

Home Page

Title Page

Contents



Page 1 of 55

Go Back

Full Screen

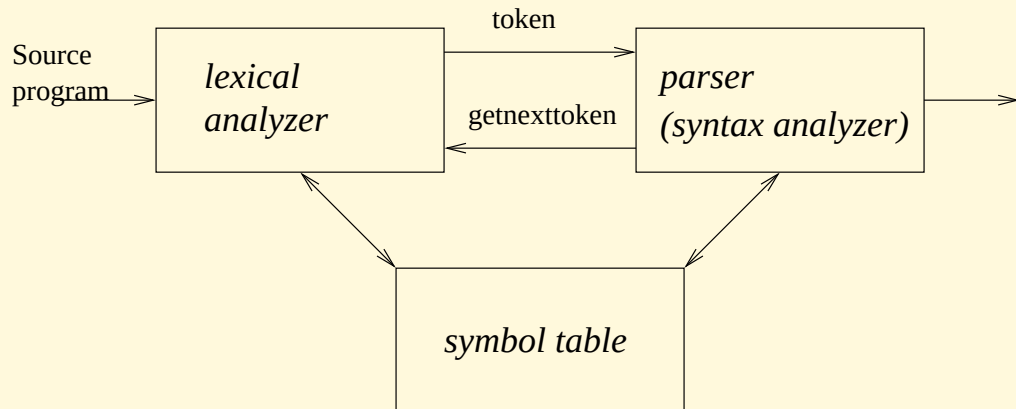
Close

Quit

LEXICAL ANALYSIS

Lecture Slides for the course
Introduction to Compiler Design and Construction
Srinath Srinivasa
Indian Institute of Information Technology, Bangalore
India 560066
sri@iiitb.ac.in

ROLE OF THE LEXICAL ANALYZER



The lexical analyzer works in lock-step with the parser. The parser requests the lexical analyzer for the next token whenever it requires one.

Lexical analyzer may also perform other auxiliary operations like removing redundant white spaces, removing token separators (like semicolon ;) etc.

TOKENS, PATTERNS, LEXEMES

Token: A logical building block of the language. (Ex. **id**, **keyword**, etc.)

Pattern: Usually a regular expression. A rule that describes a token. (Ex. **id** = $[A - Z a - z_][A - Z a - z_0 - 9]^*$)

Lexeme: Actual text that matches a pattern and is recognized as a token. (Ex: `int` identified as token **keyword**)

ATTRIBUTES FOR TOKENS

When a token represents many lexemes, the lexical analyzer has to also send the lexeme in addition to the token when invoked by the parser.

Example: Consider the statement:

$$\text{dist} = 0.5 * g * t * t;$$

1. `getnexttoken()` returns (**id**, symbol table pointer for `dist`)
2. `getnexttoken()` returns (**assign_op**)
3. `getnexttoken()` returns (**float_num**, 0.5)

...

STRATEGIES FOR IMPLEMENTING A SCANNER

1. Use a lexical-analyzer generator like flex to produce a lexical analyzer from a regular expression based specification. The generator provides routines for reading and buffering the input.
2. Design a lexical analyzer in a high-level programming language like C and use the input and buffering techniques provided by the language.
3. Write lexical analyzer in assembly language and explicitly manage input and buffering.

The above are in increasing order of complexity and efficiency.

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 6 of 55

Go Back

Full Screen

Close

Quit

LOOKAHEAD AND BUFFERING

Many times, a scanner has to *look ahead* several characters from the current character in order to recognize a token. For example `int` is a keyword in C, while the term `inp` may be a variable name. When the character 'i' is encountered, the scanner cannot decide whether it is a keyword or a variable name until it reads two more characters.

In order to efficiently move back and forth in the input stream, *input buffering* is used.

Home Page

Title Page

Contents



Page 7 of 55

Go Back

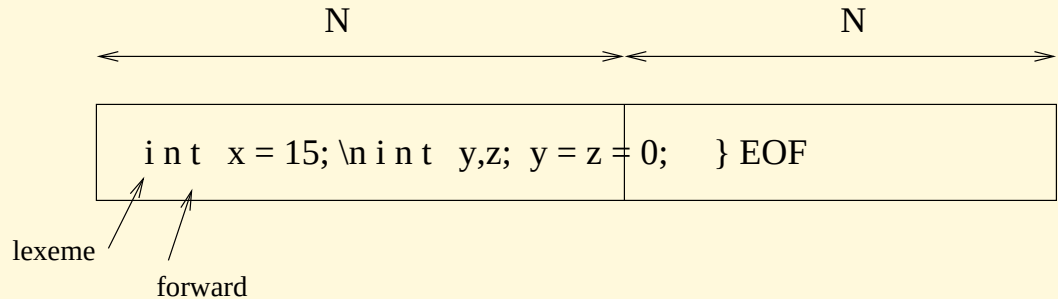
Full Screen

Close

Quit

BUFFERING

Lookahead with 2N buffering:



SPECIFICATION OF TOKENS

Regular Expressions

1. Any atom (character) is a regular expression
2. If r and s are regular expressions, then so are $r \cdot s$, $r + s$, r^* and (r)

The language described by a regular expression is called a *regular set*. The operators $+$ and $\cdot$ are called *regular operators*.

SPECIFICATION OF TOKENS

Properties of Regular Expressions

$r + s = s + r$ (union is commutative)

$r(s + t) = rs + rt$, $(s + t)r = sr + tr$ (concatenation distributes over union)

$r + (s + t) = (r + s) + t$ (union is associative)

$\epsilon r = r$, $r\epsilon = r$ (ϵ is the identity element for concatenation)

$r^{**} = r^*$ (closure is idempotent)

SPECIFICATION OF TOKENS

Regular Definitions: It is often desirable to name regular expression groupings for being able to recognize tokens at different levels.

Example: Identifiers in C are defined as a letter or underscore followed by zero or more letters or underscores or digits. Regular definition for id is as follows:

letter $\rightarrow A + B + C + \dots + Z + a + b \dots + z$

digit $\rightarrow 0 + 1 + 2 + \dots + 9$

id $\rightarrow (\text{letter}+_*)(\text{letter}+_*\text{digit})^*$

Note: *Be careful with recursive regular definitions! They may not be regular.*

Home Page

Title Page

Contents



Page 11 of 55

Go Back

Full Screen

Close

Quit

SOME EXERCISES

1. Write regular definitions for specifying a floating point number in a programming language like C.
2. Write regular definitions for specifying an integer array declaration in a language like C.

RECOGNITION OF TOKENS

Recognition of tokens implies implementing a regular expression recognizer. This entails implementation of a DFA.

A Deterministic Finite Automaton (DFA) is defined as a tuple $DFA = (S, A, s_0, \delta, H)$, where

S is a set of “states” of the DFA,

A is the input alphabet,

s_0 is the start state where a DFA begins execution,

$\delta : S \times A \rightarrow S$ is the state transition function,

and $H \subseteq S$ is a set of “accept” states.

IMPLEMENTATION OF A DFA

Algorithm to simulate a DFA that returns “yes” if the input string is accepted, else returns “no”.

Algorithm DFA

1. $s = s_0$;
2. $c = \text{getchar}()$;
3. while ($c \neq \text{EOF}$) do
 - (a) $s = \text{move}(s, c)$;
 - (b) $c = \text{getchar}()$;
4. if ($s \in F$) then return “yes”;
else return “no”;

Exercise: Give an example implementation of the `move()` function.

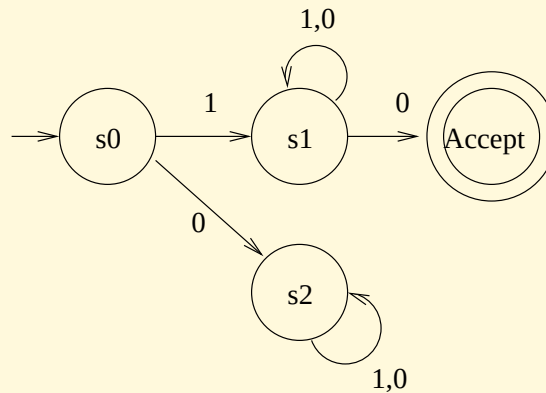
NON-DETERMINISM

Usually, DFAs cannot be directly constructed from specifications of regular expressions. DFAs are obtained from regular expressions via two kinds of non-deterministic state machines.

Non-determinism in transitions creates Non-deterministic Finite Automaton (NFA), which is easier to construct, but is no more expressive than a DFA.

Non-determinism in input alphabet creates NFA with ϵ moves (ϵ -NFA), which is useful for composition of state machines, but is no more expressive than a DFA.

NFA: EXAMPLE



The above state machine is an NFA having a non-deterministic transition at state s_1 . On reading 0 it may either stay in s_1 or go to accept. Note the natural way in which this NFA can be built from its regular expression $1(1 + 0)^*0$.

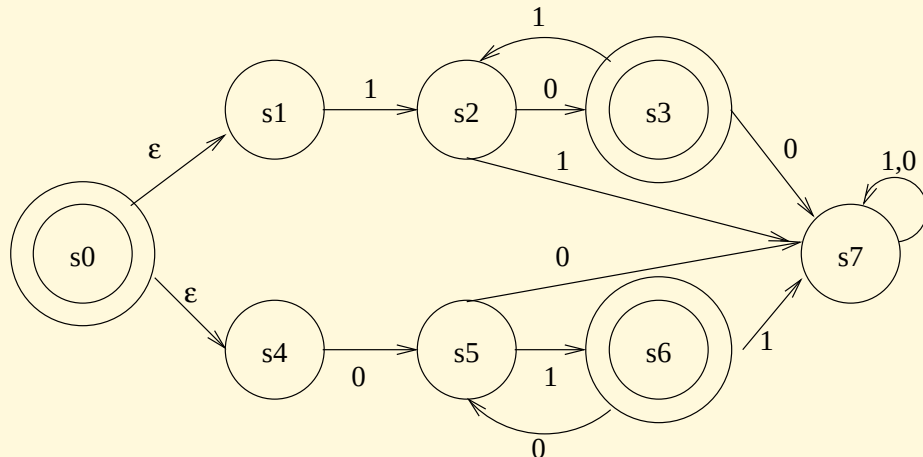
NFA: DEFINITION

A Non-deterministic Finite Automaton (NFA) is defined as $NFA = (S, A, s_0, \delta, H)$ where S is a set of states, A is the input alphabet, s_0 is the start state, $\delta : S \times A \rightarrow 2^S$ is the transition function, and $H \subseteq S$ is the set of accept states.

2^S is the *power set* of S , that is, the set of all possible subsets of S . The transition function maps a state and input pair to a set of possible states instead of a single state.

Example: In the previous slide, transition at state s_1 on reading a 0 is defined as $(s_1, 0, \{s_1, Accept\})$.

NFA WITH ϵ MOVES: ϵ -NFA



The above figure shows a state machine with ϵ moves that is equivalent to the regular expression $(10)^* + (01)^*$. The upper half of the state machine recognizes $(10)^*$ and the lower half recognizes $(01)^*$. The machine non-deterministically moves to the upper half or the lower half when started from state s_0 . Note the natural translation from the regular expression.

Home Page

Title Page

Contents



Page 18 of 55

Go Back

Full Screen

Close

Quit

REDUCIBILITY

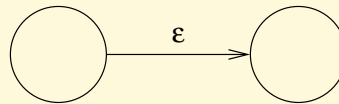
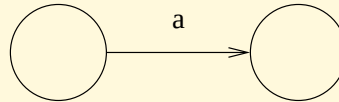
Regular Expression $\xrightarrow{\text{reducible to}}$ ϵ -NFA

ϵ -NFA $\xrightarrow{\text{reducible to}}$ NFA

NFA $\xrightarrow{\text{reducible to}}$ DFA

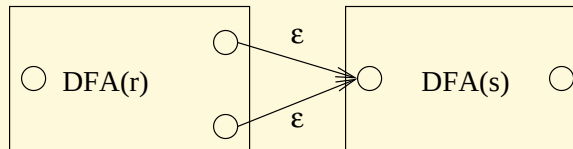
REGULAR EXPRESSIONS TO ϵ -NFA

Given an alphabet A , any $a \in A$ including ϵ is a regular expression.



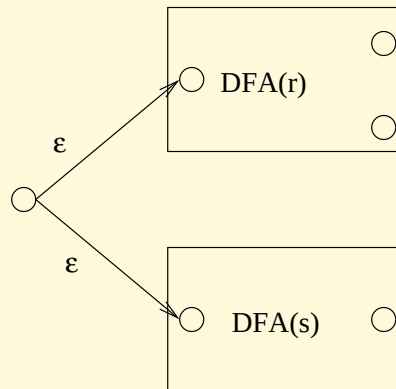
REGULAR EXPRESSIONS TO ϵ -NFA

If r and s are regular expressions, then so is $r \cdot s$.



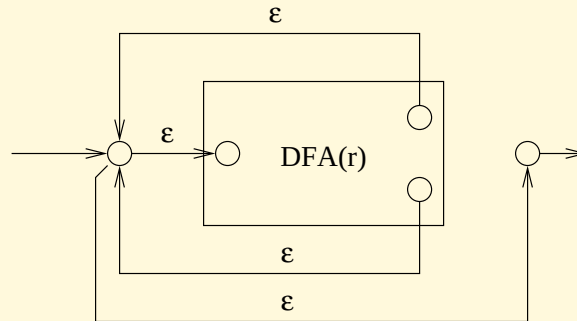
REGULAR EXPRESSIONS TO ϵ -NFA

If r and s are regular expressions, then so is $r + s$.



REGULAR EXPRESSIONS TO ϵ -NFA

If r is a regular expression, then so is r^* .



SOME EXERCISES

Convert the following regular expressions into ϵ -NFAs.

1. $0 + (1 + 0)^*00$

2. **zero** $\rightarrow 0$; **one** $\rightarrow 1$; **bit** $\rightarrow \text{zero} + \text{one}$; **bits** $\rightarrow \text{bit}^*$

3. $(011 + 101 + 110 + 111)^*$

4. $(000 + 111)^* + (101 + 010)^+$

CONVERSION FROM ϵ -NFA TO NFA

Define the operation ϵ -closure as follows:

$$\epsilon\text{-closure}(s) = \{s' \mid s \xrightarrow{\epsilon} s'\}$$

Given an ϵ -NFA $(S', A \cup \{\epsilon\}, s_0, \delta, H)$, construct an equivalent NFA (S, A, s_0, δ^*, H) without ϵ moves as follows:

$$\delta^*(s, a) = \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(s), a)).$$

ALGORITHM FOR ϵ -NFA TO NFA

1. Initially set $S' = \epsilon\text{-closure}(s_0)$ and all are “unmarked”
2. While there is an unmarked state t in S' do
 - (a) mark t
 - (b) for each input symbol $a \in A$ do
 - i. $u = \epsilon\text{-closure}(\delta(t, a))$
 - ii. If u is not in S' then add u as an unmarked state in S'
 - iii. $\delta^*(t, a) = u$

Exercise: Write an algorithm for computing ϵ -closure of any state s in an ϵ -NFA.

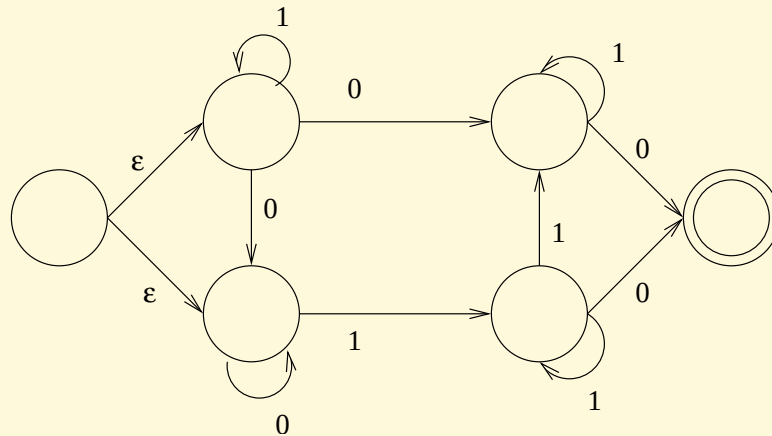
CONVERSION FROM NFA TO DFA

Conversion from NFA to DFA also utilizes a closure property. If $\delta(s, a) = \{s_1, s_2, \dots, s_n\}$ in the NFA, then the set $\{s_1, s_2, \dots, s_n\}$ is considered as a single state in the DFA.

1. Initially set $S' = s_0$ and is “unmarked”
2. While there is an unmarked state t in S' do
 - (a) mark t
 - (b) for each input symbol $a \in A$ do
 - i. $u = \delta(t, a)$
 - ii. If u is not in S' then add u as an unmarked state in S'
 - iii. $\delta^*(t, a) = u$

A SMALL EXERCISE

Convert the following NFA into a DFA.



SPACE-TIME COMPLEXITIES

Given a regular expression r and an input string x , the complexities of an NFA and a DFA creation using the previous algorithms are as follows:

NFA: $O(|r|)$ space complexity; $O(|r| \times |x|)$ time complexity

DFA: $O(2^{\|r\|})$ space complexity; $O(|x|)$ time complexity

OPTIMIZATION OF DFAS

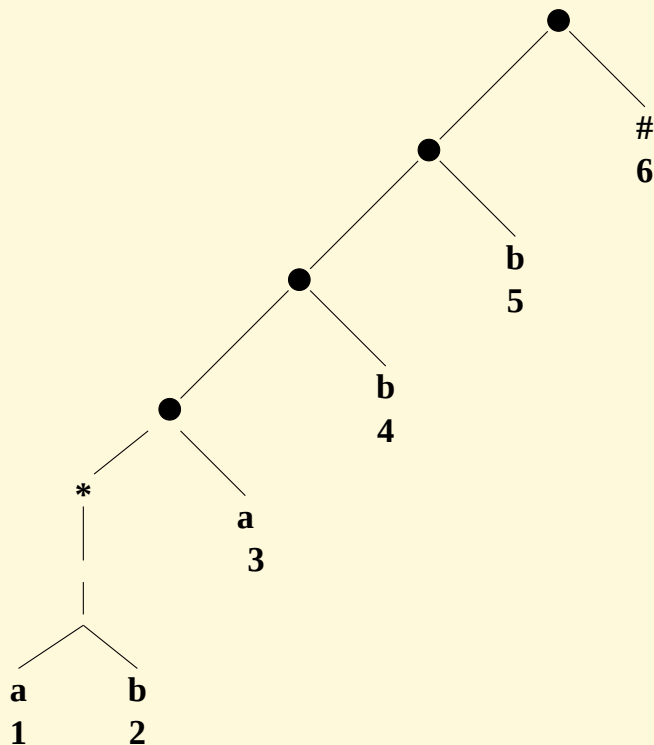
Important states: Any states in an ϵ -NFA that can at least one non-null transition is called an “important” state.

In an optimal state machine all states are important states.

Augmented regular expression: Since the accepting state does not have any transitions, we introduce an “augmented” character # from the accepting state. This classifies the accepting state as an important state. The regular expression $(r)\#$ is called the augmented regular expression of the original expression r .

SYNTAX TREE OF A REGULAR EXPRESSION

Consider the (augmented) regular expression $(a \mid b)^*abb\#$. Figure below shows a “syntax tree” for the regular expression.



A leaf node in the tree represents a symbol and a non-leaf node represents a regular operator $\mid$, $*$ or $\cdot$.

Every non- ϵ leaf node is provided a number which is called the “position” of the leaf (in a resulting state machine)

[Home Page](#)

[Title Page](#)

[Contents](#)

[◀◀](#)

[▶▶](#)

[◀](#)

[▶](#)

Page 30 of 55

[Go Back](#)

[Full Screen](#)

[Close](#)

[Quit](#)

CONVERSION FROM REGULAR EXPRESSION TO DFA

Define the following functions:

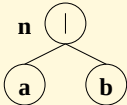
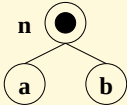
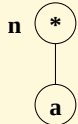
firstpos(n) Where n is any node in the syntax tree, $firstpos(n)$ is the set of all positions that can match the first symbol of a substring rooted at n .

lastpos(n) Same as above, except $lastpos(n)$ matches the set of all positions that can match the last symbol of a substring rooted at n .

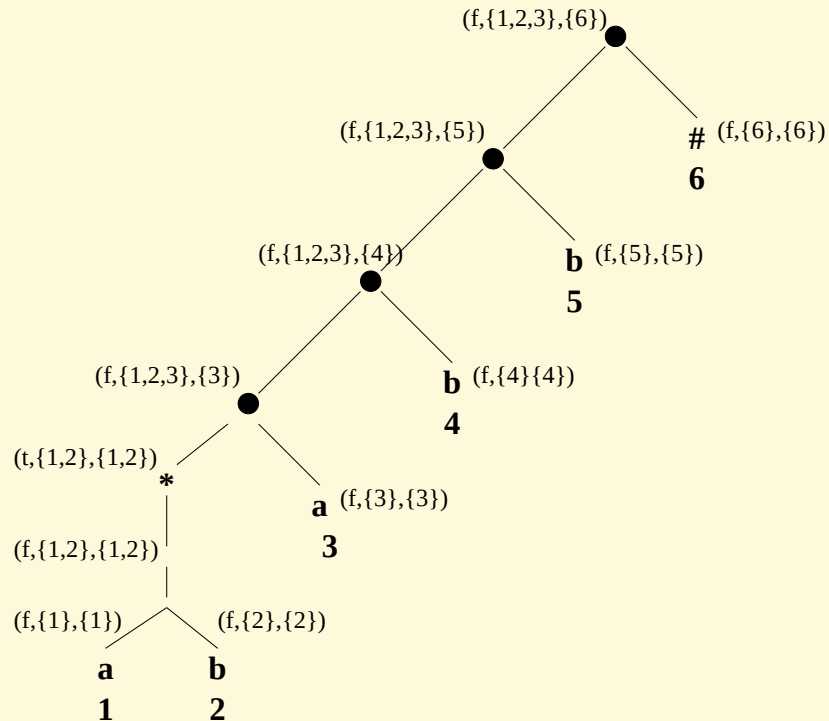
nullable(n) is true if ϵ is a valid string of the subtree under n .

followpos(i) Where i is a position, $followpos(i)$ is the set of all positions that can follow position i for any valid string of the regular expression.

RULES FOR CALCULATING NULLABLE(), FIRSTPOS() AND LASTPOS()

n	nullable(n)	firstpos(n)	lastpos(n)
n is a leaf node containing null	true	{}	{}
non-null leaf node	false	{i} (position of leaf node)	{i}
	nullable(a) or nullable(b)	firstpos(a) U firstpos(b)	lastpos(a) U lastpos(b)
	nullable(a) and nullable(b)	if (nullable(a)) then firstpos(a) U firstpos(b) else firstpos(a)	if (nullable(b)) then lastpos(b) U lastpos(a) else lastpos(b)
	true	firstpos(a)	lastpos(a)

EXAMPLE OF NULLABLE(), FIRSTPOS() AND LASTPOS()



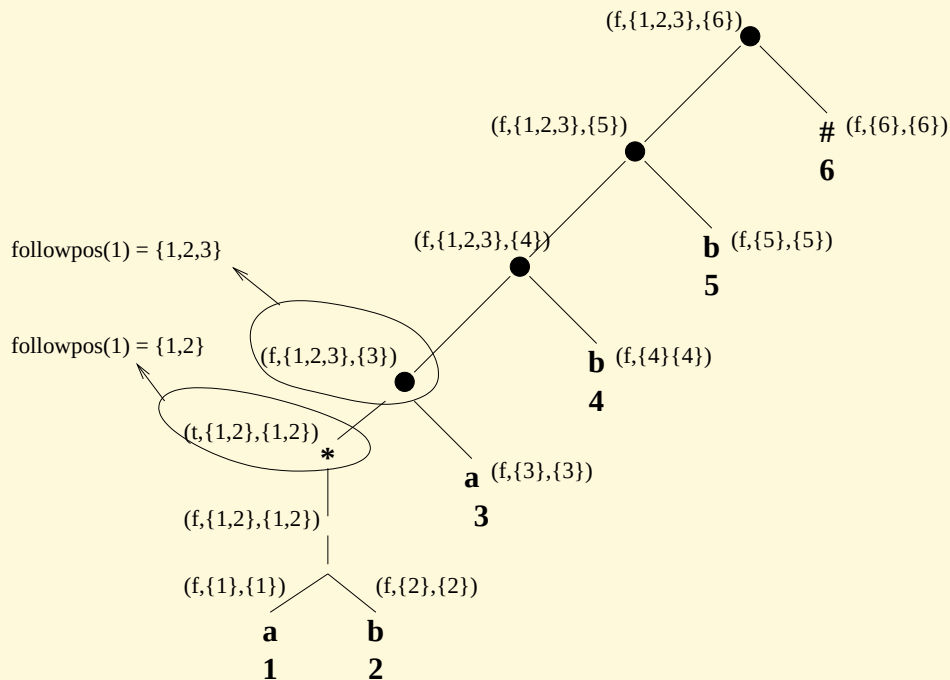
(nullable(), firstpos(), lastpos())

RULES FOR CALCULATING FOLLOWPOS()

For any position i , $followpos(i)$ is calculated as follows:

1. If n is a concatenation node $n = a \cdot b$, and i is in $lastpos(a)$ then all positions in $firstpos(b)$ are in $followpos(i)$
2. If n is a star node $n = a^*$, and i is a position in $lastpos(n)$, then all positions in $firstpos(n)$ are in $followpos(i)$

EXAMPLE FOR CALCULATING FOLLOWPOS()



Home Page

Title Page

Contents



Page 36 of 55

Go Back

Full Screen

Close

Quit

FOLLOWPOS() VALUES

$\text{followpos}(1) = \{1,2,3\}$

$\text{followpos}(2) = \{1,2,3\}$

$\text{followpos}(3) = \{4\}$

$\text{followpos}(4) = \{5\}$

$\text{followpos}(5) = \{6\}$

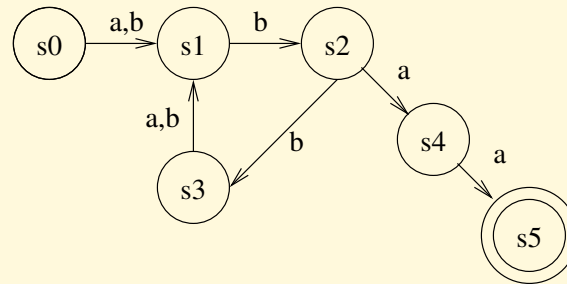
$\text{followpos}(6) = -$

CONSTRUCTION OF A DFA USING FOLLOWPOS() VALUES

Given a syntax tree and the nullable(), firstpos() and lastpos() values for all nodes:

1. For the root node r of the syntax tree, put $firstpos(r)$ as the start state in the set of states S of the DFA
2. While there is an unmarked state t in S do
 - (a) mark t
 - (b) foreach input symbol a do
 - i. Let p be all positions in t that marks symbol a . Let u be the set of positions in $followpos(p)$.
 - ii. If u is not in S , add u as an unmarked state into S .
 - iii. $\delta(t,a) = u$

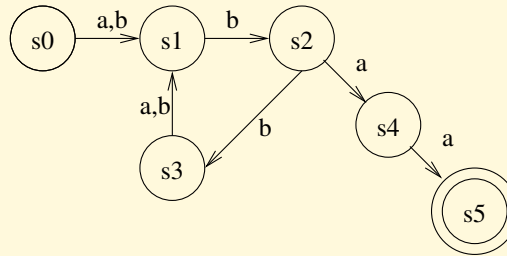
STATE REDUCTION IN A DFA



Groups:

$\{s5\}$ $\{s0, s1, s2, s3, s4\}$

STATE REDUCTION IN A DFA



Groups:

{s5} {s0, s1, s2, s3, s4}

With input char = 'a'

Groups:

{s5} {s4} {s0, s1, s2, s3}

With input char = 'a'

Groups:

{s5} {s4} {s2} {s0,s1,s3}

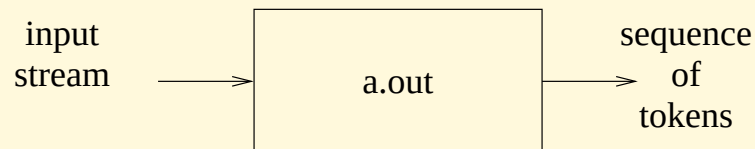
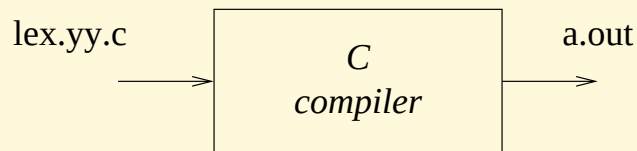
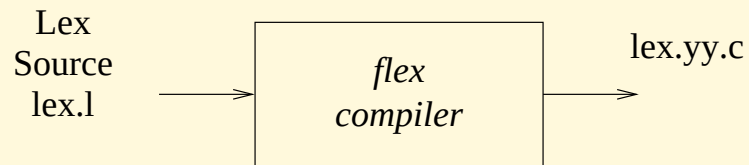
With input char = 'b'

Groups:

{s5} {s4} {s2} {s1} {s0,s3}

FLEX: A LANGUAGE FOR SPECIFYING LEXICAL ANALYZERS

Flex or (*fast lex*) is a later version of the popular lexical analyzer `lex`.



Home Page

Title Page

Contents



Page 41 of 55

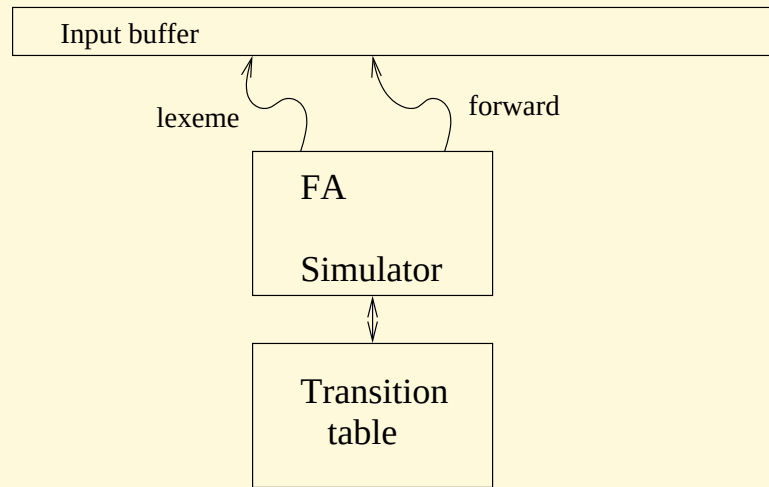
Go Back

Full Screen

Close

Quit

MODEL OF A FLEX COMPILER



Home Page

Title Page

Contents



Page 42 of 55

Go Back

Full Screen

Close

Quit

FORMAT OF A FLEX INPUT FILE

Regular definitions

%%

Action Definitions

%%

Auxiliary functions (user code)

FLEX REGULAR DEFINITIONS

Flex definitions are of the form:

name definition

Every occurrence of *name* is replaced by (*definition*) in the patterns. Example:

DIGIT [0-9]

is a definition. In the pattern, if DIGIT is referred as:

{DIGIT}+

it is replaced by ([0-9])+

FLEX ACTION DEFINITIONS

Action definitions are of the form:

pattern action

where *pattern* must be unindented and *action* must begin on the same line.

Example:

```
if|then|else|for|while|do    {  
                                printf("A keyword: %s\n", yytext);  
                                }
```

Actions may span multiple lines by using braces { and }. If an action statement contains a single | character, it means “same as the action for the next rule.”

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 45 of 55

Go Back

Full Screen

Close

Quit

AUXILIARY FUNCTIONS (USER CODE)

Auxiliary functions are copied verbatim into the `lex.yy.c` output file.

Any declarations enclosed within `%{` and `%}` in a line by themselves in the definition and action section are copied verbatim into the `lex.yy.c` file.

Home Page

Title Page

Contents

◀◀

▶▶

◀

▶

Page 46 of 55

Go Back

Full Screen

Close

Quit

GLOBAL FUNCTIONS AND VARIABLES

yylex() is the scanner function (corresponding to `getnexttoken()`) that can be invoked by the parser

yytext extern char *yytext; is a global char pointer holding the currently matched lexeme.

yylen extern int yylen; is a global int that contains the length of the currently matched lexeme.

SPECIAL ACTION DIRECTIVES

The following special directives can be provided in the actions statements:

ECHO copies yytext to the scanner's output

REJECT directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input)

yyomore() tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of yytext rather than replacing it

yyless(n) returns all but the first n characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match

unput(c) puts the character c back onto the input stream. It will be the next character scanned

input() reads the next character from the input stream

YY_FLUSH_BUFFER flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer

FLEX REGULAR EXPRESSIONS

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.

r* zero or more r's, where r is any regular expression

r+ one or more r's

r? zero or one r's

r{2,5} anywhere from two to five r's

r{2,} two or more r's

r{4} exactly 4 r's

{name} the expansion of the "name" regular definition

rs the regular expression r followed by the regular expression s; called "concatenation"

r|s either an r or an s

Source: flex man pages

FLEX REGULAR EXPRESSIONS

```
[ :alnum:] [ :alpha:] [ :blank:]  
[ :cntrl:] [ :digit:] [ :graph:]  
[ :lower:] [ :print:] [ :punct:]  
[ :space:] [ :upper:] [ :xdigit:]
```

These expressions all designate a set of characters equivalent to the corresponding standard C isXXX function. For example, `[ :alnum:]` designates those characters for which `isalnum()` returns true - i.e., any alphabetic or numeric.

Source: flex man pages

GENERATED SCANNER

Flex generates scanner code in a file called `lex.yy.c` which contains the scanning routine `yylex()`, a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, `yylex()` is declared as follows:

```
int yylex()
{
... various definitions and the actions in here ...
}
```

Whenever `yylex()` is called, it scans tokens from the global input file `yyin` (which defaults to `stdin`). It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions executes a return statement.

If `yylex()` stops scanning due to executing a return statement in one of the actions, the scanner may then be called again and it will resume scanning where it left off.

Home Page

Title Page

Contents



Page 51 of 55

Go Back

Full Screen

Close

Quit

FLEX EXAMPLE

The example below creates a scanner that reads an input file and replaces all occurrences of “hello” with the string “hello world”

```
%%  
hello  printf(''hello world'');
```

Home Page

Title Page

Contents



Page 52 of 55

Go Back

Full Screen

Close

Quit

FLEX EXAMPLE

The example below creates a scanner that reads an input file and trims all white spaces to a single space and removes all trailing white spaces from lines.

```
%%  
[ \t]+          putchar(' ');  
[ \t\n]+$       /*do nothing*/
```

FLEX EXAMPLE

The following program creates a scanner that reads an input file and counts the number of words and lines. (The input file will **not** be echoed back. Why?)

```
int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%

main()
{
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
            num_lines, num_chars );
}
```

FLEX EXAMPLE

The following program creates a scanner that reads an input buffer and recognizes numbers in them and prints their value.

```
%{
/* need this for the call to atof() below */
#include <math.h>
}%

DIGIT      [0-9]

%%

{DIGIT}+    {
              printf( "An integer: %s (%d)\n", yytext,
                      atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*
            {
              printf( "A float: %s (%g)\n", yytext,
                      atof( yytext ) );
            }

%%
main()
{
    yylex();
}
```

[Home Page](#)[Title Page](#)[Contents](#)[◀◀](#)[▶▶](#)[◀](#)[▶](#)[Page 54 of 55](#)[Go Back](#)[Full Screen](#)[Close](#)[Quit](#)

SOME EXERCISES

1. Write a flex program to scan an input C source file and replace all “float” by “double”
2. Write a flex program that scans an input C source file and produces a table of all macros (#defines) and their values.
3. Write a flex program that reads an input file that contains numbers (either integers or floating-point numbers) separated by one or more white spaces and adds all the numbers.
4. Write a flex program that reads an input file that contains numbers (either integers or floating-point numbers) separated by one or more white spaces and prints the minimum and maximum of the numbers.
5. Write a flex program that scans a text file and puts a HTML mailto tag around every email address present in the file.
6. Write a flex program that scans a text file and puts a HTML href tag around every URL present in the file.
7. Write a flex program that scans an input C source file and recognizes identifiers and keywords. The scanner should output a list of pairs of the form (*token*, *lexeme*), where *token* is either “identifier” or “keyword” and *lexeme* is the actual string matched.