Lexical Analysis

ACM India Summer School (PC-COE)

Manas Thakur

June 19th, 2024

Credits

- Slides from
  - Prof. Amitabha Sanyal, CSE, IIT Bombay
  - Prof. Uday Khedker, CSE, IIT Bombay
  - Prof. Rupesh Nasre, CSE, IIT Madras
  - Prof. Yannis Smaragdakis, University of Athens
  - *Prof. Manas Thakur, CSE, IIT Bombay!*

Introduction

The input program – as you see it.

```
main ()
{
    int i,sum;
    sum = 0;
    for (i=1; i<=10; i++)
       sum = sum + i;
    printf("%d\n",sum);
}
```

Introduction

The same program – as the compiler initially sees it. A continuous sequence of characters without any structure

```
main␣()↩{↩␣␣␣␣int␣i,sum;↩␣␣␣␣sum␣=␣0;↩␣␣␣␣for␣(i
=1;␣i<=10;␣i++);␣␣␣␣sum␣=␣sum␣+␣i;↩␣␣␣␣printf("%d\n",
sum);↩}
```

␣ – The blank space character

↩ – The return character

How do you make the compiler see what you see?

Discovering the structure of the program

Step 1:

a. Break up this string into the smallest meaningful units.

main	␣	(	)	↩	{	↩	␣	␣	␣	␣	int	␣	i	,	sum		
;	↩	␣	␣	␣	␣	sum	␣	=	␣	0	;	↩	␣	␣	␣	␣	
for	␣	(	i	=	1	;	␣	i	<=	10	;	␣	i	++	)	;	␣
␣	␣	␣	sum	␣	=	␣	sum	␣	+	␣	i	;	↩	␣	␣	␣	
␣	printf	(	"%d\n"	,	sum	)	;	↩	}								

We get a sequence of *lexemes* or *tokens*.

Discovering the structure of the program

Step 1:

b. During this process, remove the `␣` and the `↩` characters.

```
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i
; printf ( "%d\n" , sum ) ; }
```

Steps 1a. and 1b. are interleaved.

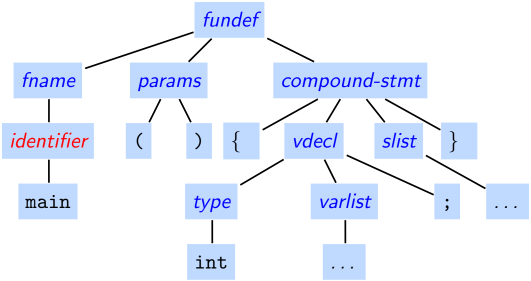
This is *lexical analysis* or *scanning*.

Discovering the structure of the program

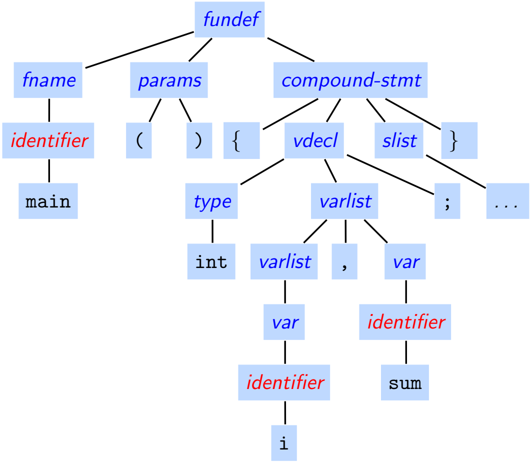
Step 2:

Now group the lexemes to form larger structures.

```
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i
; printf ( "%d\n" , sum ) ; }
```



Discovering the structure of the program



This is *syntax analysis* or *parsing*.

Discovering the structure of the program

Why is structure finding done in two steps?

- The process of breaking a program into lexemes (scanning) is easier. Use a separate technique to do this.
- Reduces the work to be done by the parser.

However, there are tools (Antlr for example) that indeed combine scanning with parsing.

Lexemes, Tokens and Patterns

Definition: *Lexical analysis* is the operation of dividing the input program into a sequence of *lexemes* (*tokens*).

Distinguish between

- *lexemes* – smallest logical units (words) of a program.
  Examples – `i, sum, for, 10, ++, "%d\n", <=`.
- *tokens* – sets of similar lexemes, i.e. lexemes which have a common syntactic description.
  Examples –
  identifier = {`i, sum, buffer`, ...}
  int_constant = {`1, 10, ...`}
  addop = {`+, -`}

Lexemes, Tokens and Patterns

What is the basis for grouping lexemes into tokens?

- Why can't `addop` and `mulop` be combined? Why can't `+` be a token by itself?

Lexemes which play similar roles during syntax analysis are grouped into a common token.

- Operators in `addop` and `mulop` have different roles – `mulop` has an higher precedence than `addop`.
- Each keyword plays a different role – is therefore a token by itself.
- Each punctuation symbol and each delimiter is a token by itself.
- All comments are uniformly ignored. They are all grouped under the same token.
- All identifiers are grouped in a common token.

Lexemes, Tokens and Patterns

Lexemes that are not passed to the later stages of a compiler:

- comments
- white spaces – tab, blanks and newlines
  - White spaces are more like separators between lexemes.

These too have to be detected and then ignored.

Lexemes, Tokens and Patterns

Apart from the token itself, the lexical analyser also passes other information regarding the token. These items of information are called *token attributes*

EXAMPLE

lexeme	**<token, token value>**
`3`	< const, 3>
`A`	<identifier, `A`>
`if`	<if, –>
`=`	<assignop, –>
`>`	<relop, `>`>
`;`	<semicolon, –>

Identifying and classifying tokens: Example

- Input:
  - `\tif (a>b)\n\t\tx = 0;\n\telse\n\t\tx = 1;`
- Say we have the following token types:
  - keywords, operators, identifiers, literals (constants), special symbols, white space
- How many tokens are there in this string?
- Example output (excluding white spaces):
  - <keyword, ‘if’>
  - <special_symbol, ‘(‘>
  - <identifier, ‘a’>
  - …

Lexemes, Tokens and Patterns

The lexical analyser:

- detects the next lexeme
- categorises it into the right token
- passes to the syntax analyser
  - the token name for further syntax analysis
  - the lexeme itself, in some form, for stages beyond syntax analysis

Lexemes, Tokens and Patterns

How does one describe the lexemes that make up the token *identifier*.

Variants in different languages.

- String of alphanumeric characters. The first character is an alphabet.
- a string of alphanumeric characters in which the first character is an alphabet. It has a length of at most 31.
- a string of alphabet or numeric or underline characters in which the first character is an alphabet or an underline. It has a length of at most 31. Any character after the 31st are ignored.

Such descriptions are called *patterns.* The description may be informal or formal. *Regular expressions* are the most commonly used formal patterns.

Lexemes, Tokens and Patterns

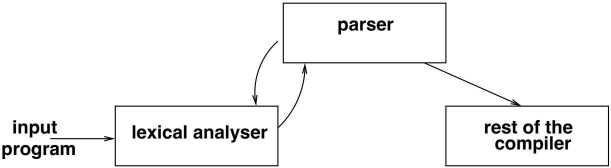
A pattern is used to

- *specify tokens* precisely
- *build a recognizer* from such specifications

Basic concepts and issues

Where does a lexical analyser fit into the rest of the compiler?

- The front end of most compilers is parser driven.
- When the parser needs the next token, it invokes the Lexical Analyser.
- Instead of analysing the entire input string, the lexical analyser sees enough of the input string to return a single token.
- The actions of the lexical analyser and parser are interleaved.



Creating a Lexical Analyzer

Two approaches:

1. *Hand code* – This is only of historical interest now.
   - Possibly more efficient.
2. *Use a generator* – To generate the lexical analyser from a formal description.
   - The generation process is faster.
   - Less prone to errors.

Automatic Generation of Lexical Analysers

- A formal description (specification) of the tokens of the source language, will consist of:
  - a regular expression describing each token, and
  - a code fragment called an action routine describing the action to be performed, on identifying each token.
- Here is a description of whole numbers and identifiers in form accepted by the lexical analyser generator Lex.

```
{DIGIT}+                    { yylval = atoi(yytext);
                              return NUM;
                            }
{LETTER}({LETTER}|{DIGIT})* { yylval = yytext;
                              return IDENTIFIER;
                            }
```

- The global variable yylval holds the token attribute (henceforth to be called token value).

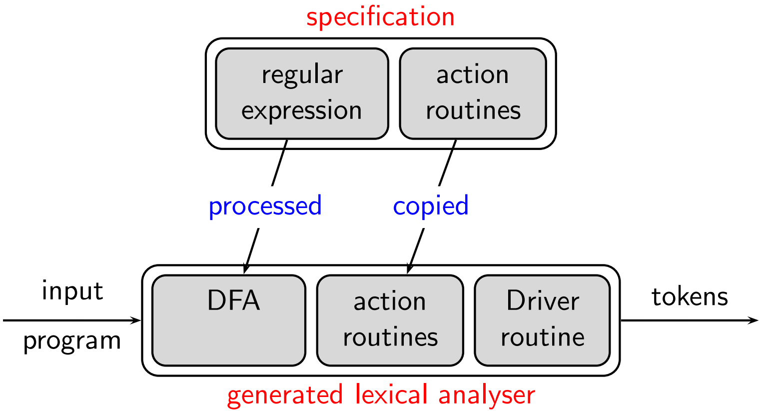
Automatic Generation of Lexical Analysers

Lex can read this description and generate a lexical analyser for whole numbers and identifiers. How?

- The generator puts together:
  - A deterministic finite automaton (DFA) constructed from the token specification.
  - A code fragment called a driver routine which can traverse any DFA.
  - Code for the action routines.
- These three things taken together constitutes the generated lexical analyser.

Automatic Generation of Lexical Analysers

- How is the lexical analyser generated from the description?



- Note that the driver routine is common for all generated lexical analysers.

Recap

In summary:

- The specification of a lexical analyser generator consists of two parts:
  1. Specification of tokens – done through regular expressions.
  2. Specification of actions - done through action routines.
- The lexical analyser generator:
  - Processes the regular expressions and forms a graph called DFA.
  - Copies the action routines without any change.
  - Adds a driver routine whose behaviour we described.

  These three things put together constitutes the lexical analyser.

Issues

- What are regular expressions? How can they be used to describe tokens?
- How can regular expresions be converted to DFA?

Introduction to Regular Expressions

A regular expressions denote a set of strings, also called *a language*. For example, **a*b** denotes the language {**b, ab, aab, aaab, . . .** }. We denote the language of a regular expression r as $L(r)$.

A single character is a regular expression.

- Examples: **a**, **Z**, **n**, **t**.
- Denotes a singleton set containing the character. **a** denotes the set {**a**}.

Introduction to Regular Expressions

ϵ is a regular expression.

- Denotes $\{\epsilon\}$, the set containing the empty string.

Introduction to Regular Expressions

If r and s are regular expressions then $r|s$ is a regular expression.

- Examples: $\mathbf{a}|\mathbf{b}|\ \ldots|\mathbf{z}|\mathbf{A}|\mathbf{B}|\ldots|\mathbf{Z}$ and $\mathbf{0}|\mathbf{1}|\ldots|\mathbf{9}$. Let us call these regular expressions **LETTER** and **DIGIT**.
- $L(r|s)$ is the union of strings in $L(r)$ and $L(s)$.

Introduction to Regular Expressions

If r and s are regular expressions then rs is a regular expression.

- Examples: `begin` – with an assumed associativity.
- `{LETTER}({LETTER}|{DIGIT})*`.
  - Notice that the braces required around `LETTER` is a lex requirement and denotes that it is a synonym for a regular expression and not the literal `LETTER`.
- $L(rs)$ is the concatenation of strings x and y such that $x \in L(r)$ and $y \in L(s)$.

Introduction to Regular Expressions

If r is a regular expressions then r^* is a regular expression.

- Examples: `({LETTER}|{DIGIT})*`
- $L(r^*)$ is the concatenation of zero or more strings from $L(r)$. Concatenation of zero strings is defined to be the null string.

Introduction to Regular Expressions

If r is a regular expressions then (r) is a regular expression. Parentheses are used for grouping.

- Examples: `({LETTER}|{DIGIT})*`
- The language denoted by (r) is $L(r)$.

Introduction to Regular Expressions

Shorthand: If r is a regular expressions then r^+ is a regular expression.

- Examples: {`DIGIT`}`+`
- $L(r^+)$ is the concatenation of one or more strings from $L(r)$.
- $r^+ = rr^*$.

Introduction to Regular Expressions

Shorthand: If r is a regular expressions then $r?$ is a regular expression.

- Examples: $\{$`DIGIT`$\}$? denotes zero or one occurrence of a digit.
- $r?$ stands for zero or one occurrence of strings in r.
- $r? = \epsilon | r$

Regular expressions provided by Lex

Expression	Describes	Example
`c`	any character c	`a`
`\c`	character c literally	`\*`
`"s"`	string s literally	`"**"`
`.`	any character except newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of line	`abc$`
`[s]`	any character in s	`[abc]`
`[^s]`	any character not in s	`[^abc]`
r`*`	zero or more r's	`a*`
r`+`	one or more r's	`a+`
r`?`	zero or one r	`a?`
$r_1 r_2$	r_1 then r_2	`ab`
$r_1 \mid r_2$	r_1 or r_2	`a|b`
(r)	r	`(a|b)`
r_1 / r_2	r_1 when followed by r_2	`abc/123`

Classwork

- Write a regex that represents strings over alphabet $\{a, b\}$ that start and end with a.
  - (a(a+b)*a) + a
- Strings with third last letter as a.
  - (a+b)*a(a+b)(a+b)
- Strings with exactly three bs.
  - a*ba*ba*ba*
- Strings over `Σ = {0,1}` with odd number of 1s:
  - HW

Example of token specification in Lex

```
[ \t\n]+                         {/*no action, no return*/}
if                               {return(IF);}
then                             {return(THEN);}
else                             {return(ELSE);}
{letter}({letter}|{digit})*      {yylval=install_id(); return(ID);}

-?{digit}+(\.{digit}+)?(E[+-]?{digit}+)?
                                 {yylval=atof(yytext); return(NUM);}

"<"                              {yylval=LT; return(RELOP);}
"<="                             {yylval=LE; return(RELOP);}
"+"                              {yylval=PLUS; return(ADDOP);}
"*"                              {yylval=MULT; return(MULOP);}
```

Tokenizing the Input Using DFAs

An Example for Scanning: Specifications

Let L and D denote the set of all letters and digits, respectively

Pattern	Token
int	INT
$L(L\|D)^*$	ID
D^+	NUM
$=$	$=$
;	;

We will scan the input string $\boxed{\text{int int32=5;}\hookleftarrow}$

Example for Scanning: DFA for the Patterns

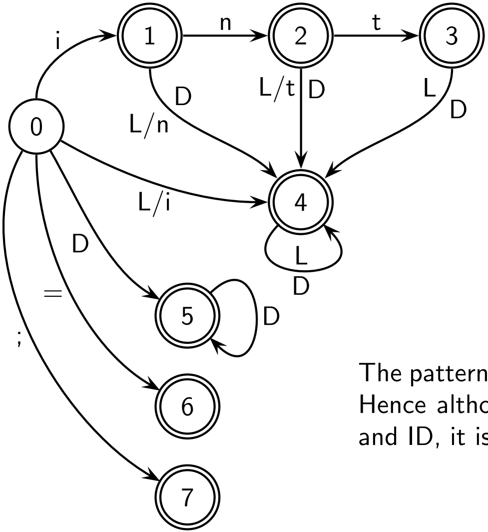
Formally, a Deterministic Finite Automaton (DFA) is a five tuple

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

where

- Σ is the input alphabet
- S is the set of states
- $s_o \in S$ is a unique start state
- $\delta : S \times \Sigma \rightarrow S$ is a transition function
- $F \subseteq S$ is a set of final states

Example for Scanning: DFA for the Patterns



States	Action
3	Found INT
1, 2, 4	Found ID
5	Found NUM
6	Found =
7	Found ;

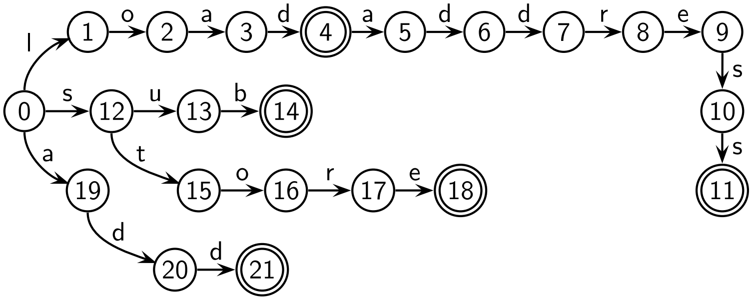
The patterns for INT precedes the pattern for ID Hence although state 3 could accept both INT and ID, it is made to accept only INT

Tutorial Problem On Scanning

- Find the occurrences of following substrings in a given input string

  load, loadaddress, add, sub, store

- Use the following automata



- Scan two input strings $\boxed{\text{loadsubadd}\hookleftarrow}$ and $\boxed{\text{loadaddsub}\hookleftarrow}$

Constructing DFAs

Constructing DFA for Multiple Patterns

- Join multiple DFAs/NFAs using ϵ transition
  Transition without consuming any input symbol
- This creates an NFA (Non-deterministic Finite Automaton)
  - Possible transition without consuming any input symbol
  - Possibly multiple transitions on the same input symbol
- Make the NFA deterministic by subset construction
  - Each state in the resulting DFA is a set of "similar" states of the NFA
  - The start state of the DFA is a union of all original start states (of multiple patterns)
  - Subsequent states are identified by finding out the sets of states of the NFA for each possible input symbol

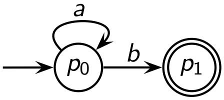
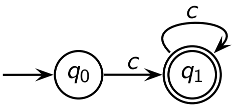
Constructing NFA for a Regular Expression

Consider a regular expression R. Apply steps 1 to 4 to construct an NFA for R inductively:

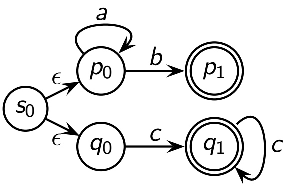
1. If R is a letter in the alphabet Σ, create a two state NFA that accepts the letter (single transition from the start state to a single final state on the letter)
2. If R is $R_1 \cdot R_2$, create an NFA by joining the two NFAs N_1 and N_2 by adding an epsilon transition from every final state of N_1 to the start state of N_2.
3. If the R is $R_1 \mid R_2$, create an NFA by joining the two NFAs N_1 and N_2 by creating a new start state s_0 and a new final state s_f. Add an epsilon transition from s_0 the start state of R_1 and similarly for R_2. Add an epsilon transition from every final state of N_1 to s_f and similarly for N_2.
4. If R is R_1*, create an NFA by adding an epslion transition from every final state of R_1 to the start state of R_1

   Alternatively, we can create a new start state s_0 with an epsilon transition to the start state of R_1 and a new final state s_f with epsilon transitions from the final states of R_1, and then add an espilon transition from s_f to s_0.

Constructing DFA for Multiple Patterns: Example 1

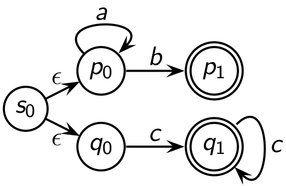
Pattern	DFA
$a^* b$	
c^+	

Constructing DFA for Multiple Patterns: Example 1

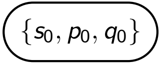


<table>
<tr><th rowspan="2">State</th><th colspan="3">Transition</th></tr>
<tr><th>a</th><th>b</th><th>c</th></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

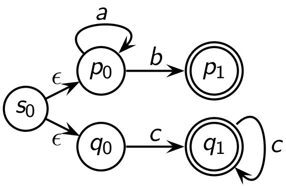
Constructing DFA for Multiple Patterns: Example 1



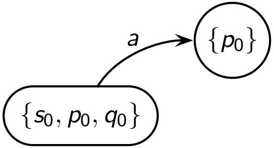
State	*Transition*		
	a	b	c
$\{s_0, p_0, q_0\}$			



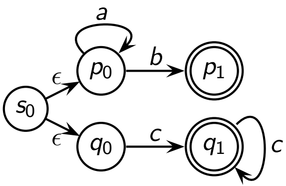
Constructing DFA for Multiple Patterns: Example 1



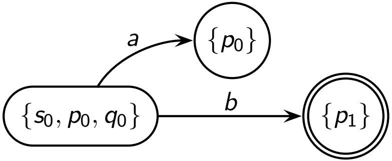
State	Transition		
	a	b	c
$\{s_0, p_0, q_0\}$	$\{p_0\}$		
$\{p_0\}$			



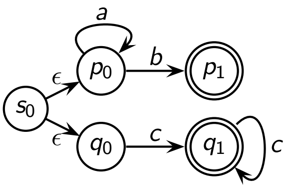
Constructing DFA for Multiple Patterns: Example 1



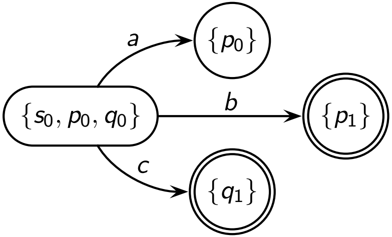
State	Transition		
	a	b	c
$\{s_0, p_0, q_0\}$	$\{p_0\}$	$\{p_1\}$	
$\{p_0\}$			
$\{p_1\}$			



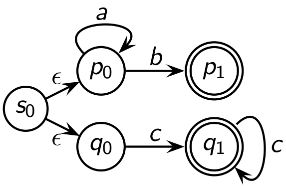
Constructing DFA for Multiple Patterns: Example 1



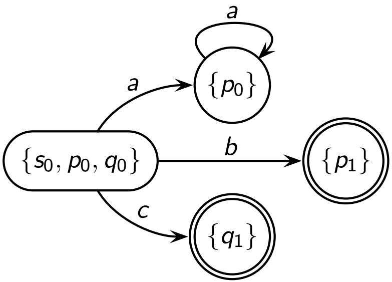
State	*Transition*		
	a	*b*	*c*
$\{s_0, p_0, q_0\}$	$\{p_0\}$	$\{p_1\}$	$\{q_1\}$
$\{p_0\}$			
$\{p_1\}$			
$\{q_1\}$			



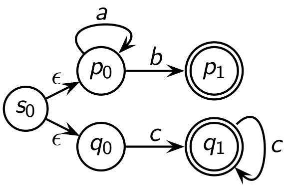
Constructing DFA for Multiple Patterns: Example 1



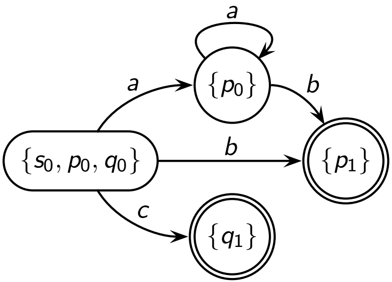
State	Transition		
	a	b	c
$\{s_0, p_0, q_0\}$	$\{p_0\}$	$\{p_1\}$	$\{q_1\}$
$\{p_0\}$	$\{p_0\}$		
$\{p_1\}$			
$\{q_1\}$			



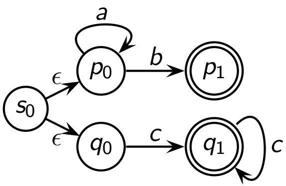
Constructing DFA for Multiple Patterns: Example 1



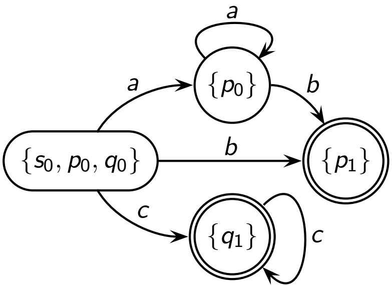
State	*Transition*		
	a	*b*	*c*
$\{s_0, p_0, q_0\}$	$\{p_0\}$	$\{p_1\}$	$\{q_1\}$
$\{p_0\}$	$\{p_0\}$	$\{p_1\}$	
$\{p_1\}$			
$\{q_1\}$			



Constructing DFA for Multiple Patterns: Example 1



State	*Transition*		
	a	b	c
$\{s_0, p_0, q_0\}$	$\{p_0\}$	$\{p_1\}$	$\{q_1\}$
$\{p_0\}$	$\{p_0\}$	$\{p_1\}$	
$\{p_1\}$			
$\{q_1\}$			$\{q_1\}$



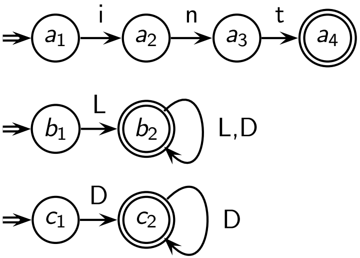
Constructing DFA for Multiple Patterns: Example 2

Let L and D denote the set of all letters and digits, respectively

Pattern	Token
int	INT
$L(L\|D)^*$	ID
D^+	NUM
=	=
;	;

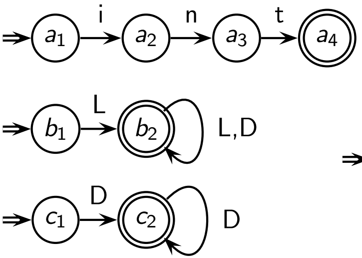
For convenience, we will ignore the last two patterns that are completely independent

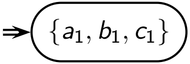
Constructing DFA for Multiple Patterns: Example 2



State	i	n	t	$L - \{i, n, t\}$	D

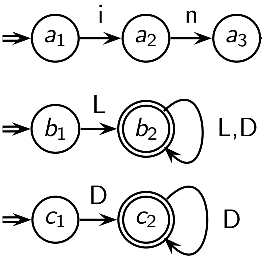
Constructing DFA for Multiple Patterns: Example 2

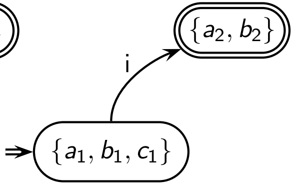




State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$					

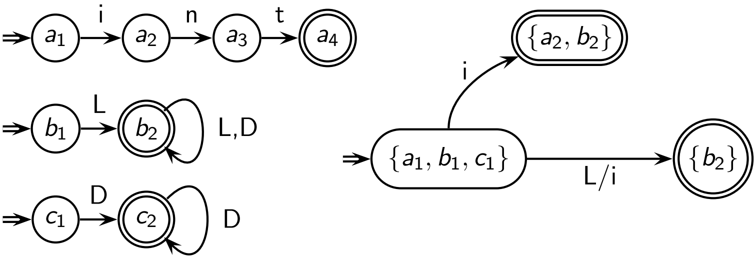
Constructing DFA for Multiple Patterns: Example 2





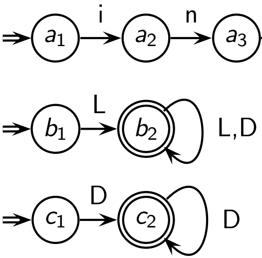
State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$				
$\{a_2, b_2\}$					

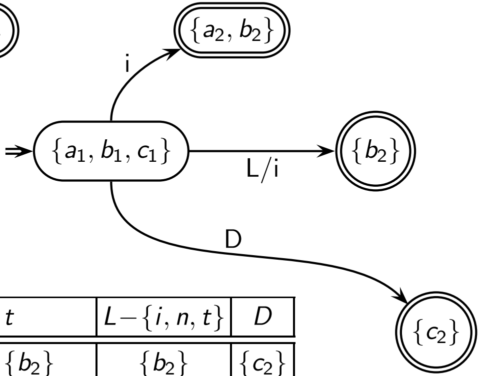
Constructing DFA for Multiple Patterns: Example 2



State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	
$\{a_2, b_2\}$					
$\{b_2\}$					

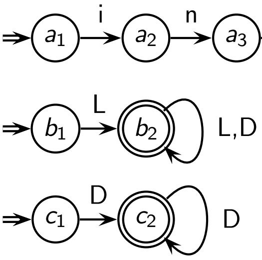
Constructing DFA for Multiple Patterns: Example 2

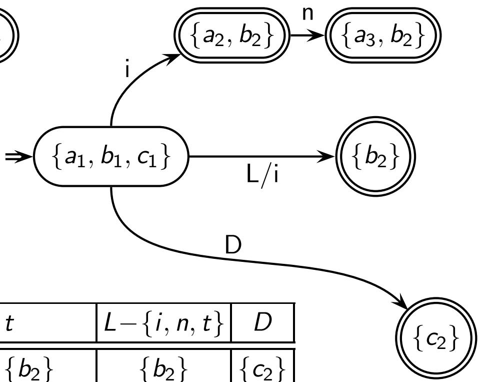




State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$					
$\{b_2\}$					
$\{c_2\}$					

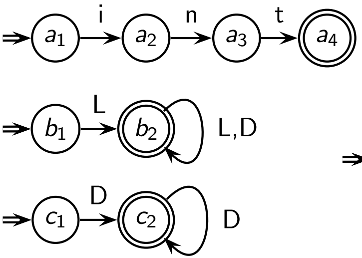
Constructing DFA for Multiple Patterns: Example 2

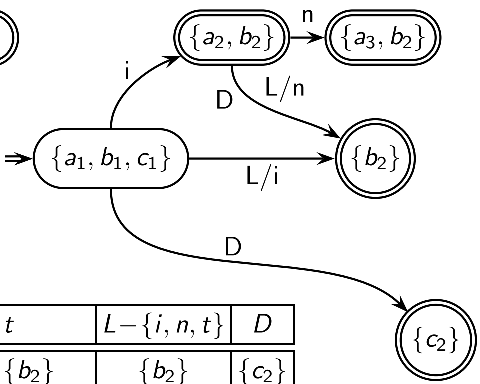




State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$		$\{a_3, b_2\}$			
$\{b_2\}$					
$\{c_2\}$					

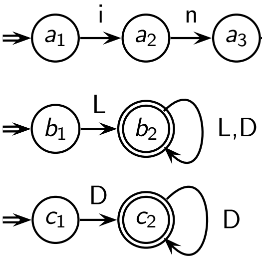
Constructing DFA for Multiple Patterns: Example 2

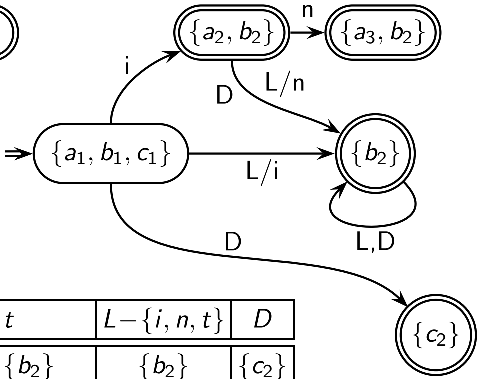




State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$	$\{b_2\}$	$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{b_2\}$					
$\{c_2\}$					
$\{a_3, b_2\}$					

Constructing DFA for Multiple Patterns: Example 2





State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$	$\{b_2\}$	$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{c_2\}$					
$\{a_3, b_2\}$					

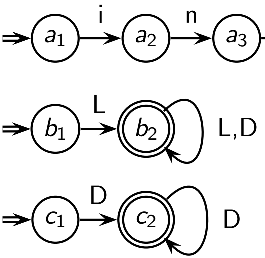
IIT Bombay
cs302: Implementation of Programming Languages

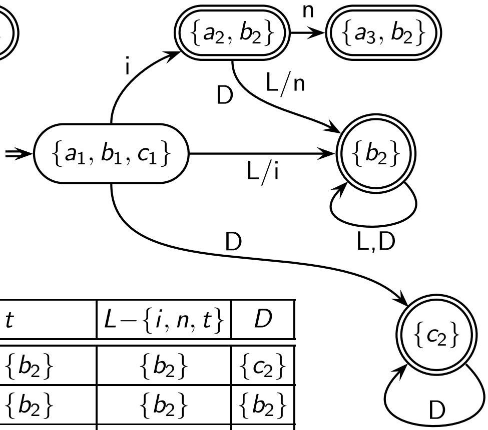
Topic:

Scanning

Section:

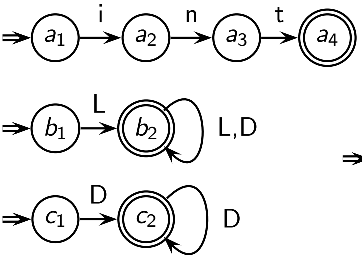
Constructing DFA for Multiple Patterns: Example 2

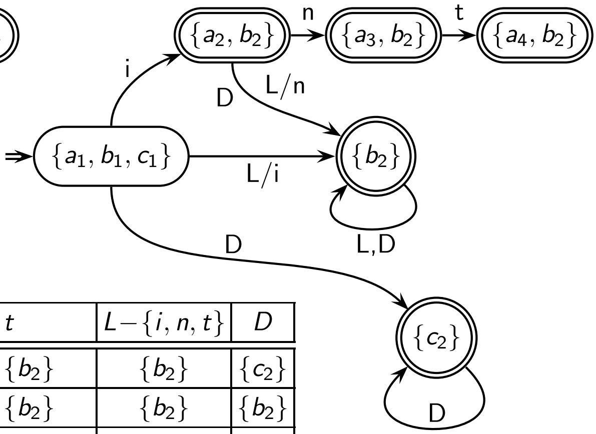




State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$	$\{b_2\}$	$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{c_2\}$					$\{c_2\}$
$\{a_3, b_2\}$					

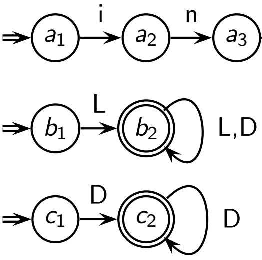
Constructing DFA for Multiple Patterns: Example 2

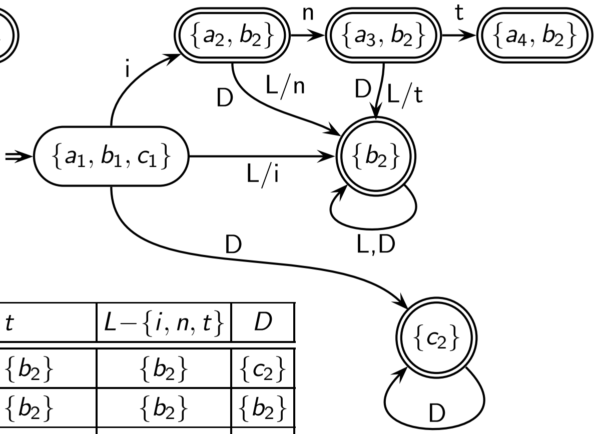




State	i	n	t	$L - \{i, n, t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$	$\{b_2\}$	$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{c_2\}$					$\{c_2\}$
$\{a_3, b_2\}$			$\{a_4, b_2\}$		
$\{a_4, b_2\}$					

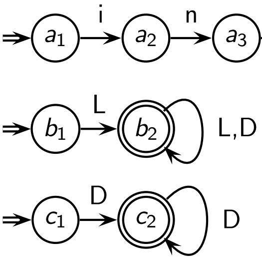
Constructing DFA for Multiple Patterns: Example 2

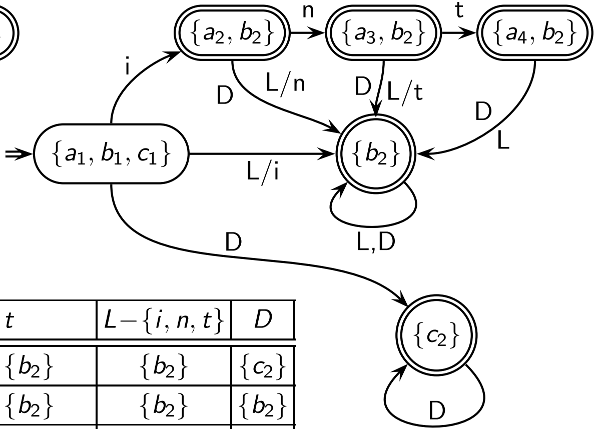




State	i	n	t	$L-\{i,n,t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$	$\{b_2\}$	$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{c_2\}$					$\{c_2\}$
$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{a_4, b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{a_4, b_2\}$					

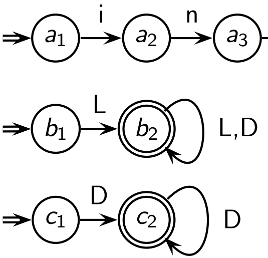
Constructing DFA for Multiple Patterns: Example 2

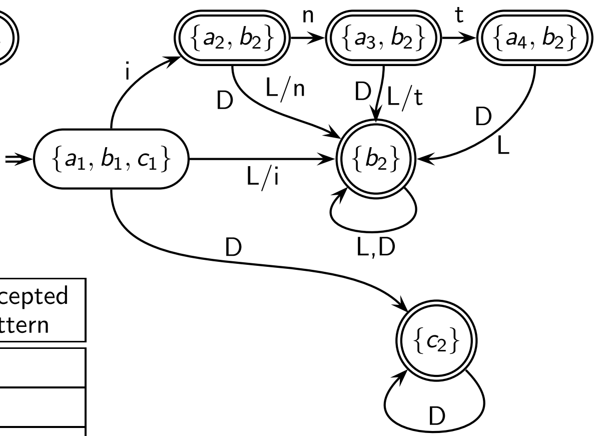




State	i	n	t	$L - \{i, n, t\}$	D
$\{a_1, b_1, c_1\}$	$\{a_2, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{c_2\}$
$\{a_2, b_2\}$	$\{b_2\}$	$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{c_2\}$					$\{c_2\}$
$\{a_3, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{a_4, b_2\}$	$\{b_2\}$	$\{b_2\}$
$\{a_4, b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$	$\{b_2\}$

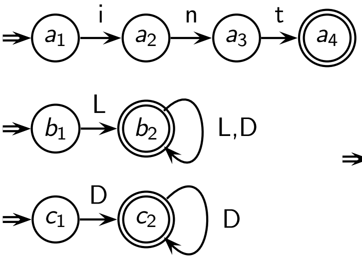
Constructing DFA for Multiple Patterns: Example 2

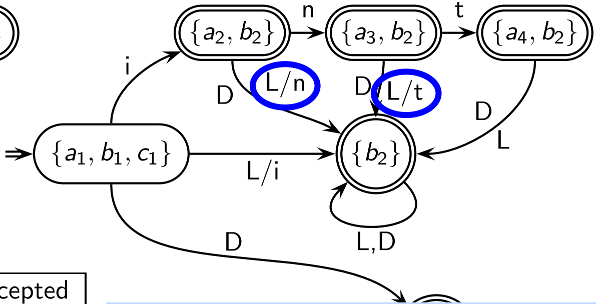




State	Possible Patterns	Accepted Pattern
$\{a_1, b_1, c_1\}$		
$\{a_2, b_2\}$	ID	ID
$\{b_2\}$	ID	ID
$\{c_2\}$	NUM	NUM
$\{a_3, b_2\}$	ID	ID
$\{a_4, b_2\}$	INT, ID	INT

Constructing DFA for Multiple Patterns: Example 2





State	Possible Patterns	Accepted Pattern
$\{a_1, b_1, c_1\}$		
$\{a_2, b_2\}$	ID	ID
$\{b_2\}$	ID	ID
$\{c_2\}$	NUM	NUM
$\{a_3, b_2\}$	ID	ID
$\{a_4, b_2\}$	INT, ID	INT

Longest match. Lexeme “int” reaches state $\{a_4, b_2\}$ whereas lexeme “integer” reaches the state $\{b_2\}$

First maching rule preferred. Transitions L/n and L/t to state $\{b_2\}$ ensure that INT is prefered over ID for the lexeme “int”

Some considerations

- How to distinguish between patterns with common prefixes:
  - <, <=, <<

Some considerations

- How to distinguish between patterns with common prefixes:
  - <, <=, <<
  - Need to "look ahead" before taking a decision

Some considerations

- How to distinguish between patterns with common prefixes:
  - <, <=, <<
  - Need to “look ahead” before taking a decision

- Clashes between token types (e.g., `then` versus `thenVar`)

Some considerations

- How to distinguish between patterns with common prefixes:
  - <, <=, <<
  - Need to “look ahead” before taking a decision
- Clashes between token types (e.g., `then` versus `thenVar`)
  - Assign priorities while checking (e.g., keywords before identifiers)

Some considerations

- How to distinguish between patterns with common prefixes:
  - <, <=, <<
  - Need to “look ahead” before taking a decision
- Clashes between token types (e.g., `then` versus `thenVar`)
  - Assign priorities while checking (e.g., keywords before identifiers)
  - Start with an identifier and if the value matches a reserved word, then change its type

Some considerations

- How to distinguish between patterns with common prefixes:
  - <, <=, <<
  - Need to “look ahead” before taking a decision

- Clashes between token types (e.g., `then` versus `thenVar`)
  - Assign priorities while checking (e.g., keywords before identifiers)
  - Start with an identifier and if the value matches a reserved word, then change its type

- Detecting and recovering from errors?

Errors in lexical analysis

- It is difficult for a lexer to identify errors.
  - Limited resources: e.g., no context information.

Errors in lexical analysis

- It is difficult for a lexer to identify errors.
  - Limited resources: e.g., no context information.
- `fi (a = f(x))`
  - Is `fi` a misspelling for `if`, or a function identifier?

Errors in lexical analysis

- It is difficult for a lexer to identify errors.
  - Limited resources: e.g., no context information.
- `fi (a = f(x))`
  - Is `fi` a misspelling for `if`, or a function identifier?
- As `fi` is a valid lexeme for the token identifier, the lexer must return the token `<id, fi>`.

Errors in lexical analysis

- It is difficult for a lexer to identify errors.
  - Limited resources: e.g., no context information.
- `fi (a = f(x))`
  - Is `fi` a misspelling for `if`, or a function identifier?
- As `fi` is a valid lexeme for the token identifier, the lexer must return the token `<id, fi>`.
- A later phase (parser or semantic analyzer) may be able to catch the error.

Errors in lexical analysis

- It is difficult for a lexer to identify errors.
  - Limited resources: e.g., no context information.
- `fi (a = f(x))`
  - Is `fi` a misspelling for `if`, or a function identifier?
- As `fi` is a valid lexeme for the token identifier, the lexer must return the token `<id, fi>`.
- A later phase (parser or semantic analyzer) may be able to catch the error.
- But some errors can be caught by a lexer:
  - int %x;
  - if (a < b);$

Errors in lexical analysis

- It is difficult for a lexer to identify errors.
  - Limited resources: e.g., no context information.
- `fi (a = f(x))`
  - Is `fi` a misspelling for `if`, or a function identifier?
- As `fi` is a valid lexeme for the token identifier, the lexer must return the token `<id, fi>`.
- A later phase (parser or semantic analyzer) may be able to catch the error.
- But some errors can be caught by a lexer:
  - int %x;
  - if (a < b);$

What should a lexer do on detecting an error?

Error handling in lexical analysis

Error handling in lexical analysis

- Panic and exit(1).

Error handling in lexical analysis

- Panic and exit(1).
- Try to recover from the error and proceed.

Error handling in lexical analysis

- Panic and exit(1).

- Try to recover from the error and proceed.

Why?

Error handling in lexical analysis

- Panic and exit(1).

- Try to recover from the error and proceed.

Why?

- We are a compiler; not an interpreter!

Error recovery in lexical analysis

Error recovery in lexical analysis

- Delete one character from the input.

Error recovery in lexical analysis

- Delete one character from the input.
- Insert a missing character into the remaining input.

Error recovery in lexical analysis

- Delete one character from the input.
- Insert a missing character into the remaining input.
  - Which one?

Error recovery in lexical analysis

- Delete one character from the input.
- Insert a missing character into the remaining input.
  - Which one?
- Replace a character by another character.

Error recovery in lexical analysis

- Delete one character from the input.
- Insert a missing character into the remaining input.
  - Which one?
- Replace a character by another character.
- Transpose two adjacent characters.

Error recovery in lexical analysis

- Delete one character from the input.
- Insert a missing character into the remaining input.
  - Which one?
- Replace a character by another character.
- Transpose two adjacent characters.
- Theoretical problem: Find the smallest number of transformations (add, delete, replace) needed to convert a source program to one that consists only of valid lexemes.
  - Too expensive in practice.

Error recovery in lexical analysis

- Delete one character from the input.
- Insert a missing character into the remaining input.
  - Which one?
- Replace a character by another character.
- Transpose two adjacent characters.
- Theoretical problem: Find the smallest number of transformations (add, delete, replace) needed to convert a source program to one that consists only of valid lexemes.
  - Too expensive in practice.
- In practice, most lexical errors involve a single character.

Limits of Regular Languages

- Not all languages are regular.

Limits of Regular Languages

- Not all languages are regular.
- Try constructing an FA for the following languages:
  - $L = \{0^n1^n\}$
  - $L = \{wcw^r \mid w \in \Sigma*\}$

Limits of Regular Languages

- Not all languages are regular.
- Try constructing an FA for the following languages:
  - $L = \{0^n 1^n\}$
  - $L = \{wcw^r \mid w \in \Sigma^*\}$

  Note: neither of these is a regular expression!

Limits of Regular Languages

- Not all languages are regular.
- Try constructing an FA for the following languages:
  - $L = \{0^n1^n\}$
  - $L = \{wcw^r \mid w \in \Sigma*\}$

  Note: neither of these is a regular expression!
- FAs cannot count properly!

Limits of Regular Languages

- Not all languages are regular.
- Try constructing an FA for the following languages:
  - $L = \{0^n1^n\}$
  - $L = \{wcw^r \mid w \in \Sigma*\}$

  Note: neither of these is a regular expression!
- FAs cannot count properly!
- However, this is a little subtle. One can construct FAs for:
  - Alternating 0s and 1s
    - (ε | 1)(01)*(ε | 0)
  - Sets of pairs of 0s and 1s
    - (01 | 10)+

Limits of Regular Languages

- Not all languages are regular.
- Try constructing an FA for the following languages:
  - $L = \{0^n1^n\}$
  - $L = \{wcw^r \mid w \in \Sigma*\}$

  Note: neither of these is a regular expression!
- FAs cannot count properly!
- However, this is a little subtle. One can construct FAs for:
  - Alternating 0s and 1s
    - (ε | 1)(01)*(ε | 0)
  - Sets of pairs of 0s and 1s
    - (01 | 10)+

We start parsing sentences in the next class.

Research @ CompL, IITB

- Precise yet efficient program analysis for languages like Java with staged compilation.
- Performing more aggressive speculative optimizations in JITs using static analysis.
- Discovering new optimizations for upcoming features such as value types.
- Saving compilation effort by recording dynamism in languages like R, Python & JS.
- Transforming Scala code to easily parallelizable functional style using IDE plugins.
- Cheering up compiler designers with novel debuggers and visualization tools.

Research @ CompL, IITB

- Precise yet efficient program analysis for languages like Java with staged compilation.
- Performing more aggressive speculative optimizations in JITs using static analysis.
- Discovering new optimizations for upcoming features such as value types.
- Saving compilation effort by recording dynamism in languages like R, Python & JS.
- Transforming Scala code to easily parallelizable functional style using IDE plugins.
- Cheering up compiler designers with novel debuggers and visualization tools.

➤Join us and become the next CompLer!

https://www.cse.iitb.ac.in/~manas