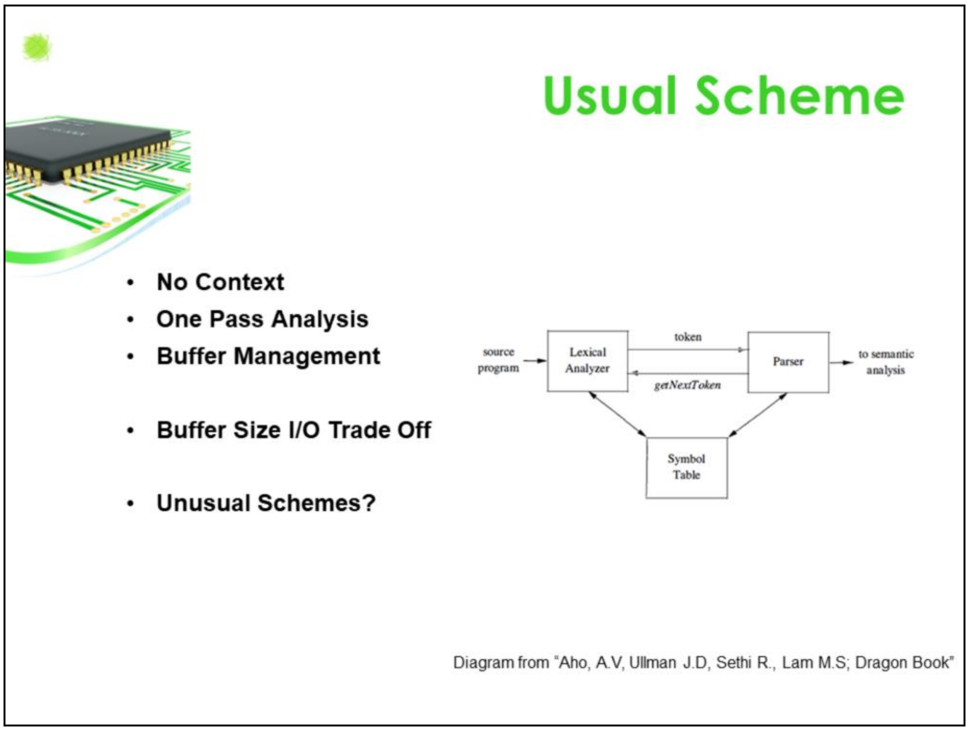
Lexical Analysis



Definition

Lexical analysis is the process of identifying the tokens which are basic building blocks of a given language.

Lexical analysis is the process of identifying the tokens which are basic building blocks of a given language. Basically, a lexical analyzer converts the series of the alphabetical elements to a series of words that are recognizable by the subsequent steps of the language processing. In this sense, the lexical analyzer can be addressed as the tentacles of the language processor on its input.



Usual Scheme

- No Context
- One Pass Analysis
- Buffer Management
- Buffer Size I/O Trade Off
- Unusual Schemes?

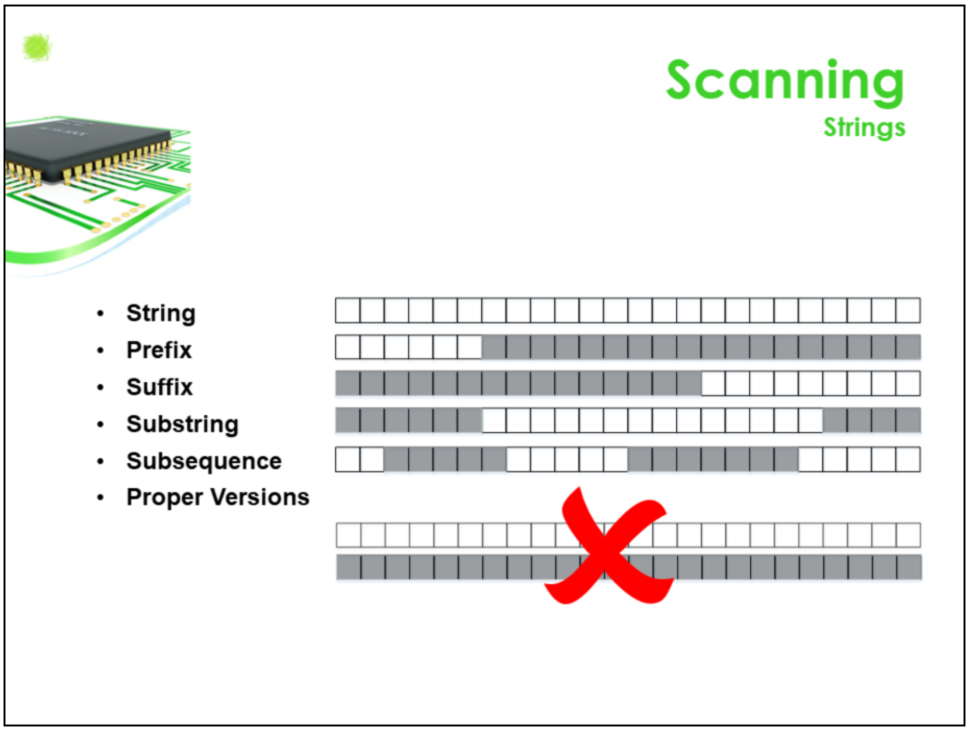
source program
Lexical Analyzer
token
getNextToken
Parser
to semantic analysis
Symbol Table

Diagram from "Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book"

The interaction between the lexical analyzer and the parser is shown as above in the infamous Dragon book. This diagram is valid for the widely accepted scheme that processes input in a single pass. According this scheme, lexical analysis phase is oblivious to the context which it is acting in. The advantage of this approach is that the source is received through a stream which does not require substantial buffer size. The whole source program does not need to be held in the process memory space. However, input buffers and techniques to manage these buffers must be integrated to the lexical analysis in a way to allow various needs such as alternate switching, and look ahead.

Buffer management is a highly implementation dependent detail. Use of alternate buffers as a character queue is a probable solution but it is not absolutely necessary.

Unusual cases can be constructed as context dependent, multi-pass analyses, whole source buffer are also applicable especially for the cases where top-down parsing is applied. Alternative approaches can be chosen based on resource availability, complexity, context dependence, and similar.



Scanning

Strings

- String
- Prefix
- Suffix
- Substring
- Subsequence
- Proper Versions

These terms are introduced here because the lexical analysis deals with strings. These definitions come handy while communicating the parts and properties of the strings.

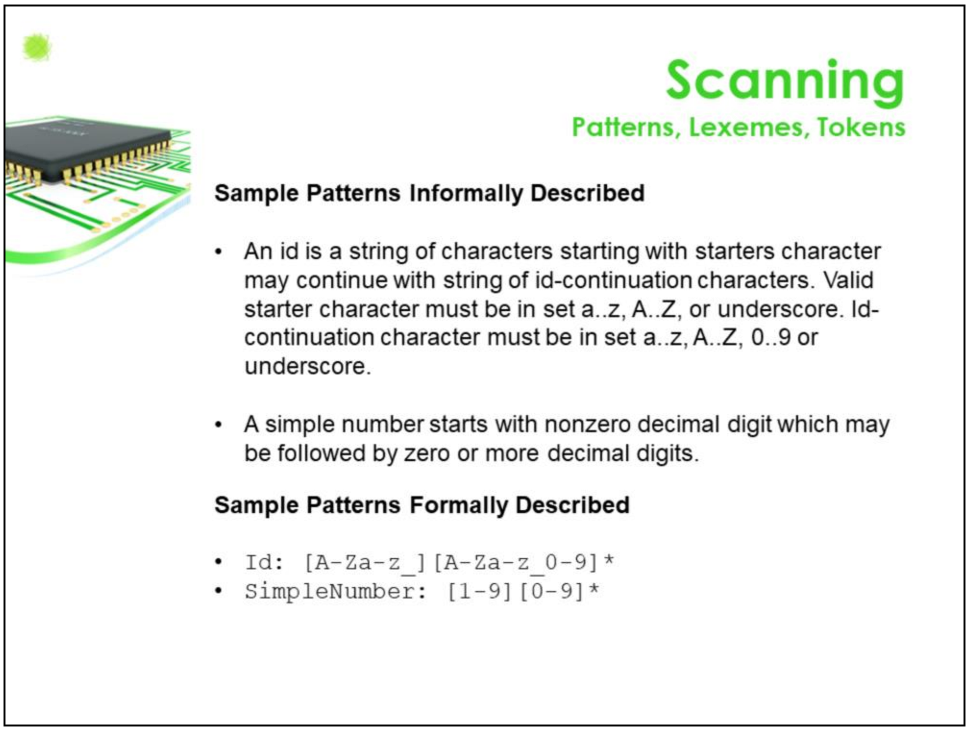
A prefix of string S is any string obtained by removing zero or more symbols from the end of S. For example, ban, banana, and ε are prefixes of banana.

A suffix of string s is any string obtained by removing zero or more symbols from the beginning of S. For example, nana, banana, and ε are suffixes of banana.

A substring of s is obtained by deleting any prefix and any suffix from S. For instance, banana, nan, and ε are substrings of banana.

The proper prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of S that are not ε or not equal to S itself.

A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana.



Scanning

Patterns, Lexemes, Tokens

Sample Patterns Informally Described

- An id is a string of characters starting with starters character may continue with string of id-continuation characters. Valid starter character must be in set a..z, A..Z, or underscore. Id-continuation character must be in set a..z, A..Z, 0..9 or underscore.
- A simple number starts with nonzero decimal digit which may be followed by zero or more decimal digits.

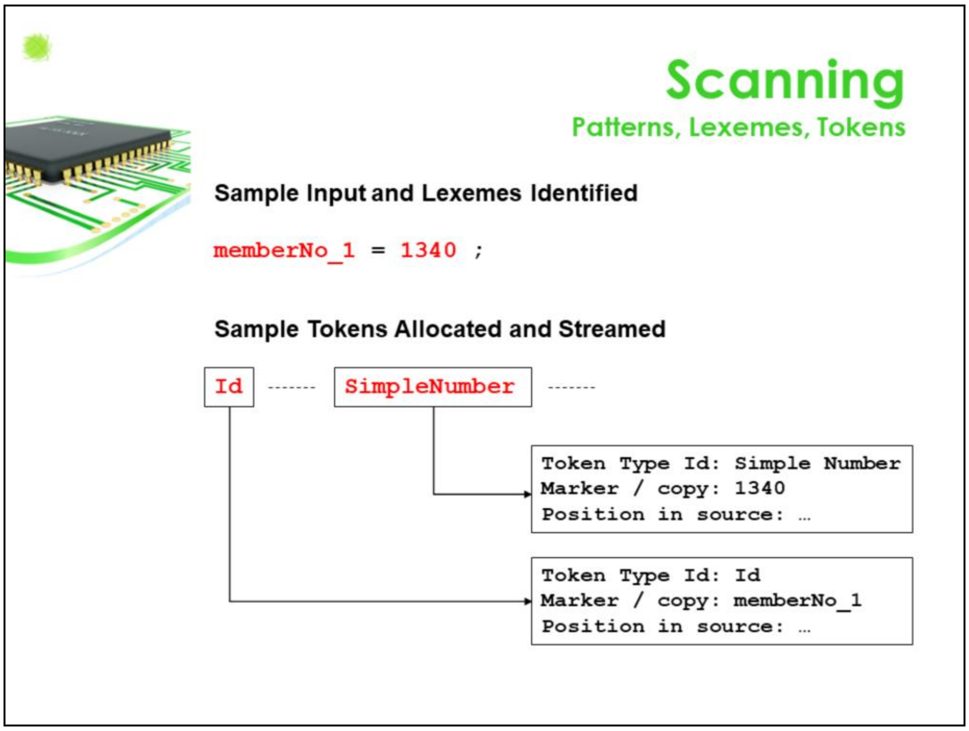
Sample Patterns Formally Described

- `Id: [A-Za-z_][A-Za-z_0-9]*`
- `SimpleNumber: [1-9][0-9]*`

A lexical analyzer is built with the capacity to recognize certain set of patterns from its input stream. In a typical language processing setting, the patterns are fixed for the reserved words, separators and delimiters; variable but rule dependent for the literal specifications like constants of various type domains. It is usually safe to state that the reserved word are limited in number but the literals are virtually unlimited, keeping the system imposed limits in mind.

Patterns are string formations that can be recognized at the lexical analysis phase. The patterns are straightforward as in the case of reserved words or complicated as in the well known case of floating point number literals. It is a common approach to use formal representations for the patterns with limited but enough flexibility.

A string that fits one of the patterns recognizable by the lexical analyzer is called a lexeme. A token is a lexeme attached with context dependent data (a symbol id, position, and similar) that can be referenced from broader contexts, which are established by the subsequent stages of language processing.



Scanning

Patterns, Lexemes, Tokens

Sample Input and Lexemes Identified

```
memberNo_1 = 1340 ;
```

Sample Tokens Allocated and Streamed

Id

SimpleNumber

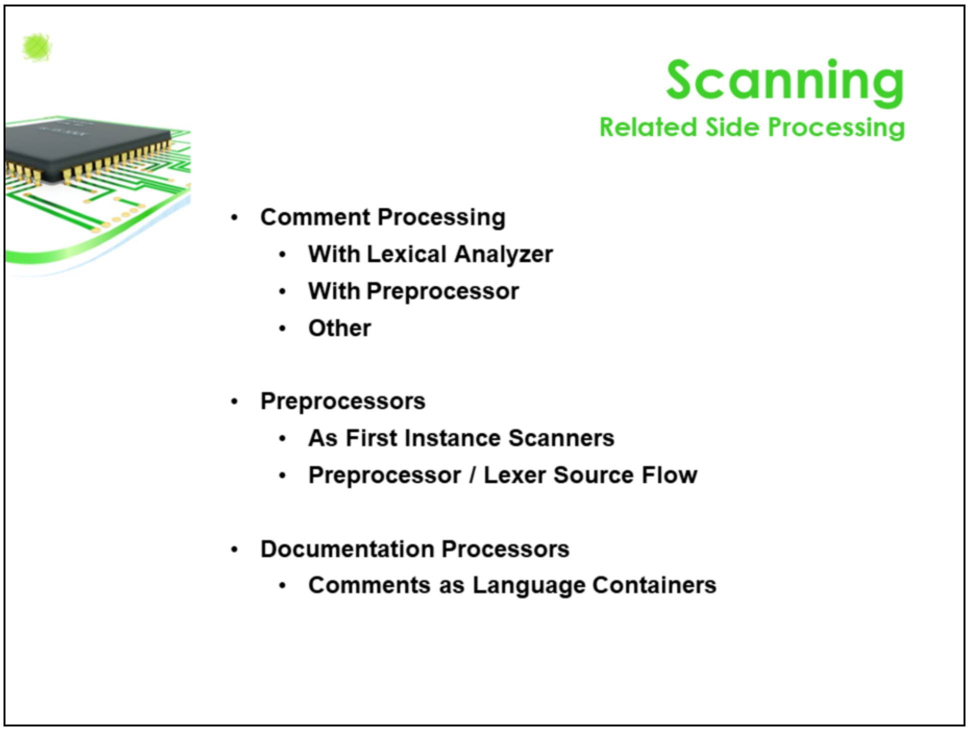
```
Token Type Id: Simple Number
Marker / copy: 1340
Position in source: …
```

```
Token Type Id: Id
Marker / copy: memberNo_1
Position in source: …
```

A lexical analyzer is built with the capacity to recognize certain set of patterns from its input stream. In a typical language processing setting, the patterns are fixed for the reserved words, separators and delimiters; variable but rule dependent for the literal specifications like constants of various type domains. It is usually safe to state that the reserved word are limited in number but the literals are virtually unlimited, keeping the system imposed limits in mind.

Patterns are string formations that can be recognized at the lexical analysis phase. The patterns are straightforward as in the case of reserved words or complicated as in the well known case of floating point number literals. It is a common approach to use formal representations for the patterns with limited but enough flexibility.

A string that fits one of the patterns recognizable by the lexical analyzer is called a lexeme. A token is a lexeme attached with context dependent data (a symbol id, position, and similar) that can be referenced from broader contexts, which are established by the subsequent stages of language processing.



Scanning

Related Side Processing

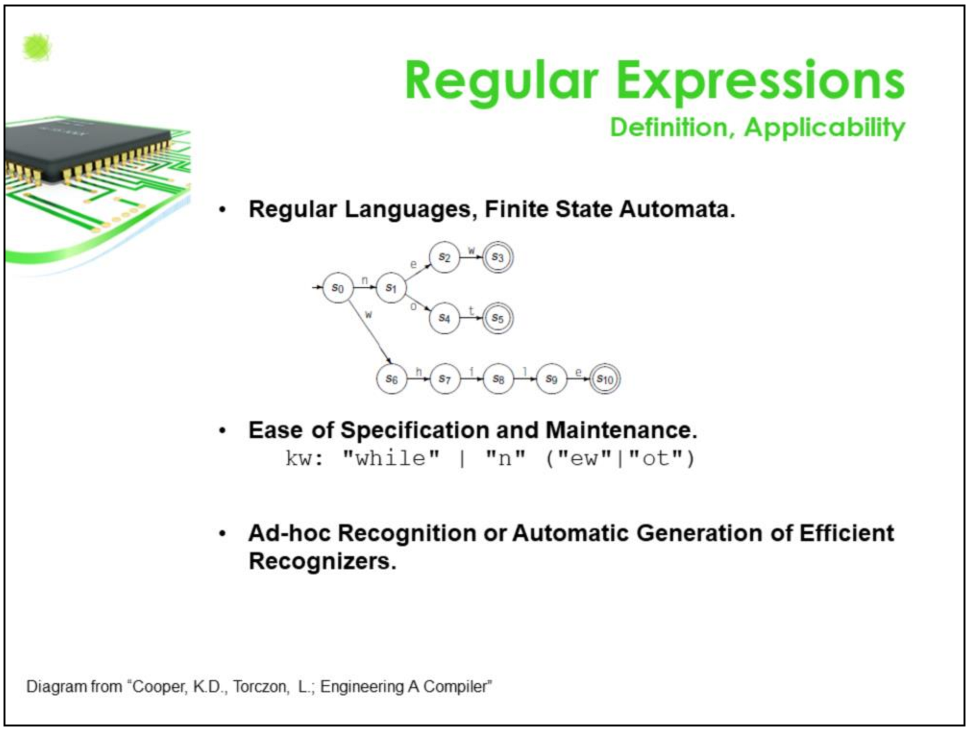
- Comment Processing
  - With Lexical Analyzer
  - With Preprocessor
  - Other
- Preprocessors
  - As First Instance Scanners
  - Preprocessor / Lexer Source Flow
- Documentation Processors
  - Comments as Language Containers

Before proceeding to the details on lexical analysis, comments are worth noting for their specific handling and processing.

In some applications, the scanners may be used to skip the parts of the input such as comment sections in the source. In such cases, the comment sections are not converted to the tokens and are not fed into the parsing stage.

Preprocessors can be placed before lexical analyzers as a component having effect over the whole language processing. In such cases the preprocessor handles delivery of critical contents such as comments, symbols, code location data, and more depending on requirements.

A single source code can be examined by the language processors other than the compiler. Preprocessors, as a prominent example, handle the commands, enable macros, conditional compilation, and file inclusion for C and C++. Further to this, the comments are fed to documentation processors such as Javadoc, JSDoc, DoxyGen, PHPDoc which are the tools named explicitly. In some language implementations such as C# and Swift, specially formed comments are presented as a source level documentation standard with no explicit reference to documentation processor. These tools are useful for generation of API documentation with the convenience of interweaving documentation in proximity of language constructs such as method / function names, parameters, variables, and so on.



Regular Expressions

Definition, Applicability

- Regular Languages, Finite State Automata.

$$\begin{tikzcd}
 & & s_2 \arrow[r, "w"] & s_3 & & \\
\arrow[r] & s_0 \arrow[r, "n"] \arrow[drr, "w"] & s_1 \arrow[u, "e"] \arrow[d, "o"] & & & \\
 & & s_4 \arrow[r, "t"] & s_5 & & \\
 & & & s_6 \arrow[r, "h"] & s_7 \arrow[r, "i"] & s_8 \arrow[r, "l"] & s_9 \arrow[r, "e"] & s_{10}
\end{tikzcd}$$

- Ease of Specification and Maintenance.

```
kw: "while" | "n" ("ew"|"ot")
```

- Ad-hoc Recognition or Automatic Generation of Efficient Recognizers.

Diagram from "Cooper, K.D., Torczon, L.; Engineering A Compiler"

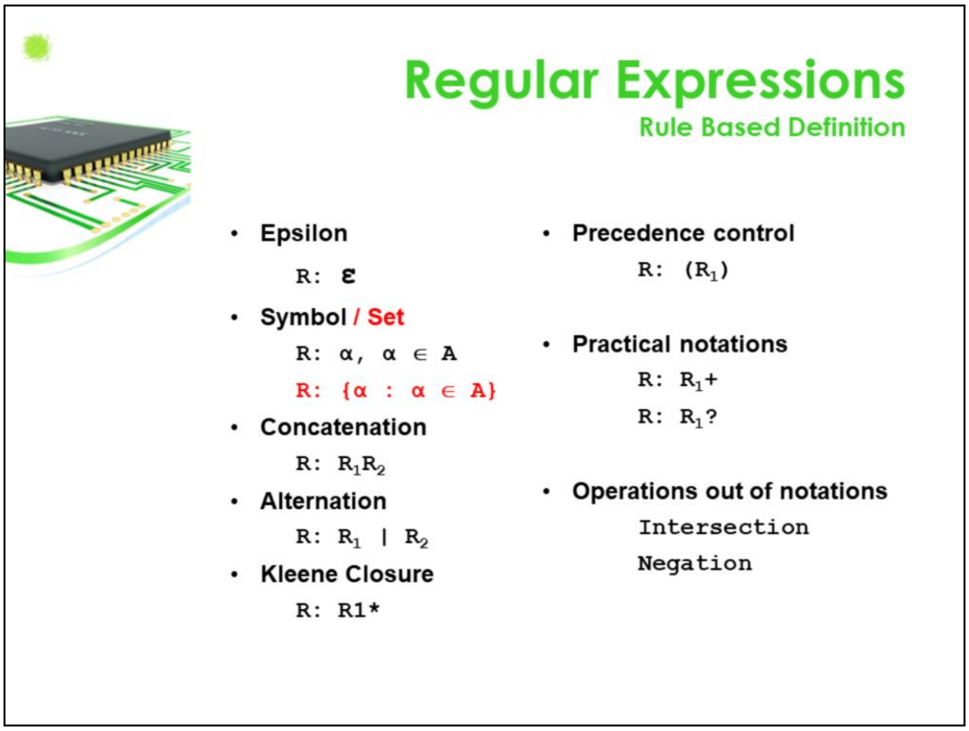
A regular expression is representative of a grammar that generates regular languages. These regular languages can be decided by a finite state automaton.

Regular expressions are string constructs with the capacity of recognizing certain character sequences. As we will see later, a regular expression corresponds to a deterministic finite automaton that will enable efficient recognition of the pattern that can be found in the source input.

Use of regular expressions to describe the patterns that will be recognized by the lexical analyzer is a common approach with the benefits as noted below.

Ease of specification and maintenance.

Ad-hoc recognition or automatic generation of efficient recognizers.



Regular Expressions

Rule Based Definition

- Epsilon

  R: ε
- Symbol / Set

  R: α, α ∈ A

  R: {α : α ∈ A}
- Concatenation

  R: R_1R_2
- Alternation

  R: $R_1 \mid R_2$
- Kleene Closure

  R: R1*
- Precedence control

  R: (R_1)
- Practical notations

  R: R_1+

  R: $R_1?$
- Operations out of notations

  Intersection

  Negation

Regular expressions are formalisms that can be introduced by using following rules.

ε denotes empty string and is a valid regular expression that recognizes an empty string.

If α is an element of the Alphabet A then α forms a regular expression r that recognizes α.

If r_1 and r_2 are regular expressions than r_1r_2 is a regular expression r_3 that recognizes the r_1 followed by r_2. This is known as concatenation.

If r_1 and r_2 are regular expression then $r_1 \mid r_2$ is a regular expression r_3 that recognizes either r_1 or r_2. This is known as alternation.

If r_1 is regular expression then r_1* is a regular expression r_2 that recognizes zero or more occurrences of r_1. This is known as Kleene Closure or simply closure.

Last three rules imply that regular expressions are closed under concatenation, alternation, and closure.

Precedence control.

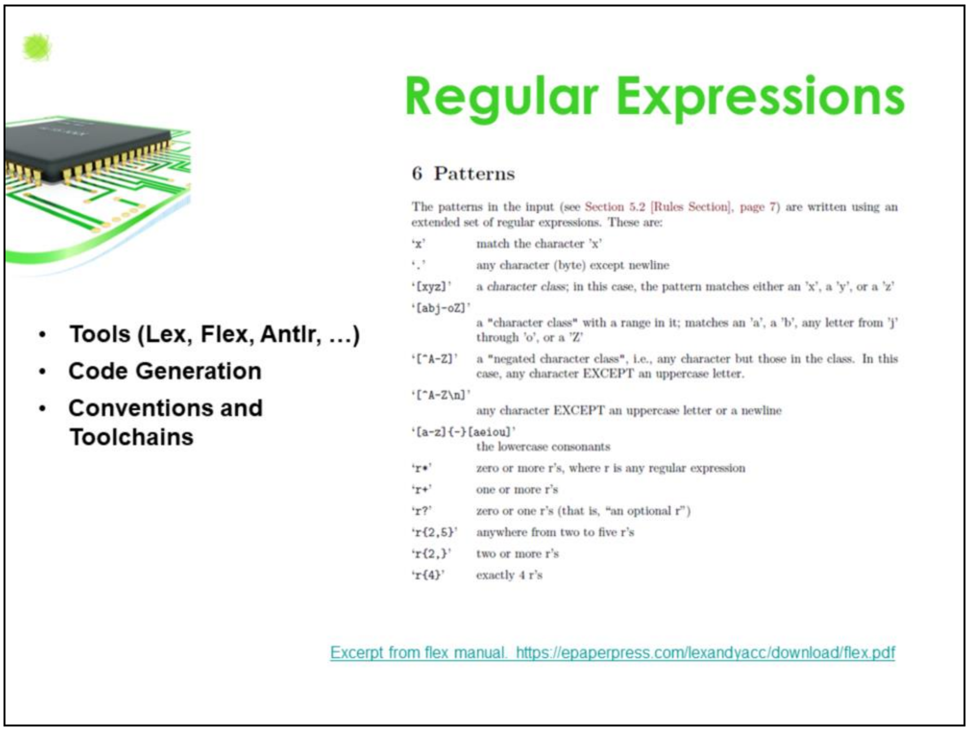
The order of precedence from the highest to the lowest is “closure”, “concatenation”, and “alternation”. Parentheses can be used to establish control over precedence. If r_1 is a regular expression, (r_1) is a regular expression r_2 that recognizes the same languages that are recognized by r_1.

Practical closure notations.

Additional closures are possible for practical use. If r_1 is a regular expression than r_1^i is a regular expression r_2 which recognizes r_1 i times. This is known as finite closure. If r_1 is a regular expression than r_1+ is a regular expression r_2 which recognizes r_1 one or more times. Finally if r_1 is a regular expression $r_1?$ is a regular expression r_2 which recognizes r_1 zero or more times.

Operations out of notations

Intersection and negation are also examined when regular languages are concerned, notations for these to operations will not be addressed in this course. However, it is good to keep in mind that, negation is inverting the “accepting” property of each node in a DFA, and intersection can algebraically be rewritten in terms of union (alternation) and negation.



Regular Expressions

- Tools (Lex, Flex, Antlr, …)
- Code Generation
- Conventions and Toolchains

6 Patterns

The patterns in the input (see Section 5.2 [Rules Section], page 7) are written using an extended set of regular expressions. These are:

`'x'`	match the character 'x'
`'.'`	any character (byte) except newline
`'[xyz]'`	a *character class*; in this case, the pattern matches either an 'x', a 'y', or a 'z'
`'[abj-oZ]'`	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
`'[^A-Z]'`	a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
`'[^A-Z\n]'`	any character EXCEPT an uppercase letter or a newline
`'[a-z]{-}[aeiou]'`	the lowercase consonants
`'r*'`	zero or more r's, where r is any regular expression
`'r+'`	one or more r's
`'r?'`	zero or one r's (that is, "an optional r")
`'r{2,5}'`	anywhere from two to five r's
`'r{2,}'`	two or more r's
`'r{4}'`	exactly 4 r's

Excerpt from flex manual. https://epaperpress.com/lexandyacc/download/flex.pdf

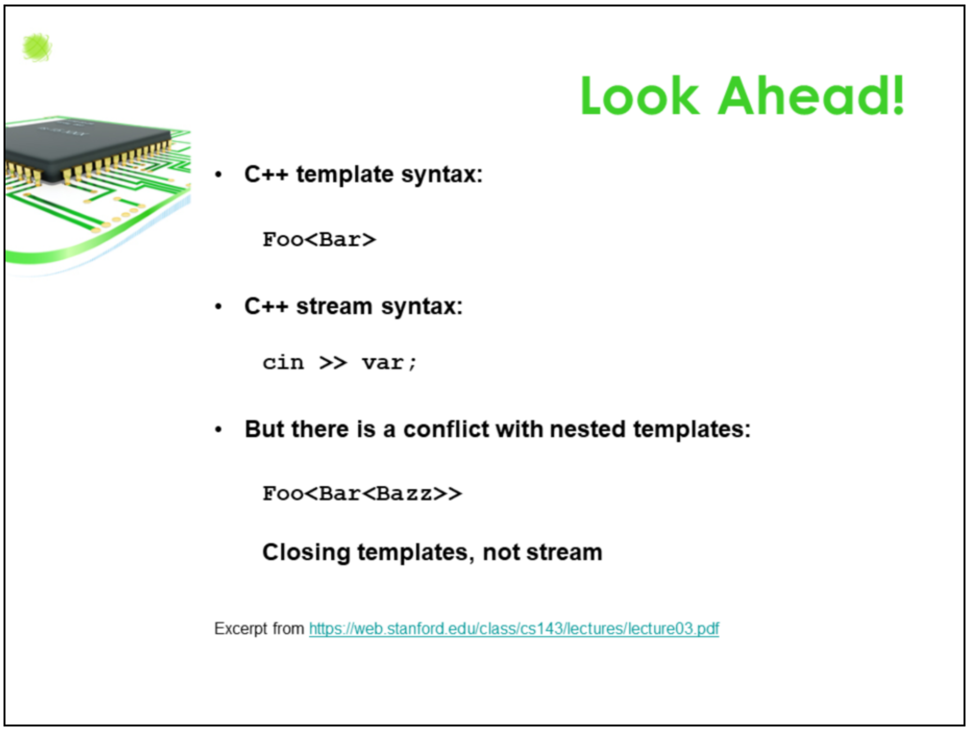
There are many tools available that make practical use of the regular expressions. The libraries are widely accessible, even JavaScript has regular expressions as part of the programming language. An interesting class of the tools making use of regular expressions is the automatic scanner generators. In this course we will focus on "flex", which will be used in the experiments. Like any tool that processes regular expressions, flex has specific notations to enable declaration of patterns with additional conventions. Be noted that the notational rules of regular expressions can be implementations specific. So, the notational conventions must be studied before developing regular expression based declarations.

Character sequences can be tested for their conformance to the patterns formally expressed by regular expressions. This conformance check is the operational basis of the lexical analyzers using the "micro-grammars" noted as regular expressions.

Regular expressions are objects that may be written as language literals. See https://www.w3schools.com/js/js_regexp.asp

See https://epaperpress.com/lexandyacc/download/flex.pdf

Note that the examples and the examinations will use notation of flex.



Look Ahead!

- C++ template syntax:

  `Foo<Bar>`

- C++ stream syntax:

  `cin >> var;`

- But there is a conflict with nested templates:

  `Foo<Bar<Bazz>>`

  Closing templates, not stream

Excerpt from https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Knowing that each regular expression corresponds to a DFA, which is a topic we will get into soon, a buffer of only one character would be sufficient to implement a lexical analyzer operating as a finite state machine (FSM). But, there may be occasions forcing us to decide the lexeme based on forthcoming characters which are not part of the lexeme being recognized.

One good example from the lecture notes of Fredrik Kjolstad is below.

C++ template syntax:

Foo<Bar>

C++ stream syntax:

cin >> var;

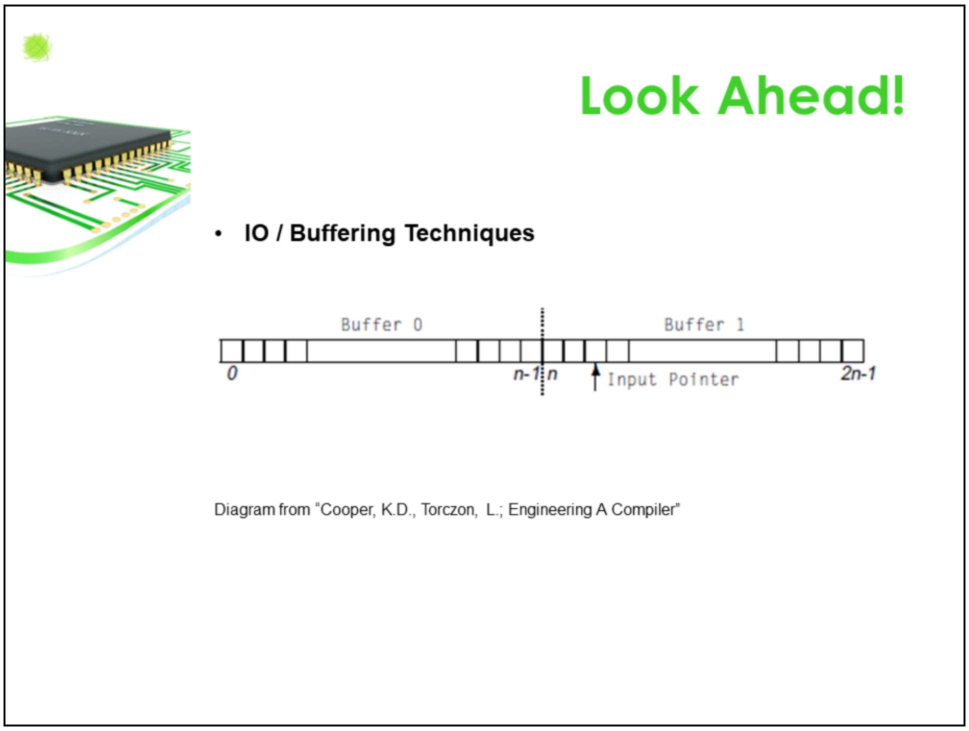
But there is a conflict with nested templates:

Foo<Bar<Bazz>>

Closing templates, not stream

The ambiguity problem in this example cannot be solved by the flex rule of “longest prevails”.

In such cases, input management techniques must be considered for looking ahead with probable solutions that utilize buffers. When buffers are proposed as solution to any particular problem, the buffer length becomes a limit to be complied by the whole analysis! Further to this, additional overhead to “unread” the data might be involved. See https://web.stanford.edu/class/cs143/lectures/lecture03.pdf



Look Ahead!

- **IO / Buffering Techniques**

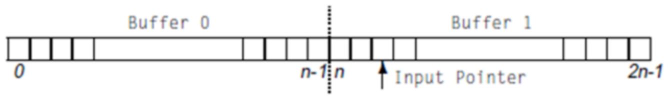
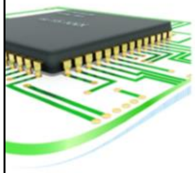


Diagram from "Cooper, K.D., Torczon, L.; Engineering A Compiler"

Use of two stage buffers managed as character queues is just an example for input management. Applications may choose their own method of handling buffers independent of low level parameters such as I/O buffer lengths as modern systems provide plenty of memory resources and very complex underlying I/O schemes that render consideration of low level parameters useless.

Dragon Book introduces “sentinels” while EAC defines “input” and “fence” pointers.

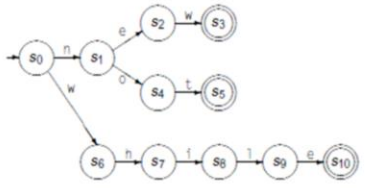


FSA Based Recognition

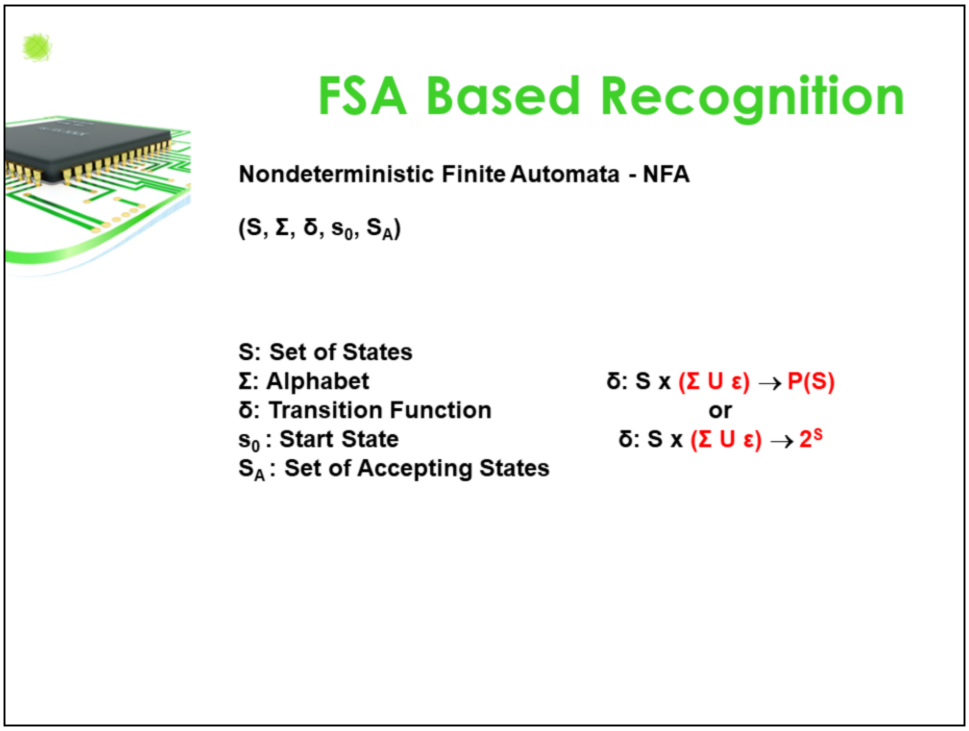
A Reminder on Finite State Automata

$(S, \Sigma, \delta, s_0, S_A)$

S: Set of States
Σ: Alphabet
δ: Transition Function
s_0 : Start State
S_A : Set of Accepting States



$\delta: S \times \Sigma \rightarrow S$



FSA Based Recognition

Nondeterministic Finite Automata - NFA

$(S, \Sigma, \delta, s_0, S_A)$

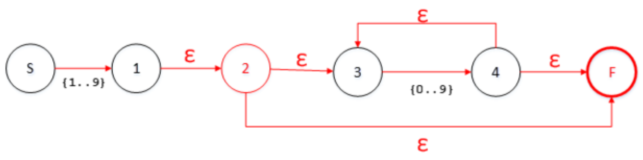
S: Set of States	
Σ: Alphabet	$\delta: S \times (\Sigma \cup \varepsilon) \rightarrow P(S)$
δ: Transition Function	or
s_0 : Start State	$\delta: S \times (\Sigma \cup \varepsilon) \rightarrow 2^S$
S_A : Set of Accepting States	

As noted before, regular expression based lexical analyzers are popular for a number of reasons including ad-hoc recognition or automatic generation of efficient recognizers.

Efficient checking of input against a regular expression requires transformation of regular expressions into structures that are used to drive the recognition algorithms in combination with the input.

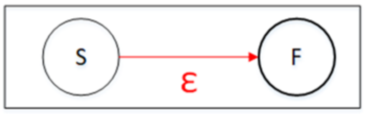
Thompson's Construction

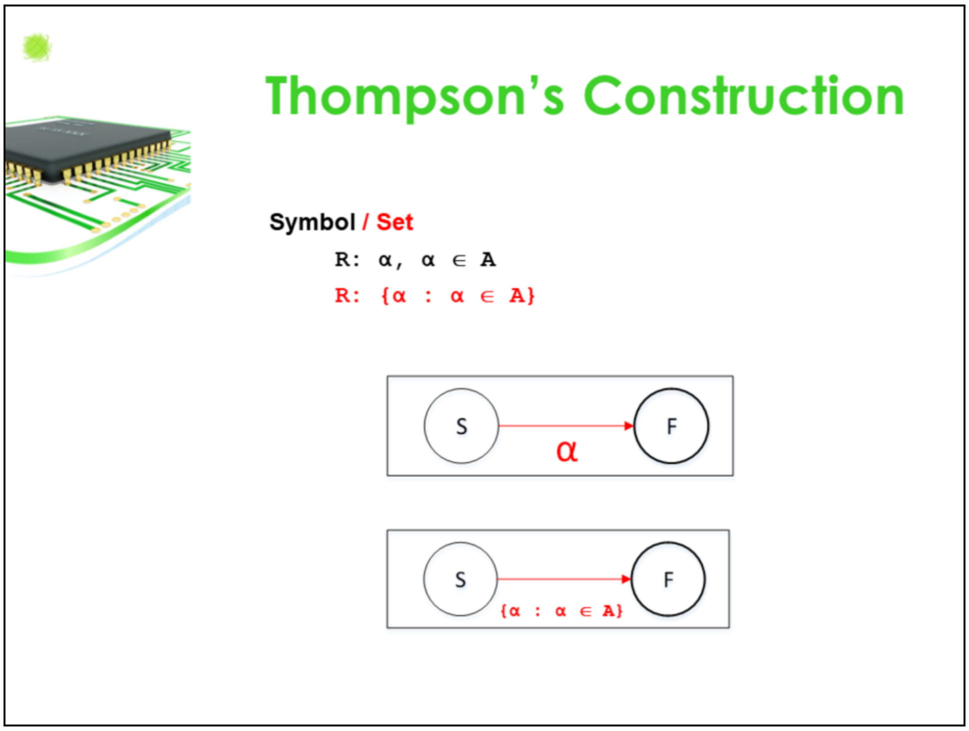
```
SimpleNumber: [1-9][0-9]*
```



Epsilon

R: ε



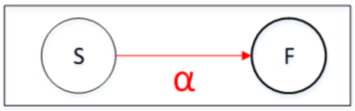


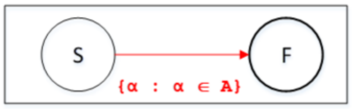
Thompson's Construction

Symbol / Set

R: α, α ∈ A

R: {α : α ∈ A}

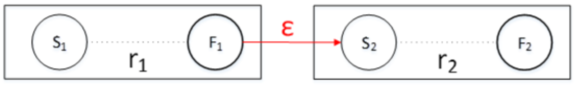




Thompson's Construction

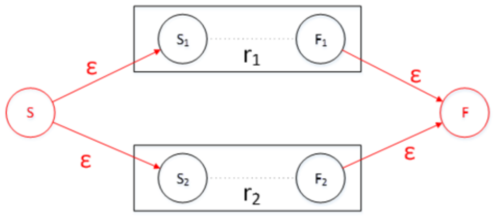
Concatenation

R: R_1R_2



Alternation

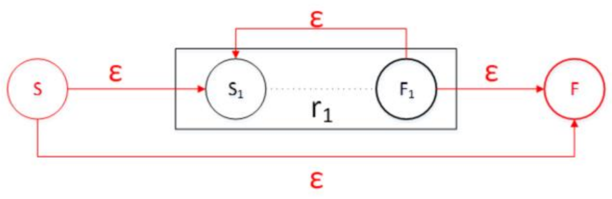
R: R_1 | R_2



Thompson's Construction

Kleene Closure

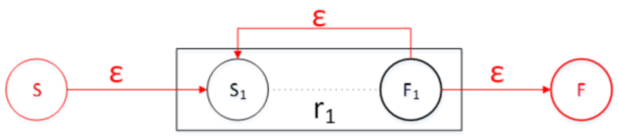
```
R: R1*
```



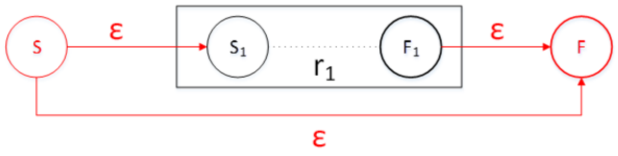
Thompson's Construction

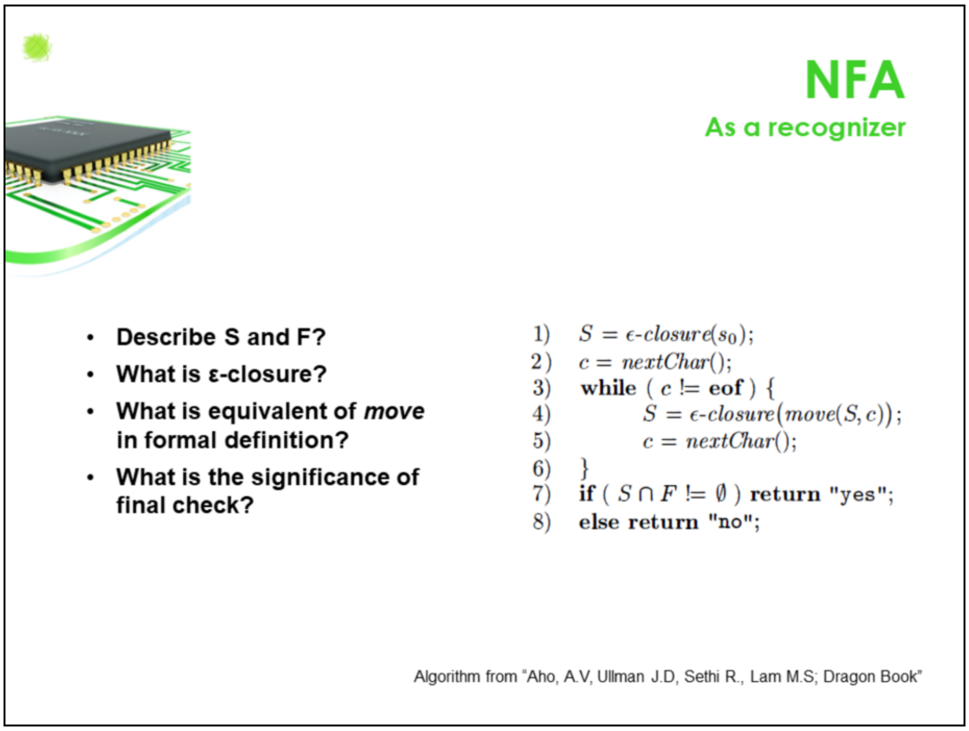
- **Practical notations**

R: R_1+



R: R_1?





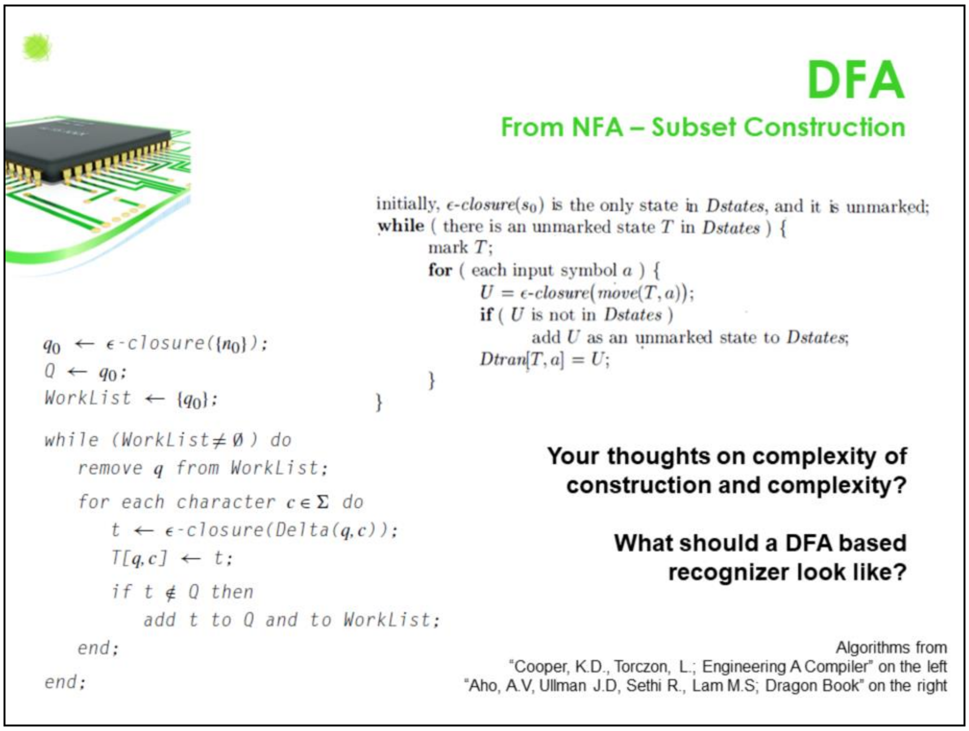
NFA

As a recognizer

- **Describe S and F?**
- **What is ε-closure?**
- **What is equivalent of *move* in formal definition?**
- **What is the significance of final check?**

```
$S = \epsilon\text{-}closure(s_0)$;
$c = nextChar()$;
while ( $c$ != eof ) {
        $S = \epsilon\text{-}closure(move(S, c))$;
        $c = nextChar()$;
}
if ( $S \cap F$ != $\emptyset$ ) return "yes";
else return "no";
```

Algorithm from "Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book"



DFA

From NFA – Subset Construction

```
initially, $\epsilon$-closure($s_0$) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state $T$ in Dstates ) {
        mark $T$;
        for ( each input symbol $a$ ) {
                $U = \epsilon$-closure(move($T, a$));
                if ( $U$ is not in Dstates )
                        add $U$ as an unmarked state to Dstates;
                Dtran[$T, a$] = $U$;
        }
}
```

```
$q_0 \leftarrow \epsilon$-closure({$n_0$});
$Q \leftarrow q_0$;
WorkList $\leftarrow$ {$q_0$};
while (WorkList $\neq \emptyset$) do
   remove $q$ from WorkList;
   for each character $c \in \Sigma$ do
      $t \leftarrow \epsilon$-closure(Delta($q,c$));
      T[$q,c$] $\leftarrow$ t;
      if $t \notin Q$ then
         add t to Q and to WorkList;
   end;
end;
```

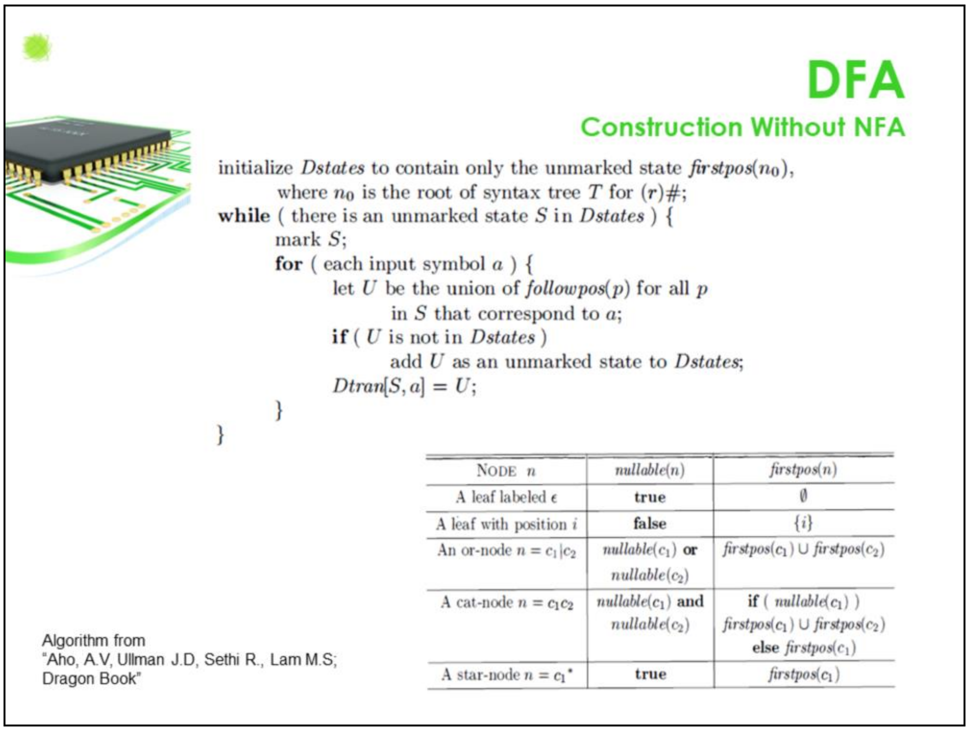
Your thoughts on complexity of construction and complexity?

What should a DFA based recognizer look like?

Algorithms from
"Cooper, K.D., Torczon, L.; Engineering A Compiler" on the left
"Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book" on the right

Subset construction (sometimes referred to as power set construction) is widely applied method to create a DFA structure having the equivalent recognition capability with increased efficiency because the transition function has no epsilon edges and an alphabet symbol leads to one and only one state.

The method establishes DFA nodes by identifying the eligible subsets from the universal set U of the NFA nodes. To do so, it visits each node in the NFA and calculates epsilon closure of the transitions for each input character. Each examination in this process produces and picks one subset from the U. In the worst case, all subsets are selected. Since the number of the picked subsets determines the number of the steps, the worst case execution time for the algorithm is O(n) = 2^n .



DFA

Construction Without NFA

```
initialize $Dstates$ to contain only the unmarked state $firstpos(n_0)$,
        where $n_0$ is the root of syntax tree $T$ for $(r)\#$;
while ( there is an unmarked state $S$ in $Dstates$ ) {
        mark $S$;
        for ( each input symbol $a$ ) {
                let $U$ be the union of $followpos(p)$ for all $p$
                        in $S$ that correspond to $a$;
                if ( $U$ is not in $Dstates$ )
                        add $U$ as an unmarked state to $Dstates$;
                $Dtran[S, a] = U$;
        }
}
```

NODE n	$nullable(n)$	$firstpos(n)$
A leaf labeled ϵ	**true**	$\emptyset$
A leaf with position i	**false**	$\{i\}$
An or-node $n = c_1 \mid c_2$	$nullable(c_1)$ **or** $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$
A cat-node $n = c_1c_2$	$nullable(c_1)$ **and** $nullable(c_2)$	**if** ($nullable(c_1)$) $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$
A star-node $n = c_1{}^*$	**true**	$firstpos(c_1)$

Algorithm from
"Aho, A.V, Ullman J.D, Sethi R., Lam M.S;
Dragon Book"

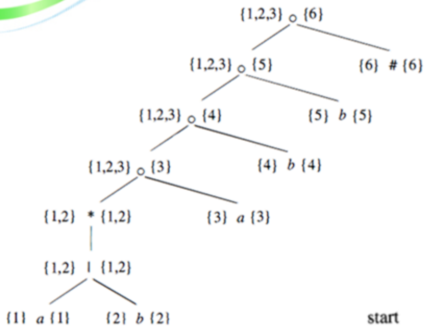
It is possible to construct DFA without using Thompson's construction, although the complexity remains as O(n) = 2^n, the resultant DFA may have fewer states.

Direct construction method applies a simple augmentation to the regular expression, establishes a syntax tree, and generates followpos table using firstpos and lastpos functions, which provide interim sets. It constructs a DFA structure by using the followpos table.

DFA

Construction Without NFA

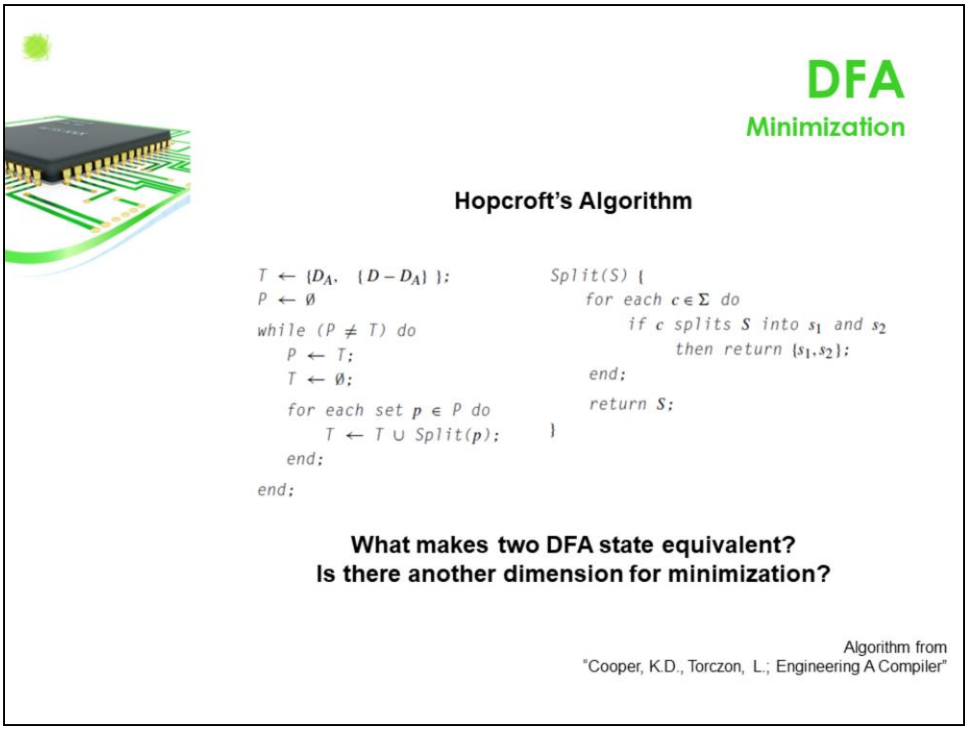
(a|b)*abb#



NODE n	*followpos*(n)
1	$\{1,2,3\}$
2	$\{1,2,3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

$$\begin{tikzcd}
\text{start} \arrow[r] & 123 \arrow[loop above, "b"] \arrow[r, "a"] & 1234 \arrow[loop below, "a"] \arrow[r, "b"] & 1235 \arrow[l, bend right, "a"'] \arrow[r, "b"] & \boxed{1236} \arrow[lll, bend right, "b"'] \arrow[ll, bend left, "a"]
\end{tikzcd}$$

Excerpts from "Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book"



DFA Minimization

Hopcroft's Algorithm

```
$T \leftarrow \{D_A,\ \{D - D_A\}\}$;
$P \leftarrow \emptyset$
while ($P \neq T$) do
    $P \leftarrow T$;
    $T \leftarrow \emptyset$;
    for each set $p \in P$ do
        $T \leftarrow T \cup$ Split($p$);
    end;
end;
```

```
Split(S) {
    for each $c \in \Sigma$ do
        if $c$ splits $S$ into $s_1$ and $s_2$
            then return $\{s_1, s_2\}$;
    end;
    return $S$;
}
```

What makes two DFA state equivalent?
Is there another dimension for minimization?

Algorithm from
"Cooper, K.D., Torczon, L.; Engineering A Compiler"

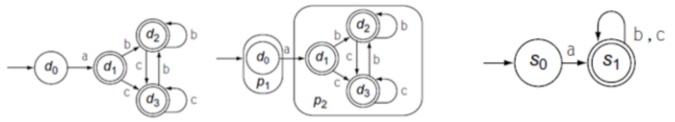
The DFA structures established by subset construction may generate nodes having identical functionality. When a group of nodes A has same target group of nodes B for all input symbols, the group of nodes A can be reduced to a single node, eliminating extra states in the DFA structure. Hopcroft's algorithm provides with an efficient method to eliminate the redundant states so that a compact set of states can be obtained.

Minimization of the states reduces the sparsity when DFA is implemented as a matrix of states and input symbols. Additional minimization may also be considered for minimization of the dimension that represents the input symbols. When rows represent states and columns represent input symbols, identification of the columns having common values helps reduce the number of the columns to represent the DFA. The unified columns are associated with mutually exclusive subsets of symbols, which may cause slight changes in DFA based recognition algorithm.

Speaking in matrix related terms, the row elimination and the column elimination algorithms can also be constructed by using standard sorting algorithms.

DFA
Minimization

a(b|c)*



(a) Original DFA (b) Initial Partition

Accept	State	a	b	c
	d_0	1	-	-
*	d_1	-	2	3
*	d_2	-	2	3
*	d_3	-	2	3

Group	Accept	State	a		b			c	
0		d_0	1		-			-	
1	*	d_1	-		2			3	
1	*	d_2	-		2			3	
1	*	d_3	-		2			3	

Applying Hopcroft's algorithm $g_0=\{d_0\}, g_1=\{d_1, d_2, d_3\}$

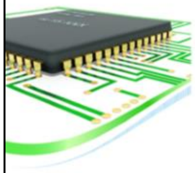
Group	Accept	State	a		b			c	
0		g_0	1		-			-	
1	*	g_1	-		1			1	

Applying Symbol Minimization m0={a}, m1={b, c}

Group	Accept	State	m0	m1
0		g_0	1	-
1	*	g_1	-	1

Excerpts from "Cooper, K.D., Torczon, L.; Engineering A Compiler"

Standard sorting algorithms can also be considered! After generating a sorting vector, enumeration of states becomes a straightforward process. Application of column minimization before state minimization is a common practice as alphabet size is usually much bigger than the number of states.

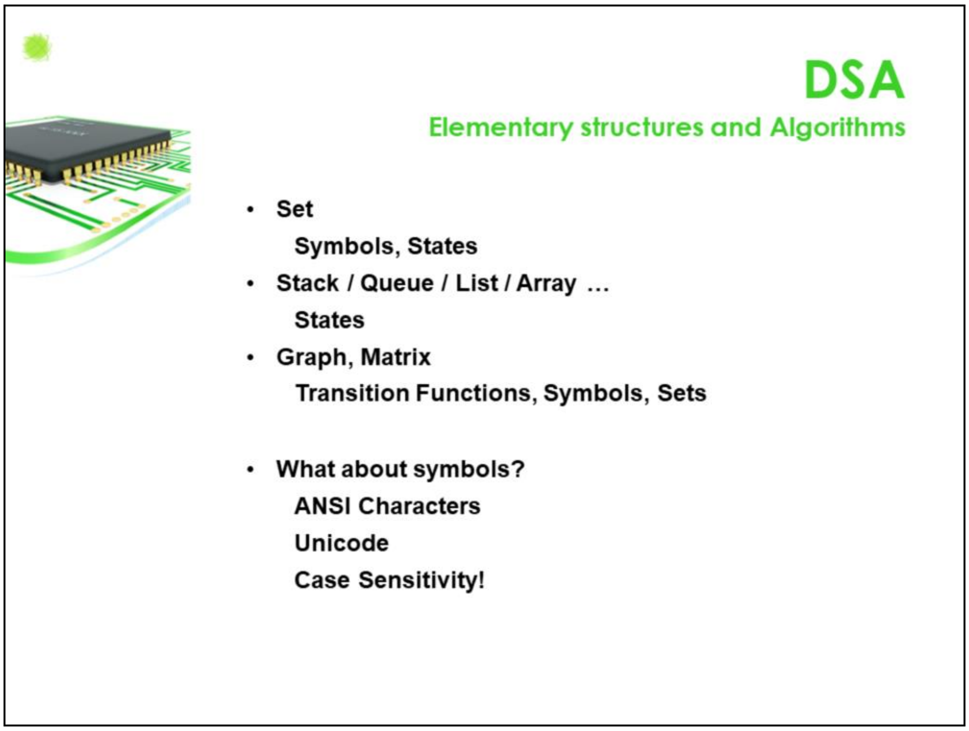


Complexities

Build and Recognition

Automaton	Initial	Per String
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

Table from "Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book"



DSA

Elementary structures and Algorithms

- Set
  - Symbols, States
- Stack / Queue / List / Array …
  - States
- Graph, Matrix
  - Transition Functions, Symbols, Sets

- What about symbols?
  - ANSI Characters
  - Unicode
  - Case Sensitivity!