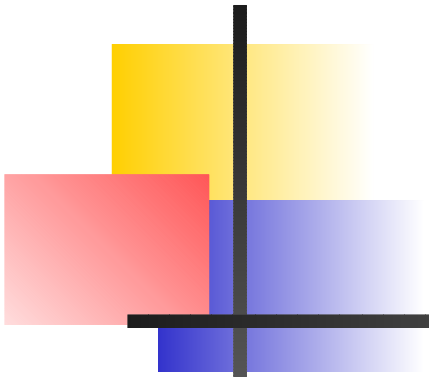


Compilers

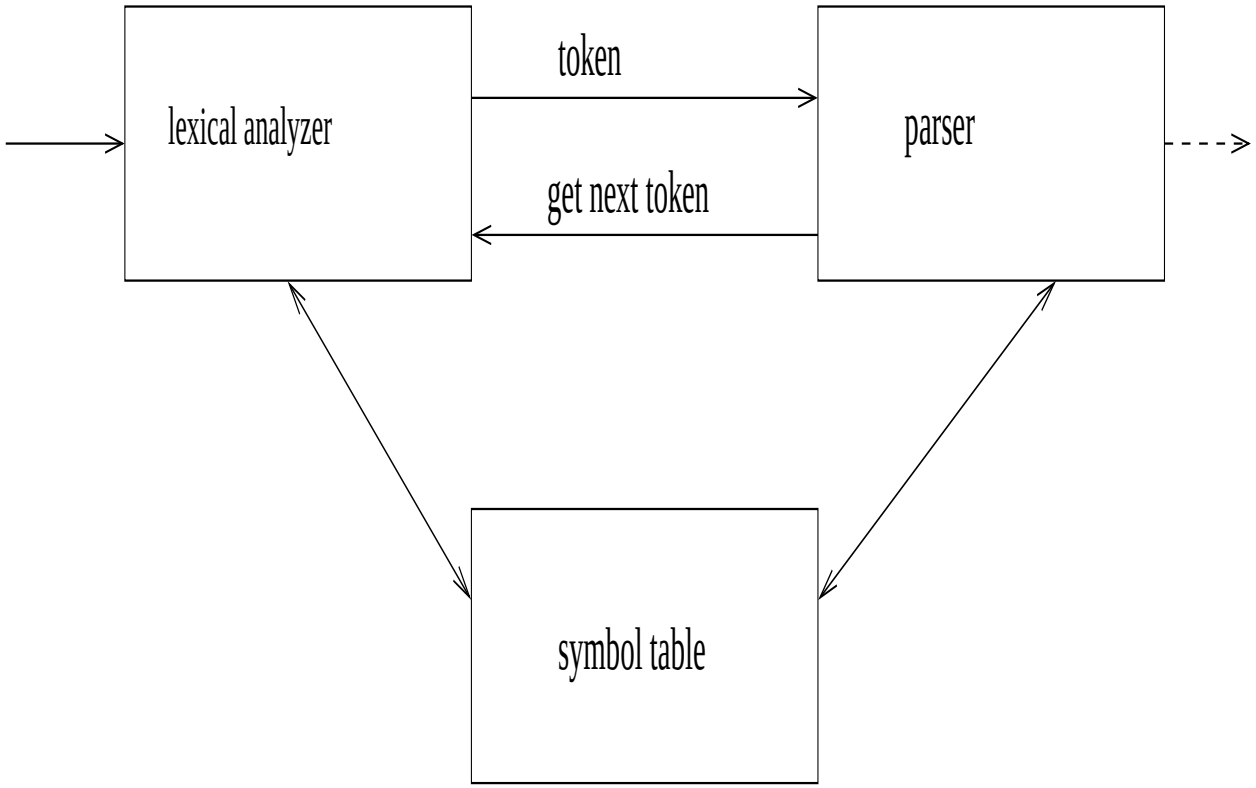
Lexical Analysis

SITE : http://www.info.univ-tours.fr/~mirian/

The Role of the Lexical Analyzer

- The first phase of a compiler.
- **Lexical analysis :** process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called lexical tokens, or just tokens, which may be handled more easily by a parser.
- The lexical analyzer reads the source text and, thus, it may perform certain secondary tasks:
  - Eliminate comments and white spaces in the form of blanks, tab and newline characters.
  - Correlate errors messages from the compiler with the source program (eg, keep track of the number of lines), ...
- The interaction with the parser is usually done by making the lexical analyzer be a sub-routine of the parser.

Interaction of lexical analyzer with parser



Tokens, Patterns, Lexemes

- **Token:** A token is a group of characters having collective meaning: typically a word or punctuation mark, separated by a lexical analyzer and passed to a parser.
- A lexeme is an actual character sequence forming a specific instance of a token, such as `num`.
- **Pattern:** A rule that describes the set of strings associated to a token. Expressed as a regular expression and describing how a particular token can be formed. For example,

  ```
  [A-Za-z][A-Za-z_0-9]*
  ```

- The pattern *matches* each string in the set.
- A lexeme is a sequence of characters in the source text that is matched by the pattern for a token.

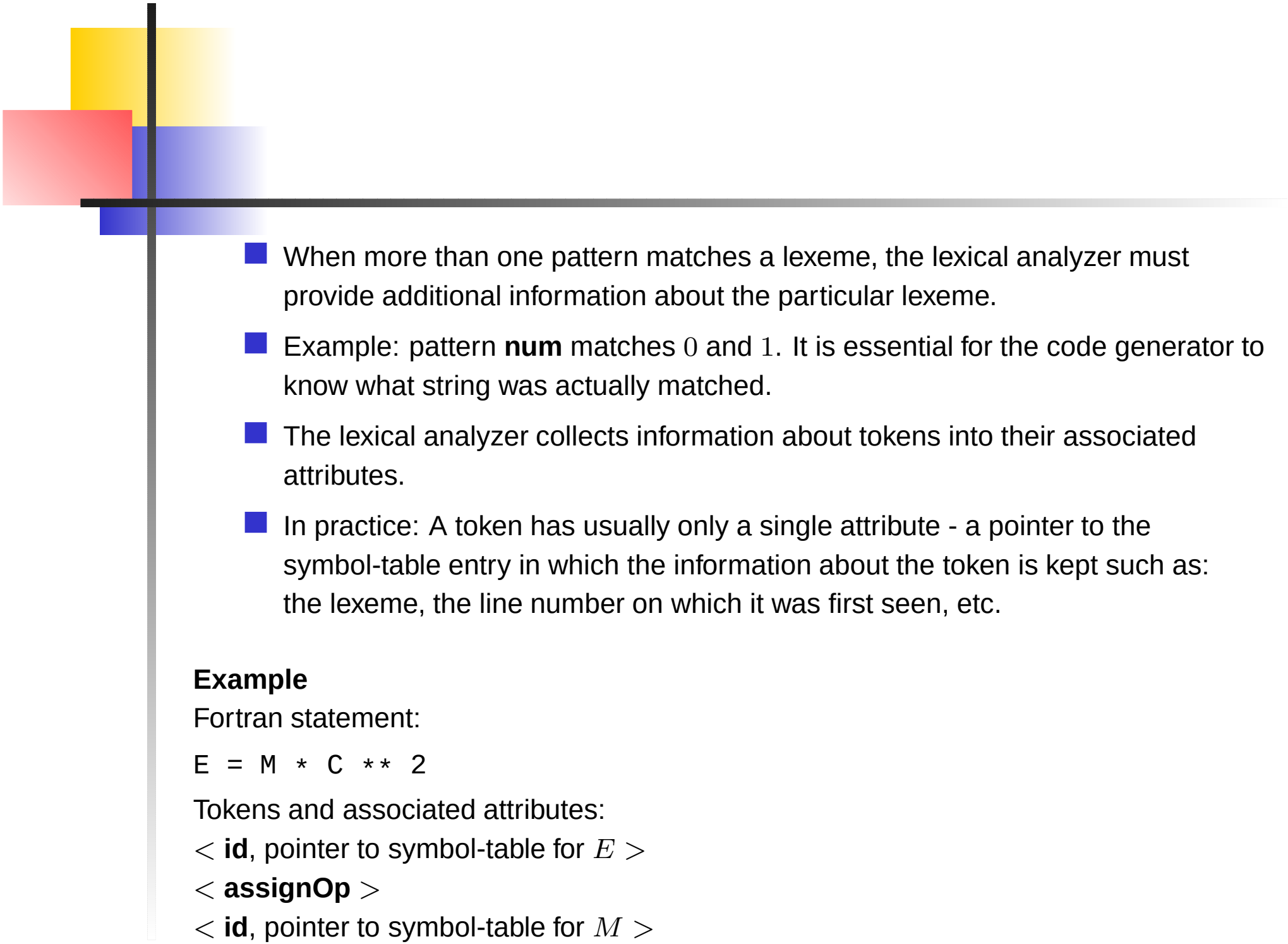
Example

Token	Sample Lexemes	(Informal) Description of Pattern
const	`const`	`const`
if	`if`	`if`
relation	$<, <=, =, <>, >, =>$	$<\|<=\|=\|<>\|>\|=>$
id	`pi, count, D2`	(**letter**.(**letter** | **digit**)$*$)
num	$3.1426, 0.6, 6.22$	any numeric constant
literal	”core dumped”	any character between ” and ” except ”

Note: In the Pascal statement

```
const pi = 3.1416
```

the substring `pi` is a lexeme for the token ”identifier”



- When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme.
- Example: pattern **num** matches 0 and 1. It is essential for the code generator to know what string was actually matched.
- The lexical analyzer collects information about tokens into their associated attributes.
- In practice: A token has usually only a single attribute - a pointer to the symbol-table entry in which the information about the token is kept such as: the lexeme, the line number on which it was first seen, etc.

Example

Fortran statement:

```
E = M * C ** 2
```

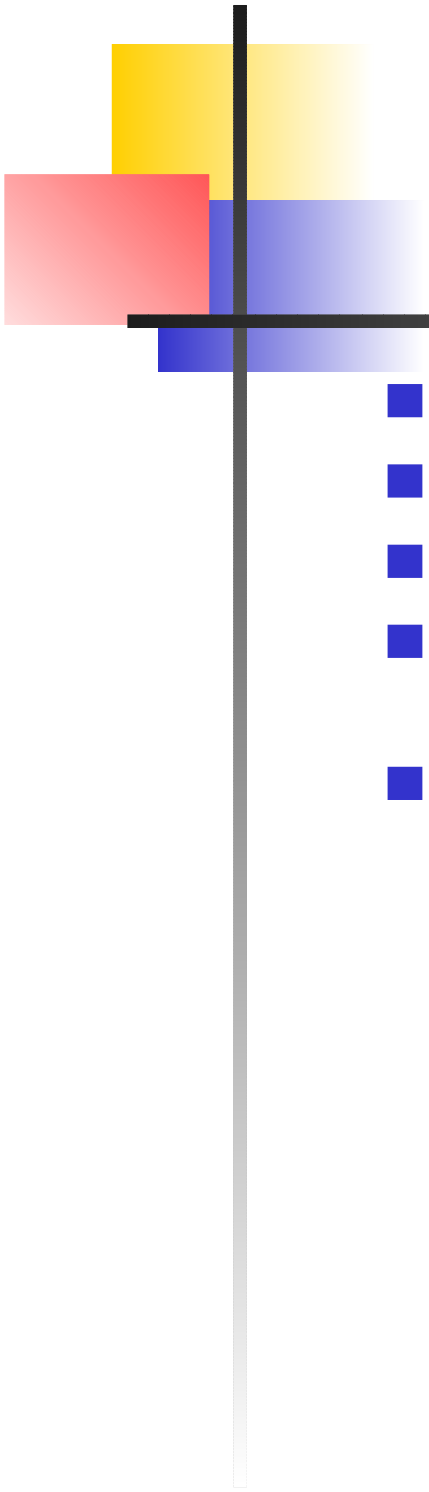
Tokens and associated attributes:

$<$ **id**, pointer to symbol-table for $E>$

$<$ **assignOp** $>$

$<$ **id**, pointer to symbol-table for $M>$

$<$ **multiOp** $>$ ...



Lexical Errors

- Few errors are discernible at the lexical level alone.
- Lexical analyzer has a very localized view of the source text.
- It cannot tell whether a string `fi` is a misspelling of a keyword `if` or an identifier.
- The lexical analyzer can detect characters that are not in the alphabet or strings that have no pattern.
- In general, when an error is found, the lexical analyzer stops (but other actions are also possible).

Stages of a lexical analyzer

Scanner

- Based on a **finite state machine**.
- If it lands on an accepting state, it takes note of the type and position of the acceptance, and continues.
- Sometimes it lands on a "dead state," which is a non-accepting state.
- When the lexical analyzer lands on the dead state, it is done. The last accepting state is the one that represent the type and length of the longest valid lexeme.
- The "extra" non valid character should be "returned" to the input buffer.

Stages of a lexical analyzer

Evaluator

- Goes over the characters of the lexeme to produce a value.
- The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser.
- Some tokens such as parentheses do not really have values, and so the evaluator function for these can return nothing.
- The evaluators for integers, identifiers, and strings can be considerably more complex.
- Sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments.

Example

In the source code of a computer program the string

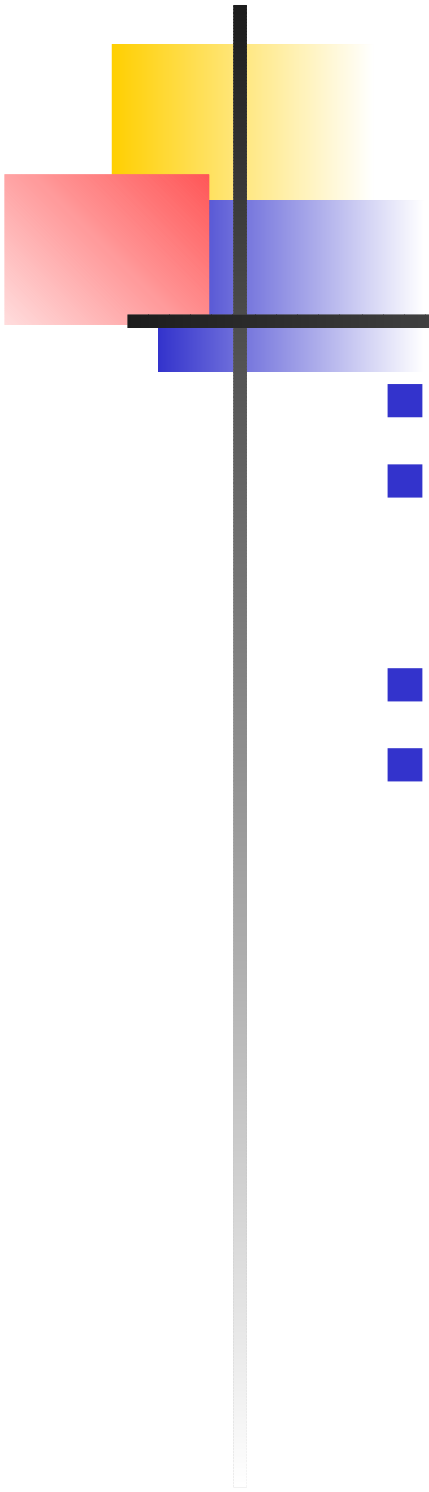
```
net_worth_future = (assets - liabilities);
```

might be converted (with whitespace suppressed) into the lexical token stream:

```
NAME "net_worth_future"
EQUALS
OPEN_PARENTHESIS
NAME "assets"
MINUS
NAME "liabilities"
CLOSE_PARENTHESIS
SEMICOLON
```

General idea of input buffering

- We use a buffer divided into two N-character halves.
- We read N input characters into each half of the buffer with one system read command rather than invoking a read command for each input character.
- If fewer than N characters remain in the input, then a special character **eof** is read.
- Two pointers to the input buffer are maintained.
- The string of characters between the two pointers is the current lexeme.
- Initially both pointers point to the first character of the lexeme to be found.
- One called the *forward pointer* scans ahead until a match for a pattern is found.
- Once the next lexeme is determined, the forward point is set to the first character of it.
- After the lexeme is processed, both pointers are set to the character immediately past the lexeme.
- With this schema, comments and white spaces can be treated as patterns that yield no token.



The importance of the lexical analysis

- Lexical analysis makes writing a parser much easier.
- Instead of having to build up names such as "net_worth_future" from their individual characters, the parser can start with tokens and concern itself only with syntactical matters.
- This leads to efficiency of programming, if not efficiency of execution.
- However, since the lexical analyzer is the subsystem that must examine every single character of the input, it can be a compute-intensive step whose performance is critical, such as when used in a compiler

Regular expressions to the lexical analysis

There is an almost perfect match between regular expressions to the lexical analysis problem with two exceptions:

1. There are many different kinds of lexemes that need to be recognized.
   - The lexer treats these differently so a simple accept reject answer is not sufficient.
   - There should be a different kind of accept answer for each different kind of lexeme.
2. A DFA reads a string from beginning to end then accepts or rejects.
3. A lexer must find the end of the lexeme in the input stream. Then the next time it is called it must find the next lexeme in the string.

Lexical Analyzer Specification

- To specify a lexical analyzer we need a state machine, sometimes called **Transition Diagram** (TD), which is similar to a FSA
- Transition Diagrams depict the actions that take place when the lexer is called by the parser to get the next token
- Differences between TD and FSA
  - FSA accepts or rejects a string. TD reads characters until finding a token, returns the read token and prepare the input buffer for the next call.
  - In a TD, there is no out-transition from accepting states (for some authors).
  - Transition labeled *other* (or not labeled) should be taken on any character except those labeling transitions out of a given state.
  - States can be marked with a $*$: This indicates states on which a input retraction must take place.
- To consider different kinds of lexeme, we usually build separate DFAs (or TD) corresponding to the regular expressions for each kind of lexeme then merge them into a single combined DFA (or TD).

Example

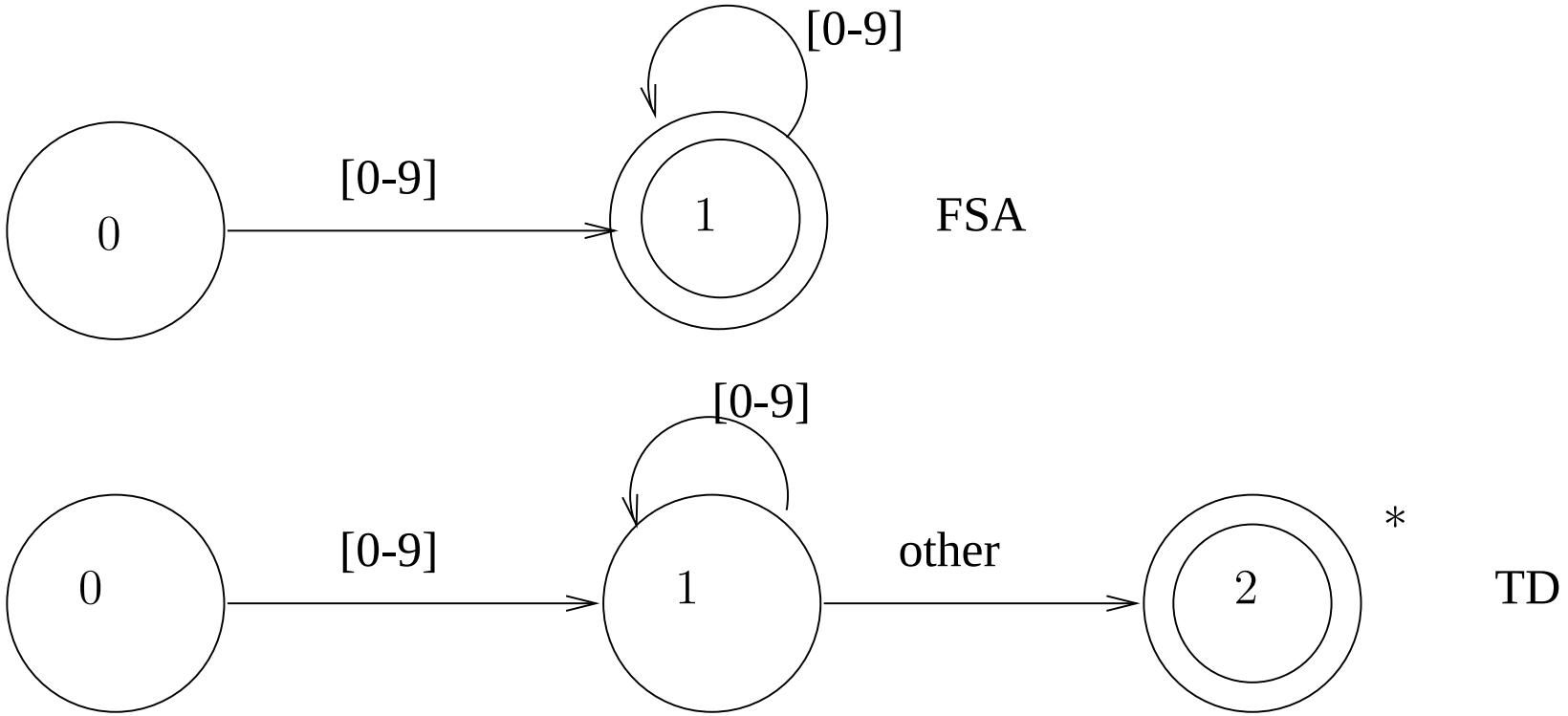


Figure 1: FSA and a TD for integers

Recognizing keywords

- Keywords: same pattern as identifiers but do not correspond to the token ”identifier”.
- Two solutions are possible:
  1. We can consider keywords as identifiers and search in a table to know whether the lexeme is an identifier or a keyword.
  2. We can consider the regular expressions of all keywords and integrate the TD obtained from them into the global TD.

Keywords as identifiers

- This technique is almost essential if the lexer is coded by hand. Without it, the number of states in a lexer for a typical programming language is several hundred (using the trick, fewer of a hundred will probably suffice).
- The technique for separating keywords from identifiers consists in initializing appropriately the symbol table in which information about identifiers is saved.
- For instance, we enter the strings `if`, `then` and `else` into the symbol table **before** any characters in the input are seen.
- When a string is recognized by the TD:
  1. The symbol table is examined
  2. If the lexeme is found there marked as a keyword
     - then the string is a keyword
     - else the string is an identifier

Example

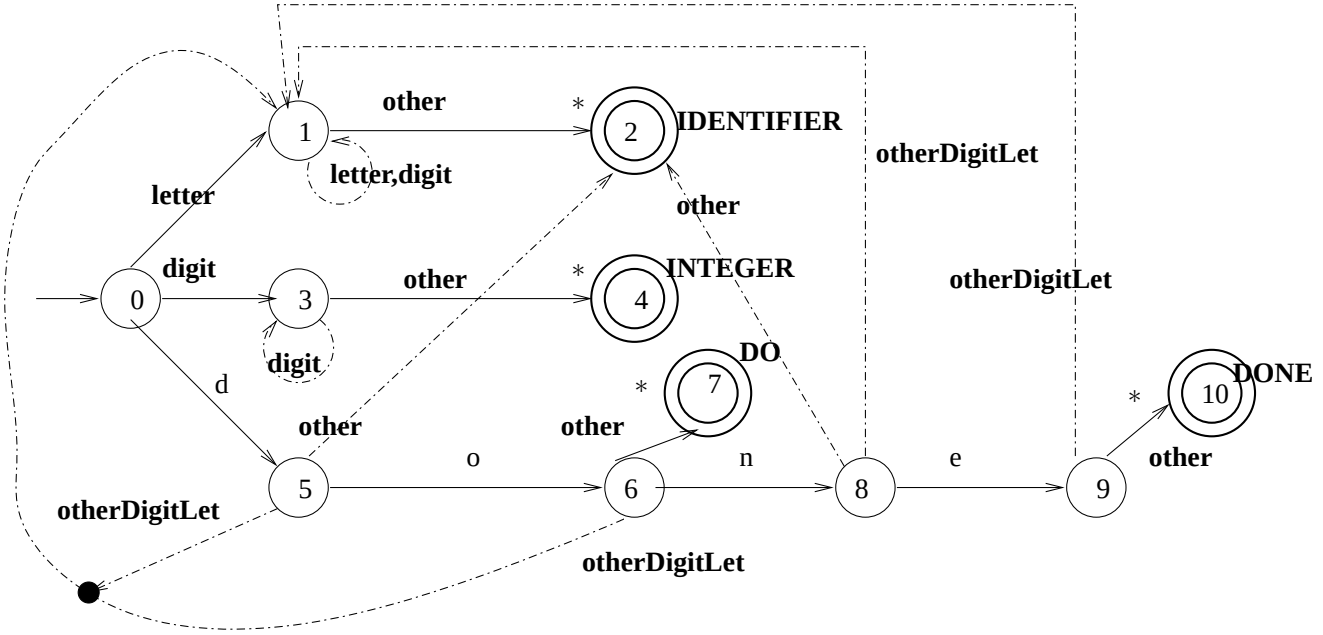
Symbol Table

`do`	keyword
`end`	keyword
`for`	keyword
`while`	keyword
`...`	
Cont	Identifier

↑
Keywords
↓
Identifiers

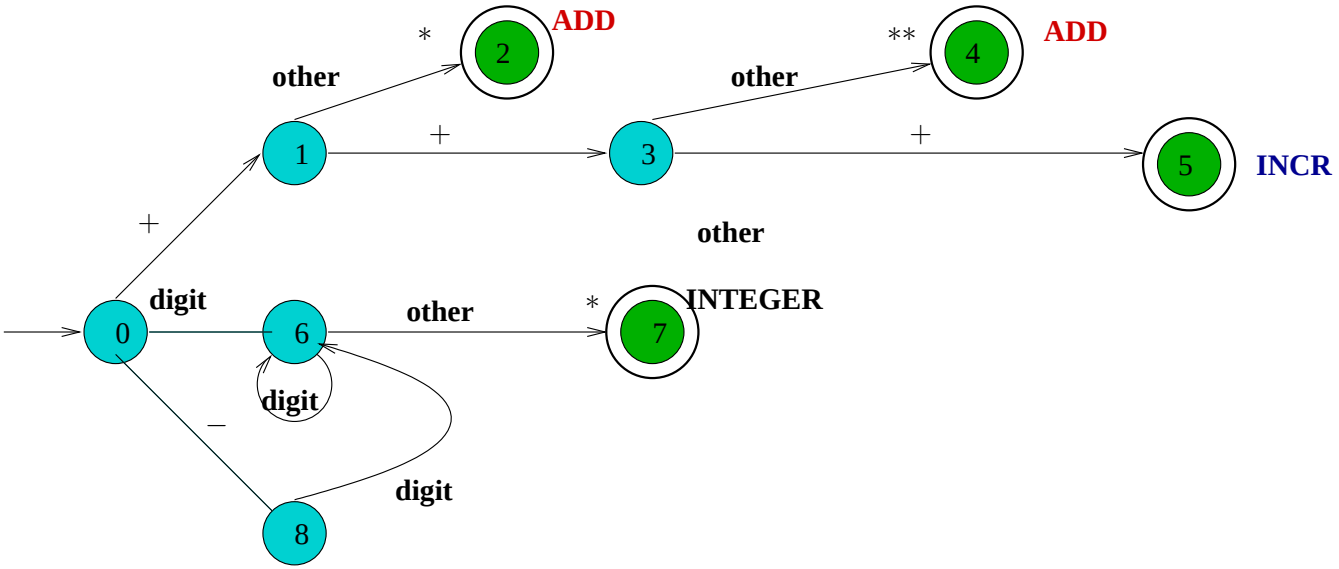
Regular expressions for all keywords

- Keywords can be prefixes of identifiers (ex: do and done).
- The lexer that results from this technique is much more complex but they are necessary when we use lexical analyzer generators from some specification.
- The specification is made by regular expressions.



Example: another transition diagram

TD for unsigned or negative signed integers, addition operation "+" and increment operator "+ + +"



Example

Transition Table

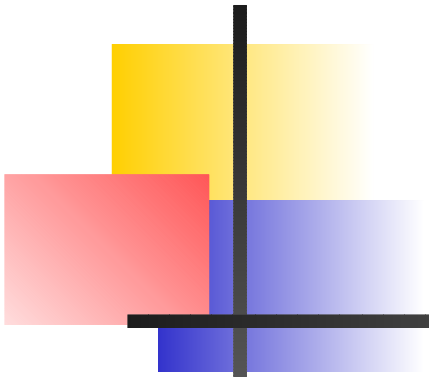
	Input				
state	+	−	D	token	retraction
0	1	8	6	−	−
1	3	2	2	−	−
2	−	−	−	ADD	1
3	5	4	4	−	−
4	−	−	−	ADD	2
5	−	−	−	INCR	0
6	7	7	6	−	−
7	−	−	−	INTERGER	1
8	$error$	$error$	−	−	−

Implementation of Lexical Analyzer

- Different ways of creating a lexical analyzer:
- To use an automatic generator of lexical analyzers (as LEX or FLEX).
- Though it is possible and sometimes necessary to write a lexer by hand, lexers are often generated by automated tools.
- These tools accept regular expressions which describe the tokens allowed in the input stream.
  Input: Specification
  - Regular expressions representing the patterns
  - Actions to take according to the detected token
- Each regular expression is associated with a phrase in a programming language which will evaluate the lexemes that match the regular expression.
- The tool then constructs a state table for the appropriate finite state machine and creates program code which contains the table, the evaluation phrases, and a routine which uses them appropriately

Lexical analyzer: use a generator or construction by hand?

- Lexical analyzer generator
  - Advantages: easier and faster development.
  - Disadvantages: the lexical analyzer is not very efficient and its maintenance can be complicated.
- To write the lexical analyzer by using a high level language
  - Advantages: More efficient and compact
  - Disadvantages: Done by hand
- To write the lexical analyzer by using a low level language
  - Advantages: Very efficient and compact
  - Disadvantages: Development is complicate



Remember

Regular expressions and the finite state machines are not capable of handling recursive patterns, such as n opening parentheses, followed by a statement, followed by n closing parentheses. They are not capable of keeping count, and verifying that n is the same on both sides.

Priority of tokens

- Longest lexeme:
  - DO and DOT (DOT is taken)
  - $>$ and $>=$ ($>=$ is taken)
- First-listed matching pattern
  - The following regular expressions appear in the lexical specification:
    - `w.h.i.l.e`: keyword while
    - **letter**.(**letter** | **digit**): identifier
  - In the input we read *while*
  - The lexer considers it as a keyword
  - If we change the order of the specification then we will never detect a keyword *while*

The use of Lex or Flex

The general form of the input expected by Flex is

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

- The second part MUST be present.

Definitions and Rules

Definitions

- Declarations of ordinary C variables and constants
- Flex definitions

Rules

The form of rules are:

- `regularexpression action`
- The actions are C/C++ code.

Flex actions

Actions are C source fragments. If it is compound, or takes more than one line, enclose with braces ({ }).

```
[a-z]+      printf("found word\n");
[A-Z][a-z]* { printf("found capitalized word:\n");
              printf(" '%s'\n",yytext);
            }
```

Regular-expression like notation

a	represents a single character
$\backslash a$ or "a"	represents a when a is a character used in the notation (to avoid ambiguity)
$a \mid b$	represents a or b
$a?$	represents zero or one occurrence of a
$a*$	represents zero or more occurrence of a
$a+$	represents one or more occurrence of a
$a\{m, n\}$	represents between m and n occurrences of a
$[a - z]$	represents a character set
$[\wedge a - z]$	represents the complement of the first character set

Regular-expression like notation

$\{name\}$	represents the regular expression defined by $name$
$\wedge a$	represents a at the start of a line
$a\$$	represents a at the end of a line
$ab\backslash xy$	represents ab when followed by xy
.	any character except newline

Examples of regular expressions in flex

$a*$	zero or more a's
$.*$	zero or more of any character except newline
$.+$	one or more characters
$[a-z]$	a lowercase letter
$[a-zA-Z]$	any alphabetic letter
$[\wedge a-zA-Z]$	any non-alphabetic character
$a.b$	a followed by any character followed by b
$rs\|tu$	rs or tu
$END\$$	the characters END followed by an end-of-line.
$a(b\|c)d$	abd or acd

Definition

Format

```
name definition
```

- `name` is just a word beginning with a letter (or an underscore) followed by zero or more letters, underscore, or dash.
- `definition` goes from the first non-whitespace character to the end of line.
- You can refer to it via `{name}`, which will expand to (definition)

Examples

```
DIGIT [0-9]

{DIGIT}*\.{DIGIT}+  ou  ([0-9])*\.([0-9])+
```

Example

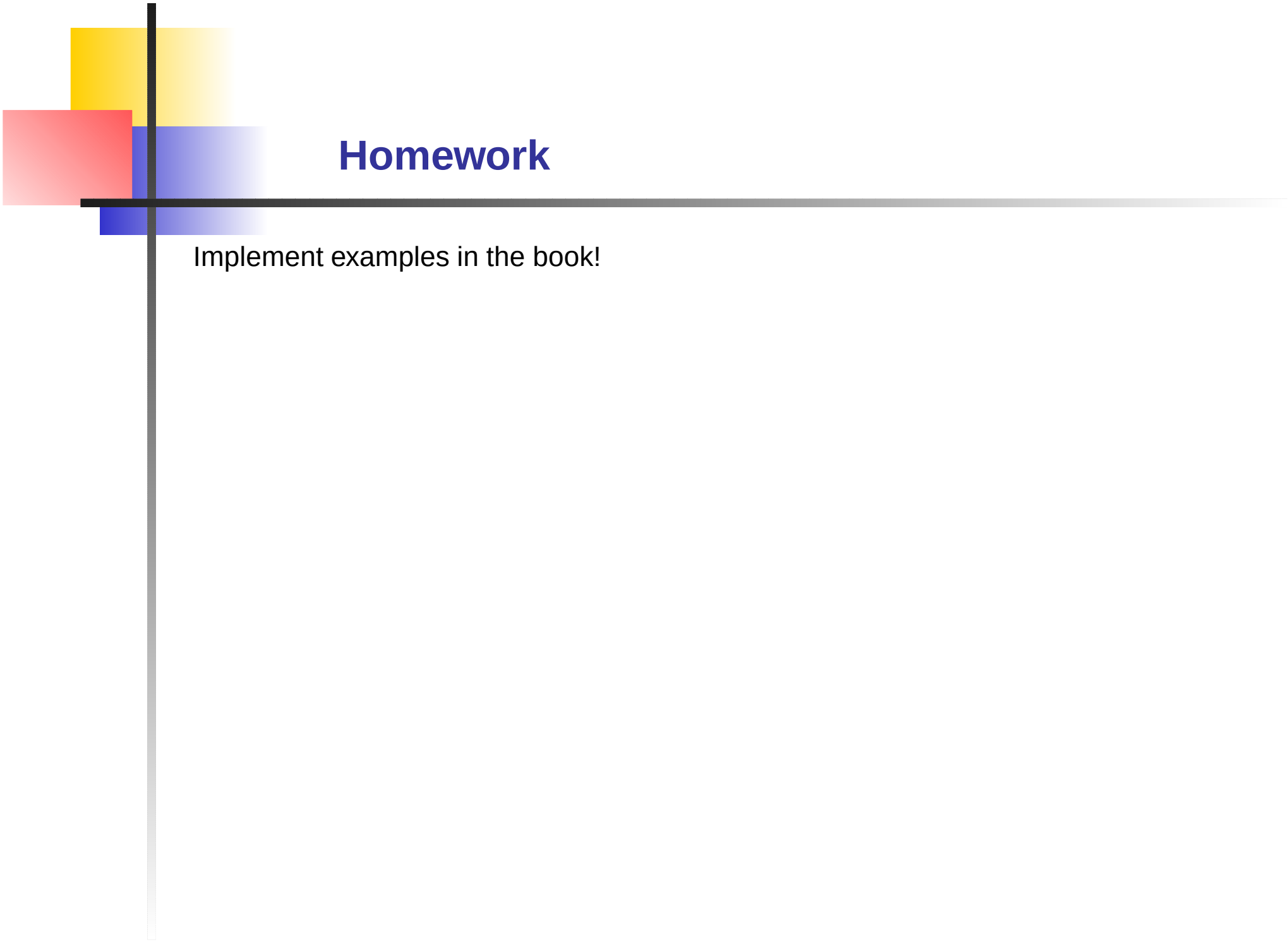
```
%option lex-compat
letter  [A-Za-z]
digit  [0-9]
identifier  {letter}({letter}|{digit})*

%%
{identifier}  {printf("identifier %s  on line %d recognised \n", y

%%
```

Example

```
%option lex-compat
digit  [0-9]
letter  [A-Za-z]
intconst  [+\-]?{digit}+
realconst  [+\-]?{digit}+\.{digit}+(e[+\-]?{digit}+)?
identifier  {letter}({letter}|{digit})*
whitespace  [ \t\n]
stringch  [^’]
string  ’{stringch}+’
otherch  [^0-9a-zA-Z+\-’ \t\n]
othersym  {otherch}+
```

Example

```
%%
main  printf("keyword main - program recognised \n");
\{  printf("begin recognised\n");
for   printf("keyword for  recognised\n");
do   printf("keyword do recognised\n");
while printf("keyword while recognised\n");
switch  printf("keyword  switch recognised\n");
case  printf("keyword case recognised \n");
if  printf("keyword if  recognised \n");
else  printf("keyword else recognised \n");
{intconst}  printf("integer %s on line %d \n", yytext,yylineno);
{realconst}  printf("real  %s on line %d \n", yytext,yylineno);
{string}  printf("string %s on line %d \n", yytext,yylineno);
{identifier}  printf("identifier %s  on line %d recognised \n", yy
{whitespace}  ; /* no action */
{othersym} ; /*no action */
%%
```



Homework

Implement examples in the book!