LEXICAL ANALYSIS

The role of Lexical Analyzer, Input Buffering,

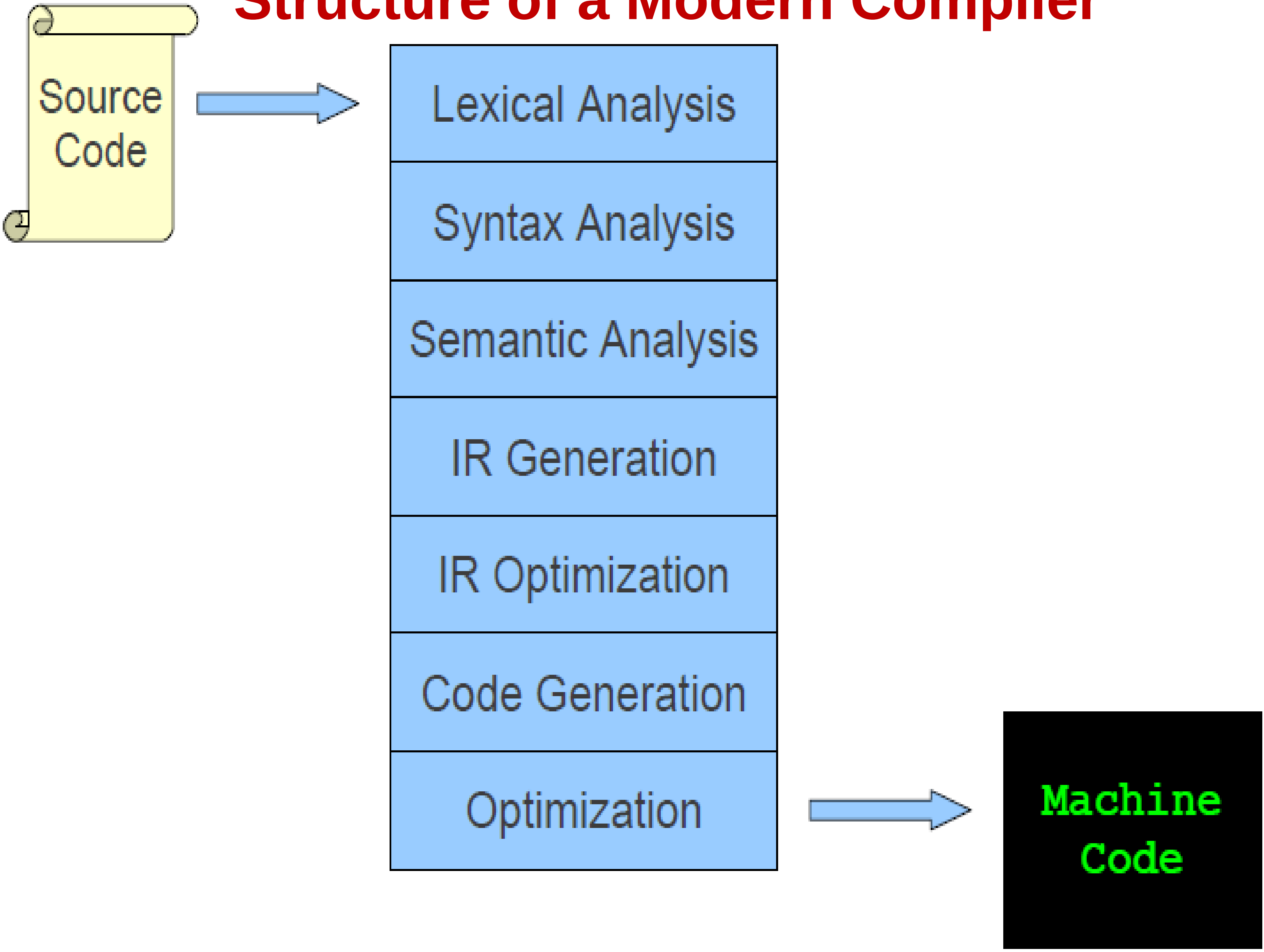
Specification of Tokens using Regular Expressions,

Review of Finite Automata, Recognition of Tokens.

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

AMAL JYOTHI COLLEGE OF ENGINEERING
KANJIRAPPALLY

Structure of a Modern Compiler



Lexical Analysis

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Code Optimization

Consider the following code :

```
while (y < z)        {
        int x = a + b;
        y += x;      }
```

- Reads the characters in the source program and groups them into tokens.
- Token represents an Identifier, or a keyword, a punctuation character or a operator.
- The character Sequence forming a token is called the **lexeme** for the token.
- The lexical Analyzer not only generate tokens but also it enters the **lexeme into the symbol table.**

Outcome of Lexical Analyzer

Token Stream:

```
while (y < z)   {
    int x = a + b;
    y += x;
 }
```

T_While

T_LeftParen

T_Identifier y

T_Less

T_Identifier z

T_RightParen

T_OpenBrace

T_Int T_Identifier x

T_Assign

T_Identifier a

T_Plus

T_Identifier b

T_Semicolon

T_Identifier y

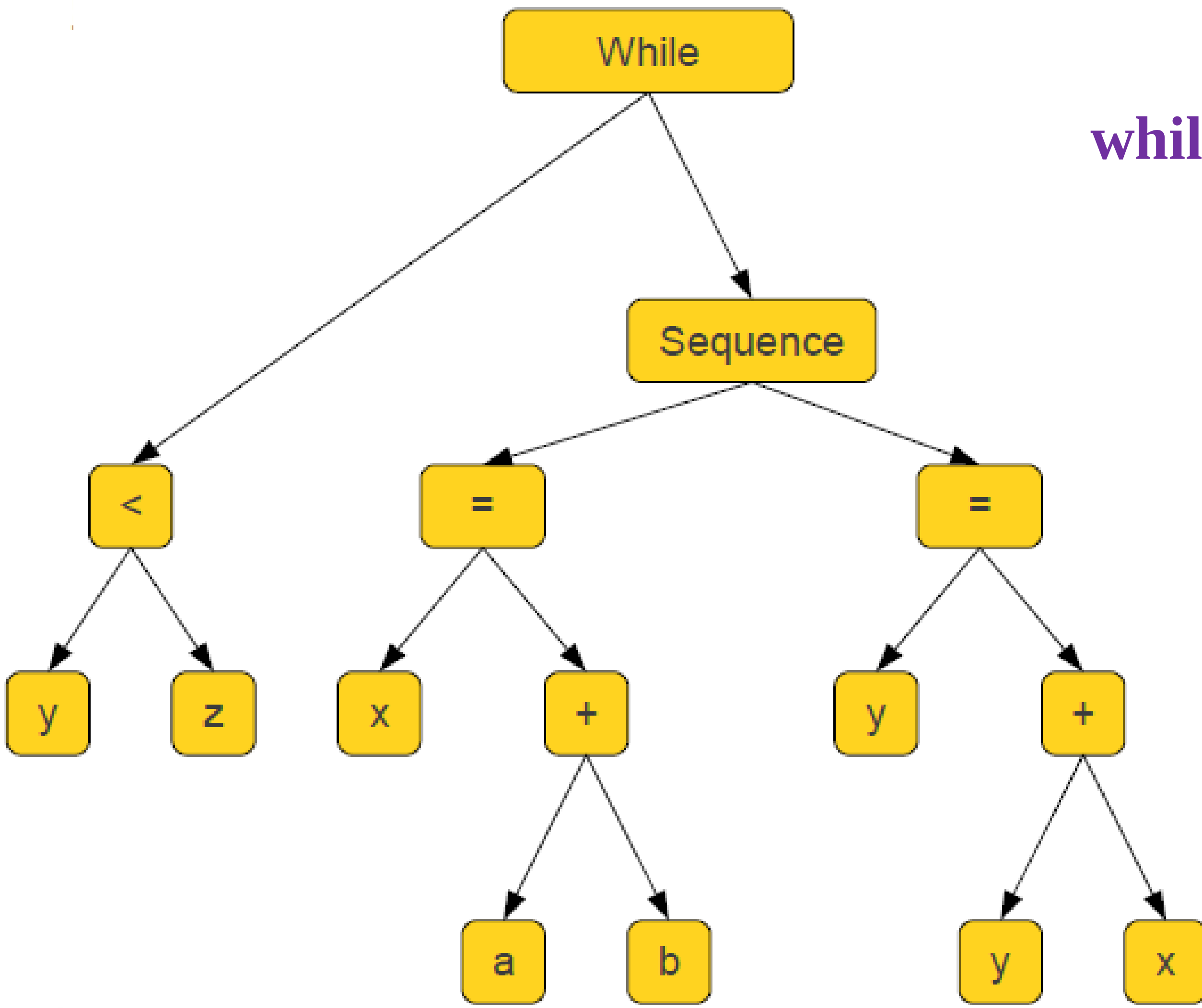
T_PlusAssign

T_Identifier x

T_Semicolon

T_CloseBrace

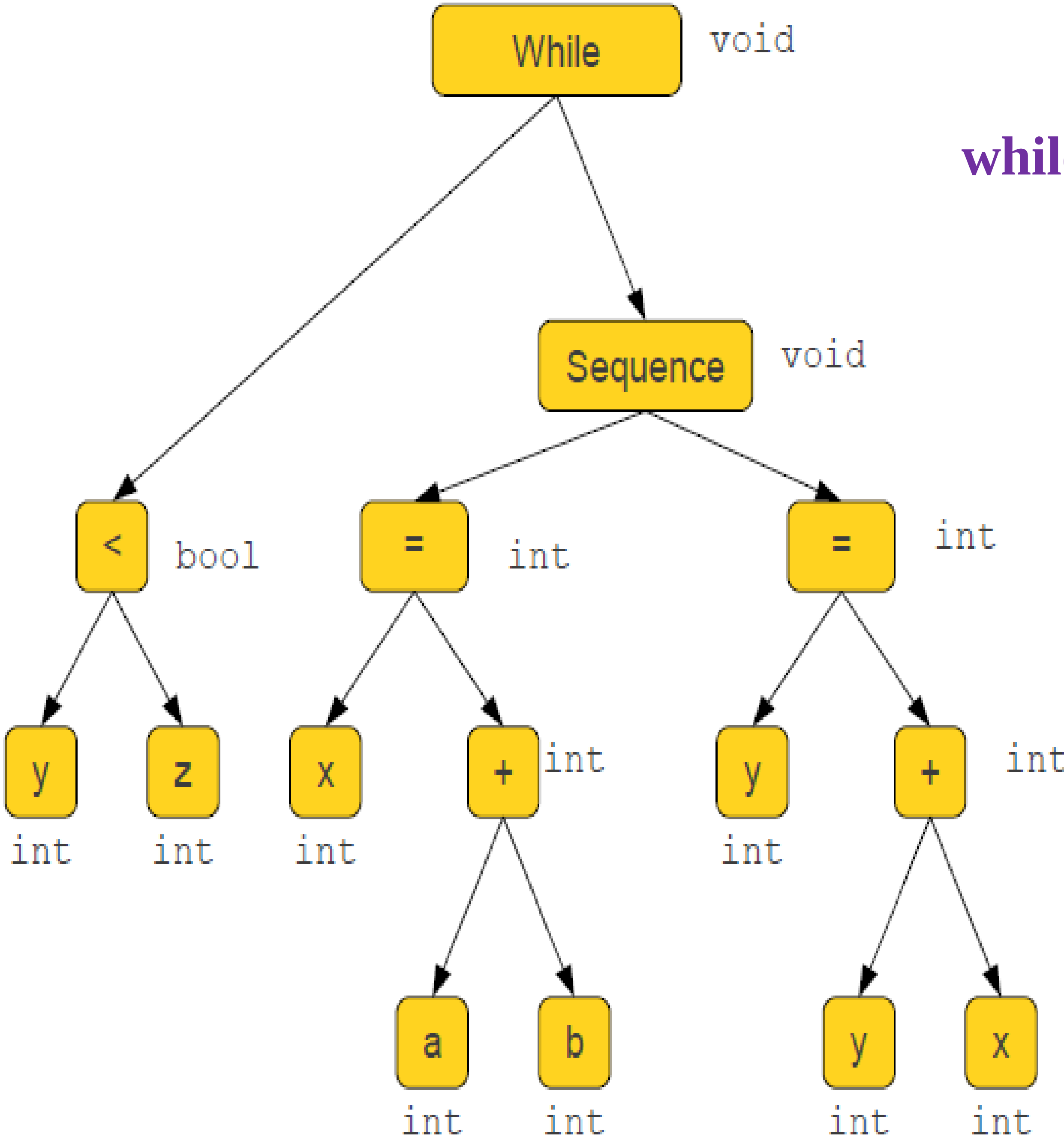
Groups the tokens into Syntactic Structures.



Syntax Tree

```
while (y < z)   {
    int x = a + b;
    y += x;
}
```

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Code Optimization



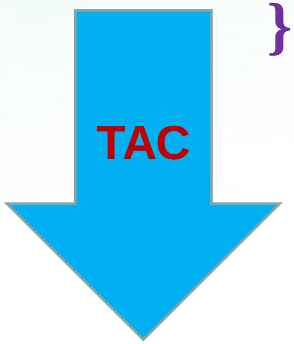

```
while (y < z)
    {
    int x = a + b;
    y += x;
    }
```

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Code Optimization

- Simple Instructions produced by the syntax Analyzer is IR.
- IR has two properties: Easy to use, Easy To translate.
- Eg: TAC (Three Address Code)

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Code Optimization

```
while (y < z)   {
    int x = a + b;
    y += x;
}
```




```
Loop: x = a + b
      y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Code Optimization

It involves:

- Detection and removal of dead code.

-Calculation of constant expressions and terms.

-Moving code outside of loops.

-Removal of unnecessary temporary variables.

•Improve the Intermediate Code so that the ultimate object program runs faster and or takes less space.

```
Loop: x = a + b
      y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

AFTER OPTIMIZATION

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

Machine code is generated. This involves:

*. Allocation of Registers and Memory.

*. Generation of correct References.

*. Generation of correct types.

*. Generation of machine code.

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Code Optimization

```
        x = a + b
Loop:   y = x + y
        _t1 = y < z
        if _t1 goto Loop
```

Code Generation

```
add $1, $2, $3
Loop: add $4, $1, $4
slt $6, $1, $5
beq $6, loop
```

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- **Code Optimization**

```
while (y < z)   {
        int x = a + b;
        y += x;
 }
```

```
add $1, $2, $3
Loop: add $4, $1, $4
slt $6, $1, $5
beq $6, loop
```

Code Optimization

```
add $1, $2, $3
Loop: add $4, $1, $4
blt $1, $5, loop
```

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);
}
```

The same program – as the compiler sees it (initially).

```
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

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

How do you make the compiler see what you see?

Step 1:

a. Break up this string into 'words'–the smallest logical 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*.

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

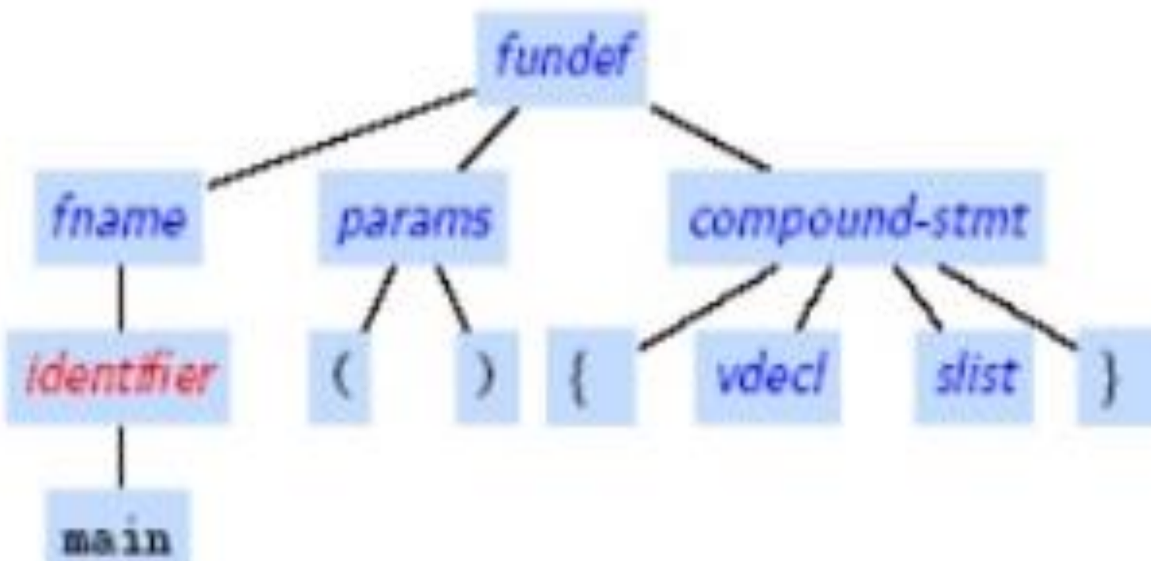
b. Clean up – remove the ␣ and the ↩ characters.

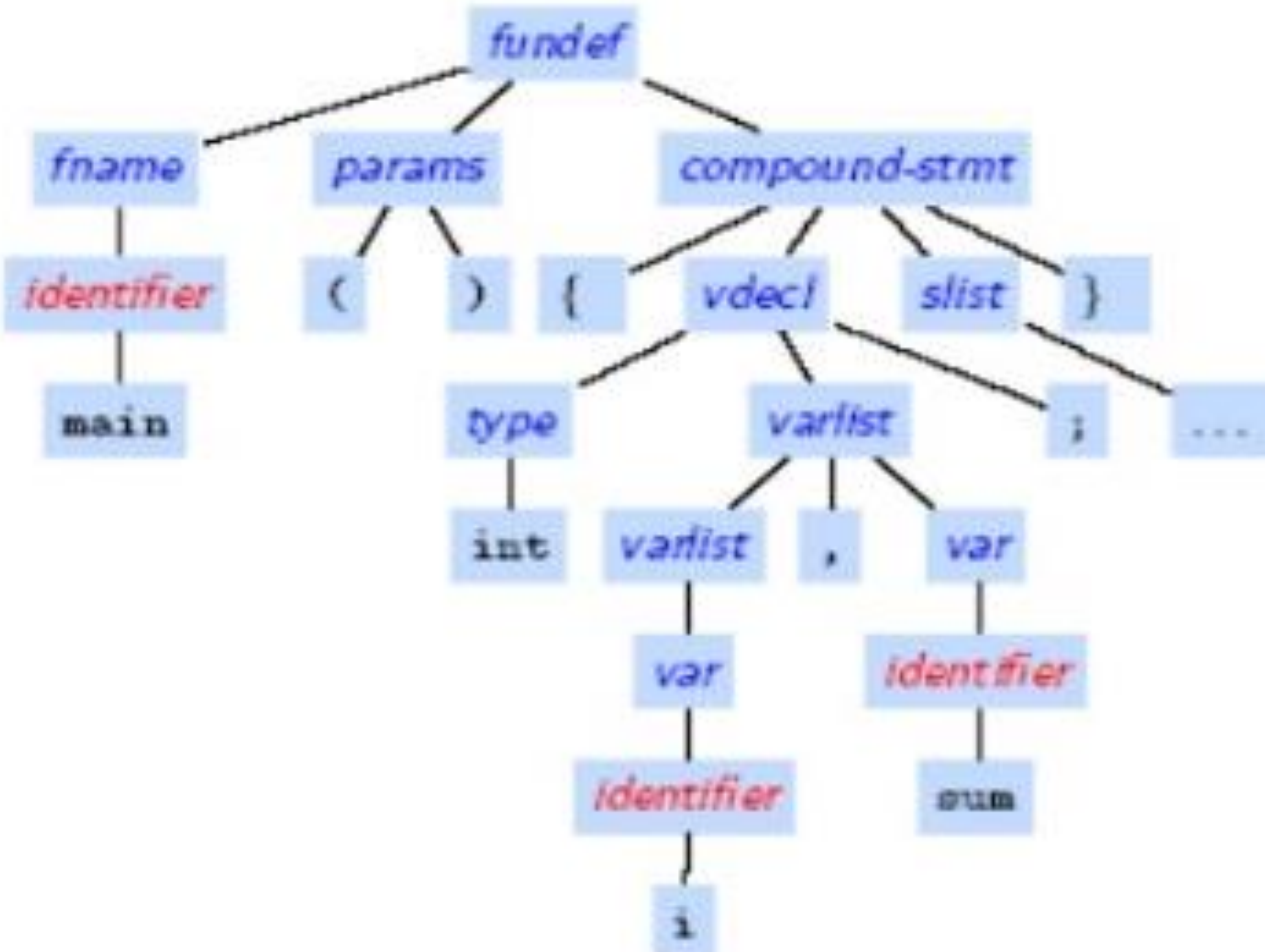
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++) ; sum = sum + i
; printf ("%d\n" , sum) ; }

Step 2:

Now group the lexemes to form larger structures.

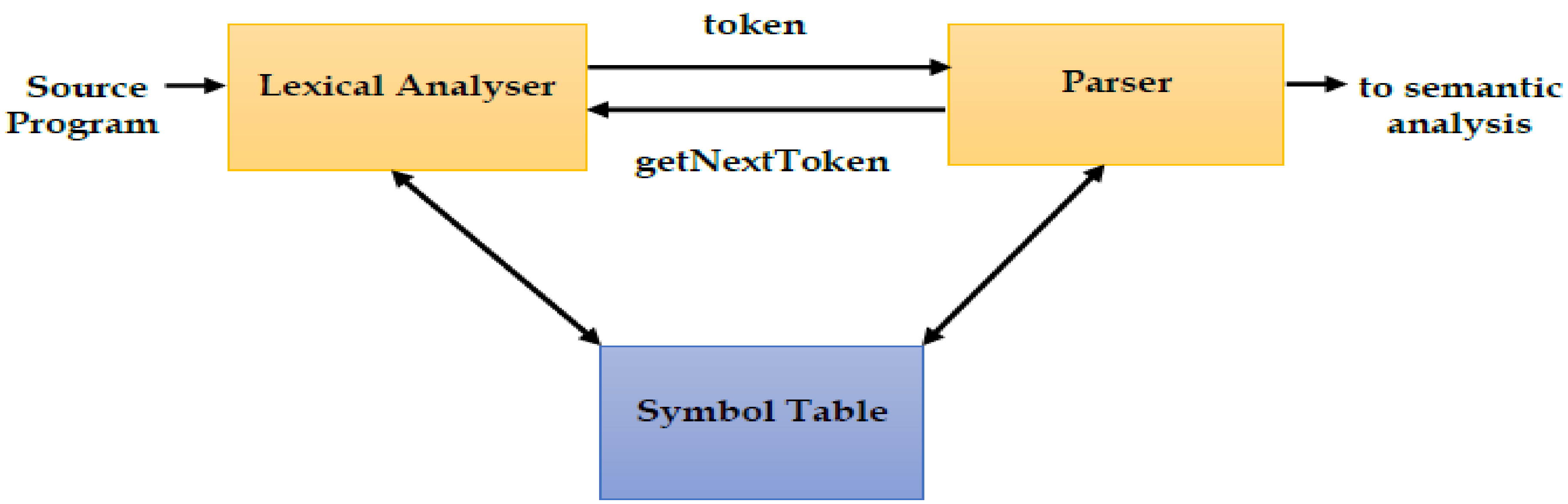
```
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i
; printf ( "%d\n" , sum ) ; }
```



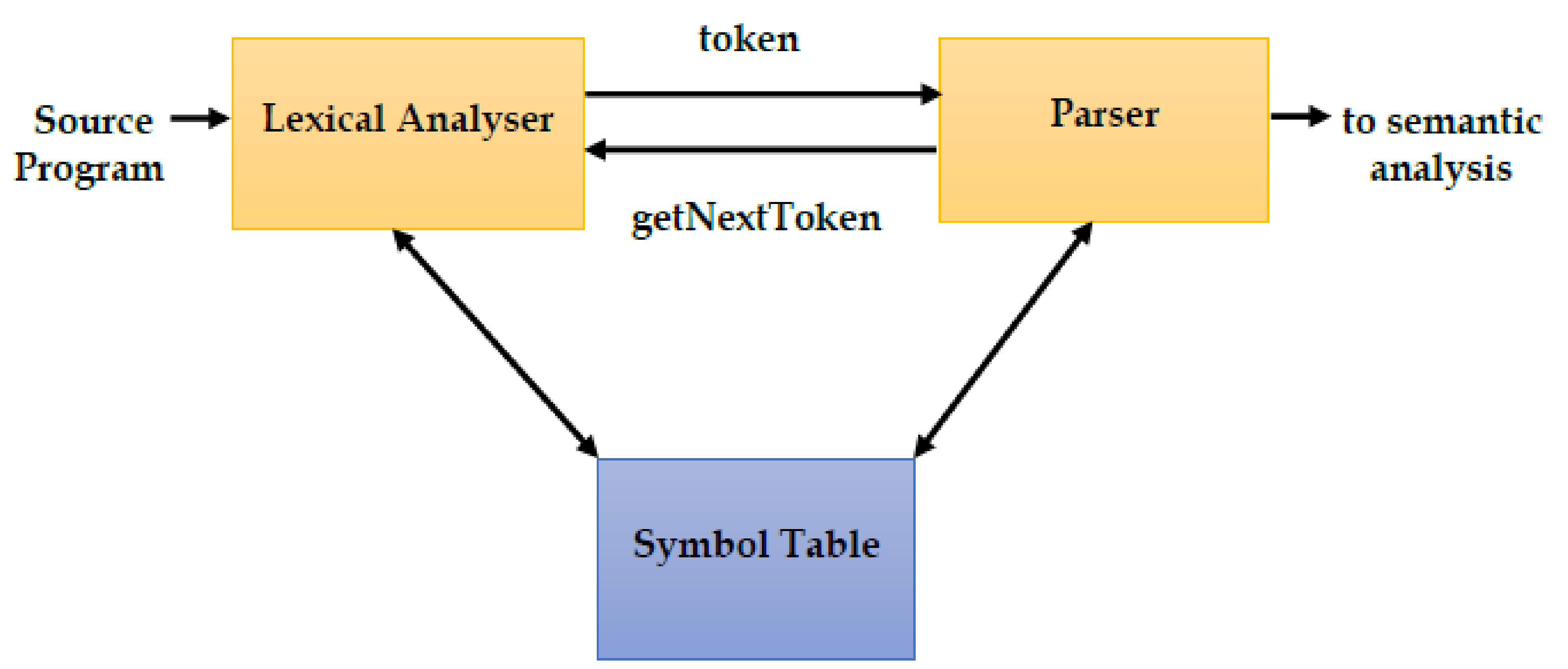


This is *syntax analysis* or *parsing*.

- As the first phase of a compiler, the main task of the lexical analyzer is to
  - read the input characters of the source program,
  - group them into lexemes, and
  - produce as output a sequence of tokens for each lexeme in the source program.



- The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



- **Secondary tasks**
  - Strips comments, white space and new line characters
  - Correlate error messages from the compiler(line no)
  - Eg : LA may keep track of the number of newline characters seen, so that a line no can be associated with error msg

- Maybe split into scanning(simple tasks) and lexical analysis phases(complex jobs)
- After lexical analysis individual characters are no longer examined.

Issues In Lexical Analysis

- Following are the reasons why lexical analysis is separated from syntax analysis

Simplicity of Design

- The separation of lexical analysis and syntactic analysis often allows us to simplify at least one of these tasks.

Eg : A parser including the conventions for comments and white space is significantly more complex

Issues In Lexical Analysis

Efficiency

- Compiler efficiency is improved.
- A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
- In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

Portability

- Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.
- Specialized symbols and characters are isolated in this phase

Attributes For Tokens

- Sometimes a token need to be associate with several pieces of information.
- The most important example is the **token id**, where we need to associate with the token a great deal of information.
- Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table.
- Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

- When more than one pattern matches a lexeme, the LA must provide additional information about the particular lexeme.
  - Eg : If pattern “num” matches both strings 0 and 1, but code generator must know which string was matched.
  - The lexical analyzer collects information about tokens into their associated attributes.
  - For example integer 31 becomes **<num, 31>.**
  - So, the constants are constructed by converting numbers to token 'num' and passing the number as its attribute.
  - Similarly, we recognize keywords and identifiers.
  - For example count = count + inc becomes id = id + id.

Attributes for Tokens

- **Example**: E = M * C ** 2
  - **<id,** pointer to symbol-table entry for E>
  - **<assign_op, >**
  - **<id,** pointer to symbol-table entry for M>
  - **<mult_op, >**
  - **<id,** pointer to symbol-table entry for C>
  - **<exp_op, >**
  - **<num,** integer value 2>

Token has usually only one attribute- a pointer to the symbol table entry in which information about the token is kept.

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

EXAMPLE

lexeme	<token, token attribute>
`3`	< const, 3>
`A`	<identifier, A>
`if`	<if, –>
`=`	<assignop, –>
`>`	<gt, –>
`;`	<semicolon, –>

Lexical Errors

- A character sequence that can't be scanned into any valid token is a lexical error.

Example:

- **int a@d=1;** here a@d is counted as an invalid literal

fi (a == b)

- fi is considered as a valid identifier, but actually it also can be a misspelt if
- Suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
- The simplest recovery strategy is "panic mode" recovery.
- We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

Error Recovery in Lexical Analyzer

- Removes one character from the remaining input
- In the panic mode, the successive characters are always ignored until we reach a well-formed token
- By inserting the missing character into the remaining input
- Replace a character with another character
- Transpose two serial characters

INPUT BUFFERING

- Scanner is the only part of the compiler which reads the complete program. It takes about 30% of compilers time.
- To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.
- Usually scanners use two-buffer scheme to look ahead on the input which is necessary to identify token with minimum time.

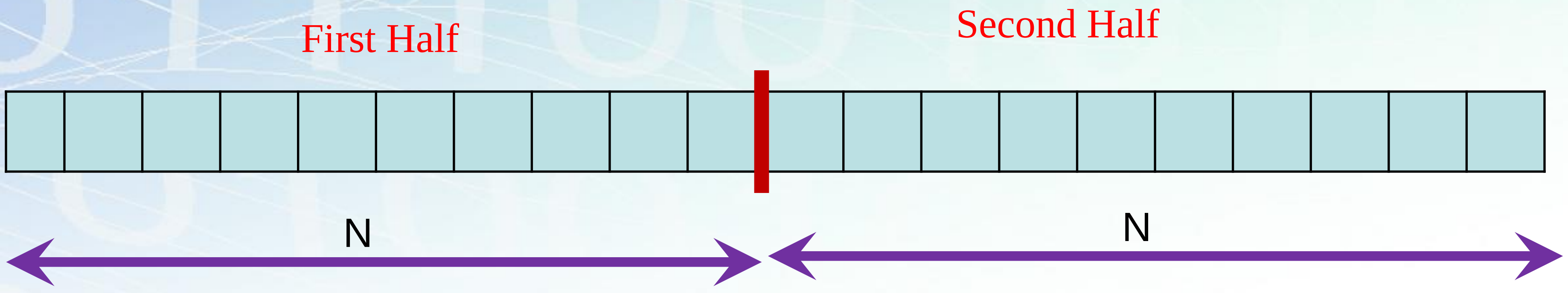
INPUT BUFFERING

- Lexical Analysers will look ahead several characters before a pattern is matched.
- There are two buffer input schemes for Look ahead

1. Buffer Pairs
2. Sentinels

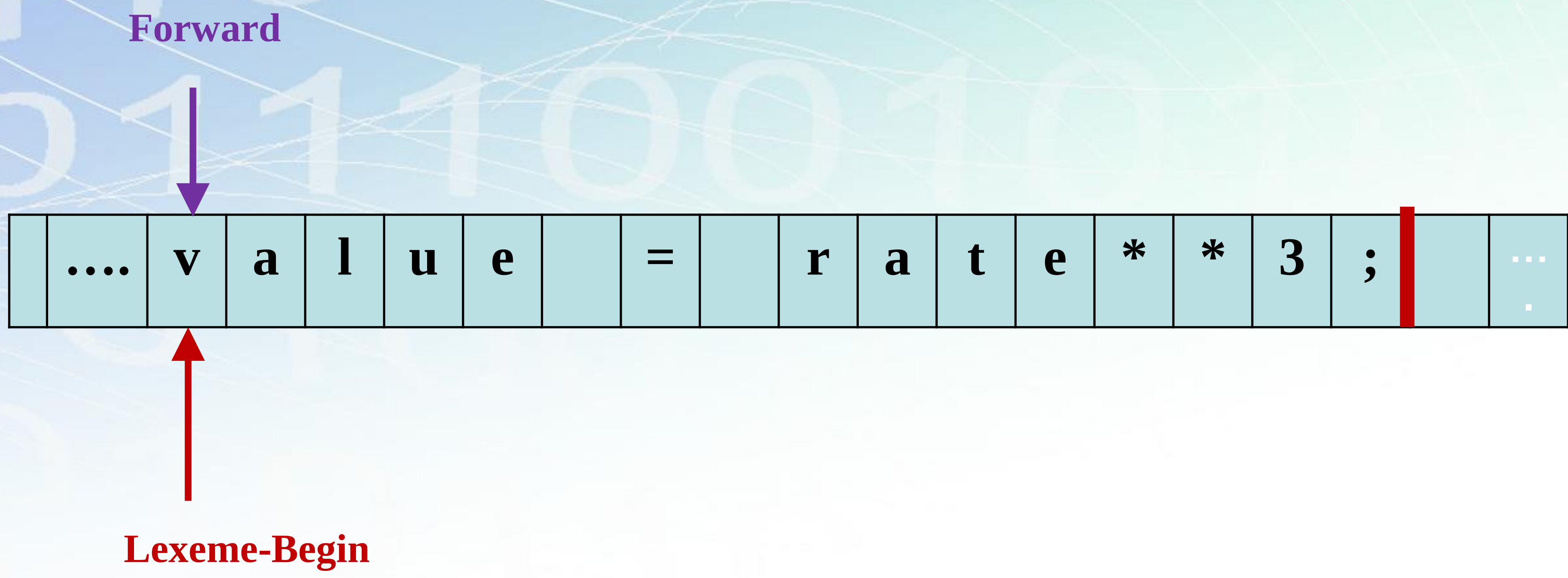
Buffer pairs

- Buffer is divided into two N-character halves
- N = Number of characters on one disk block Eg : 1024 or 4096 so on



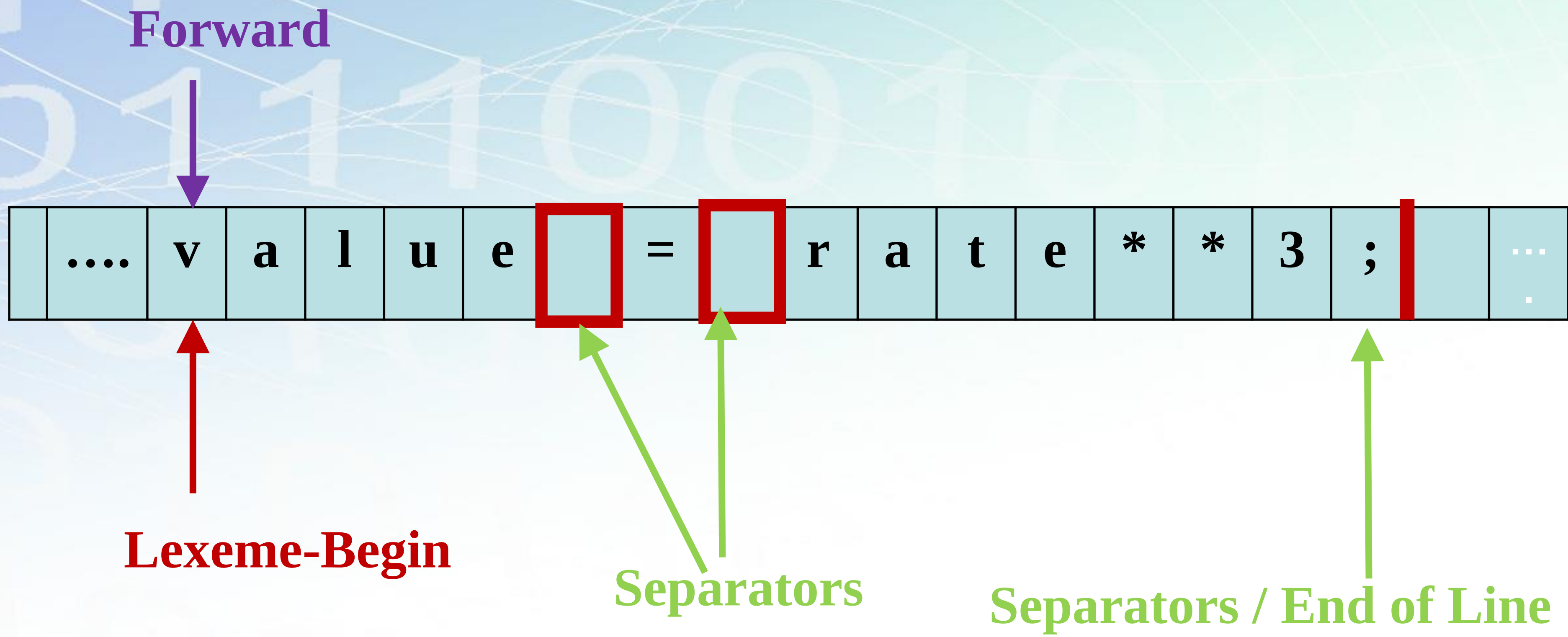
- Consider the statement

value = rate ** 3;



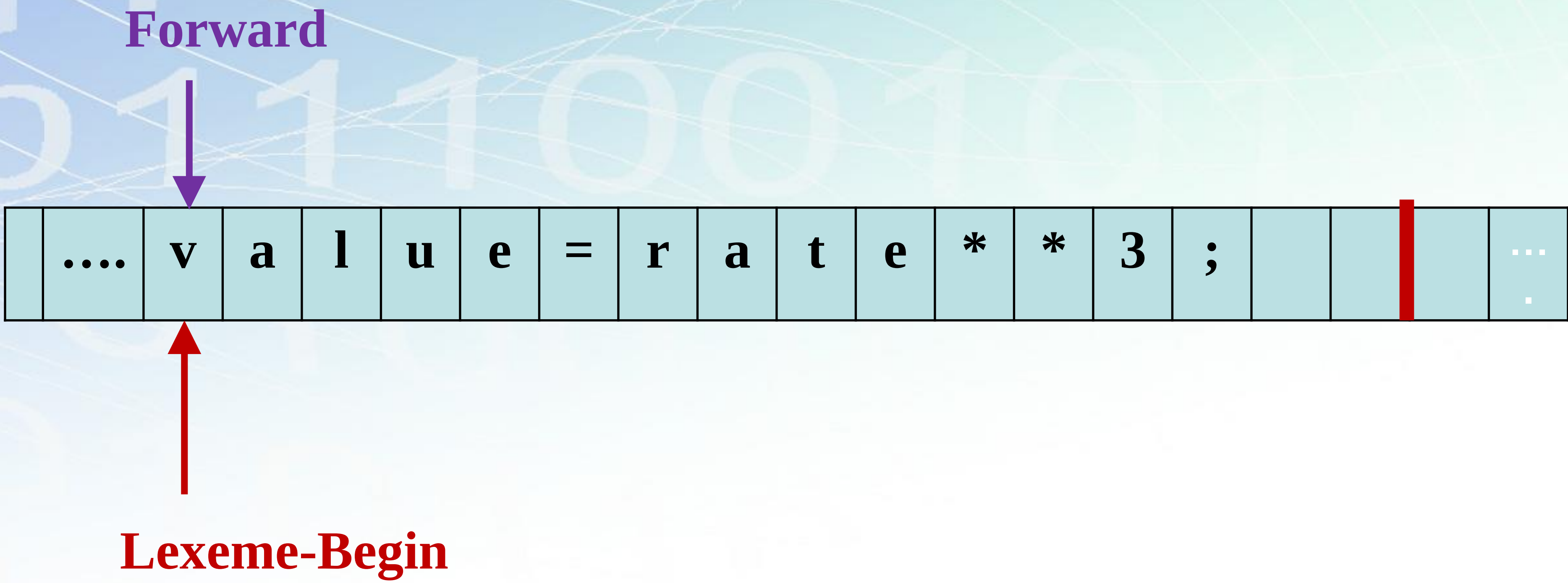
- Consider the statement

value = rate ** 3;



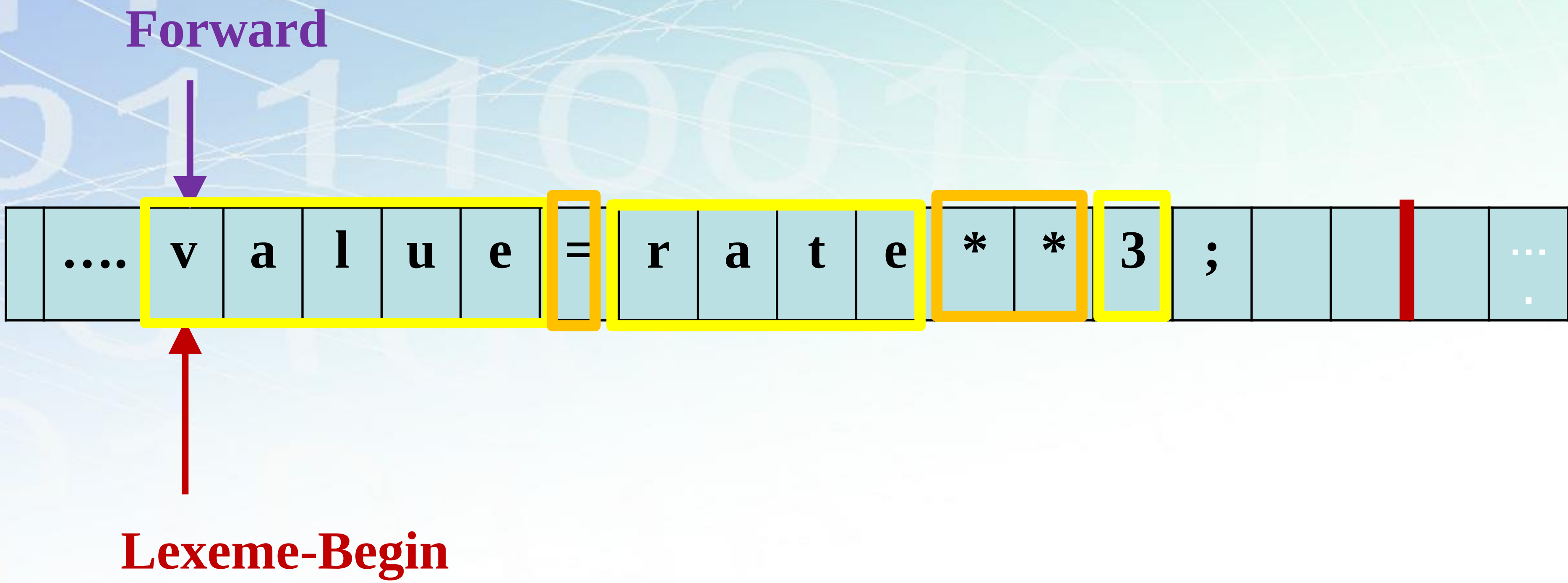
Consider the statement

value=rate3;**

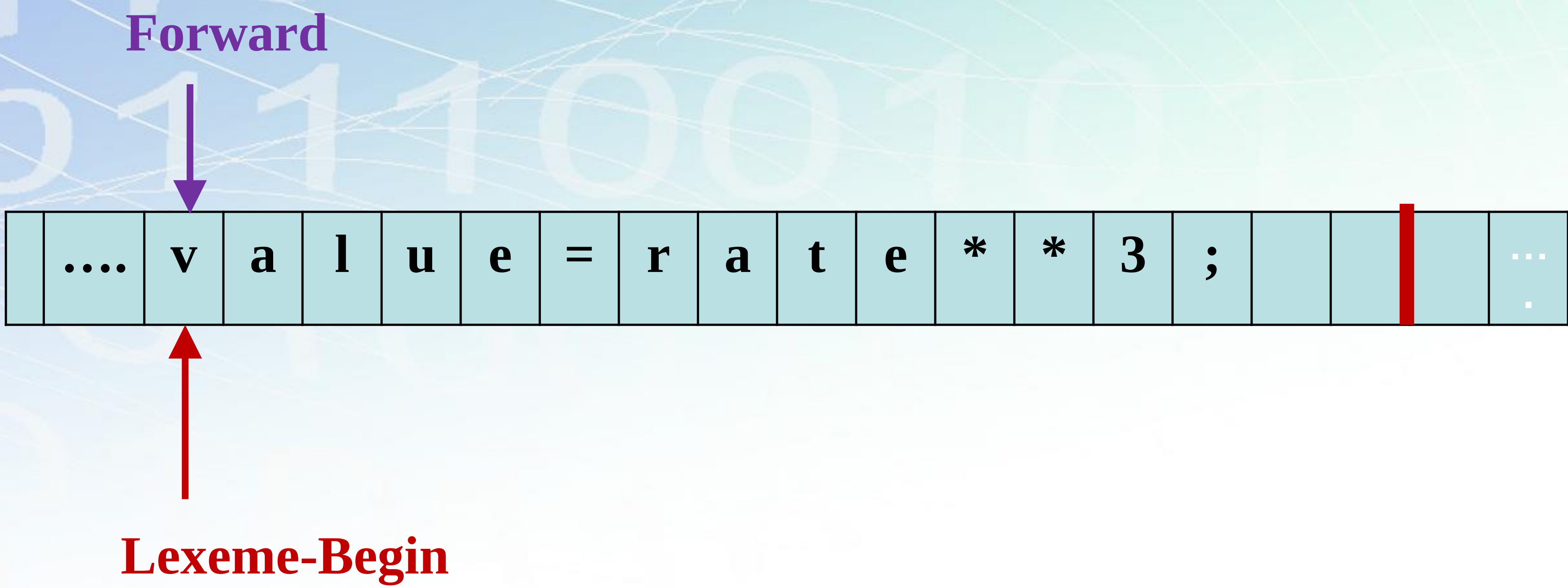


Consider the statement

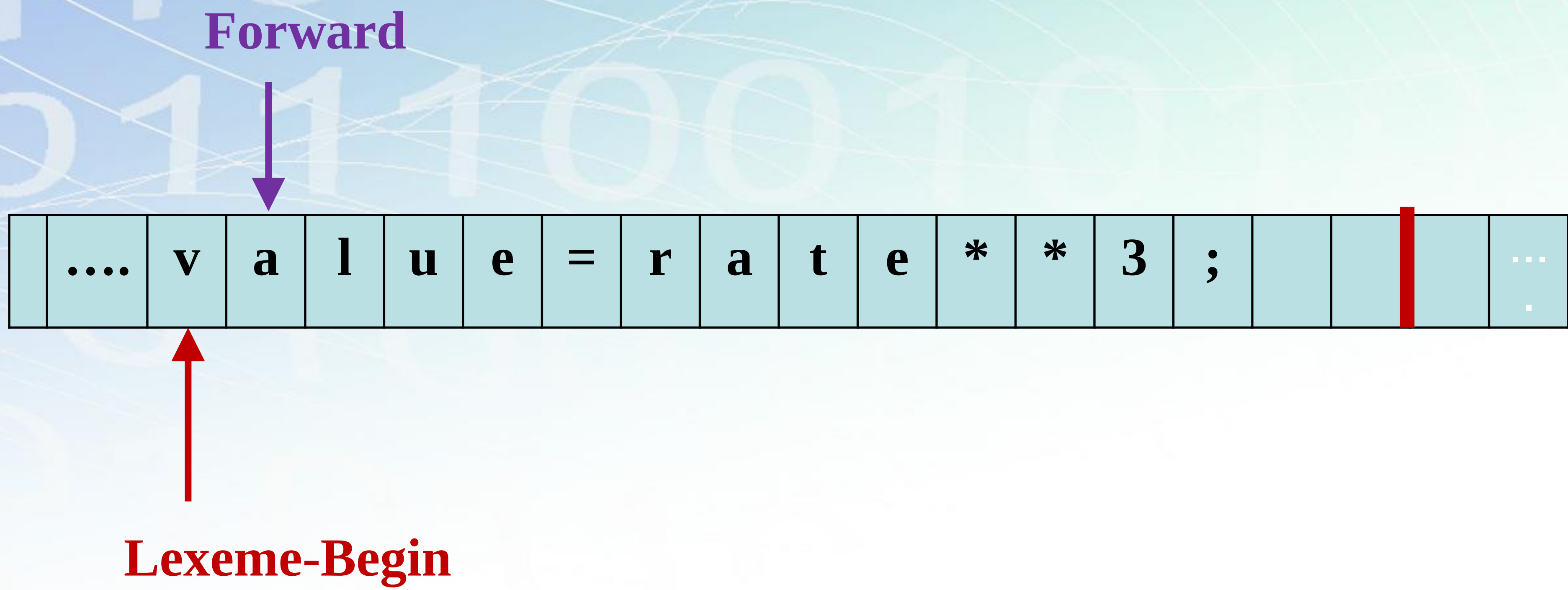
value=rate3;**



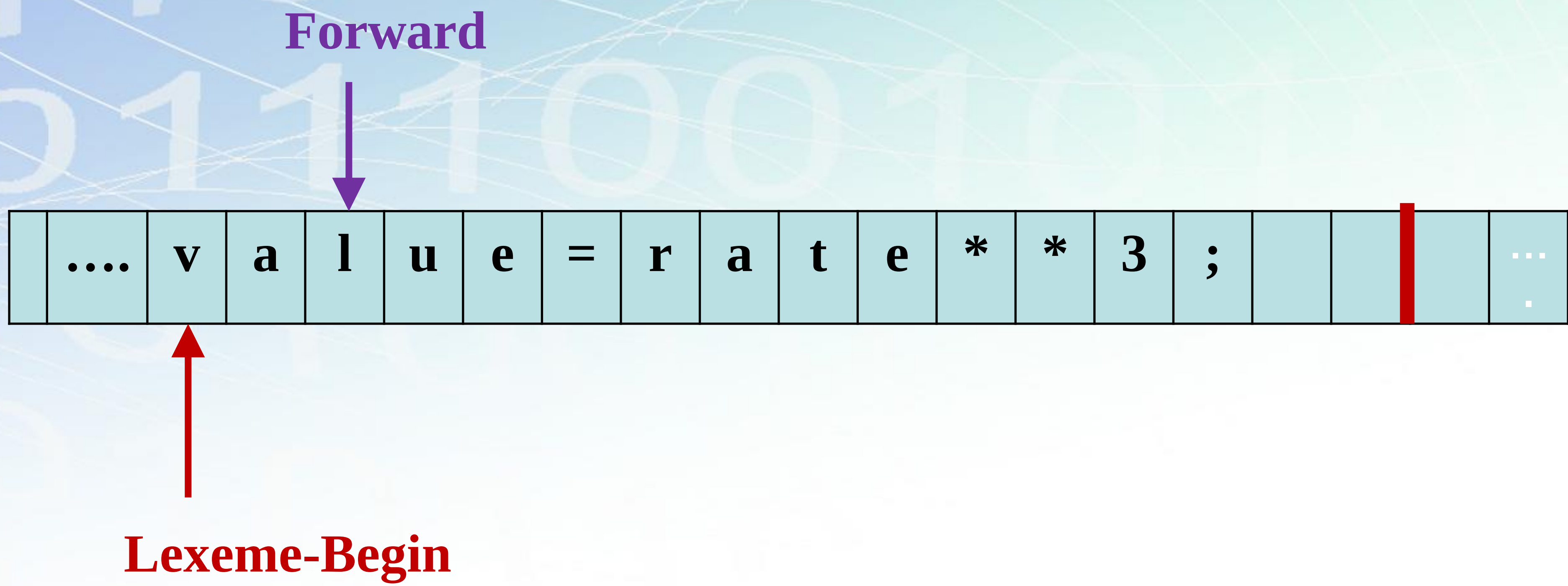
value=rate3;**



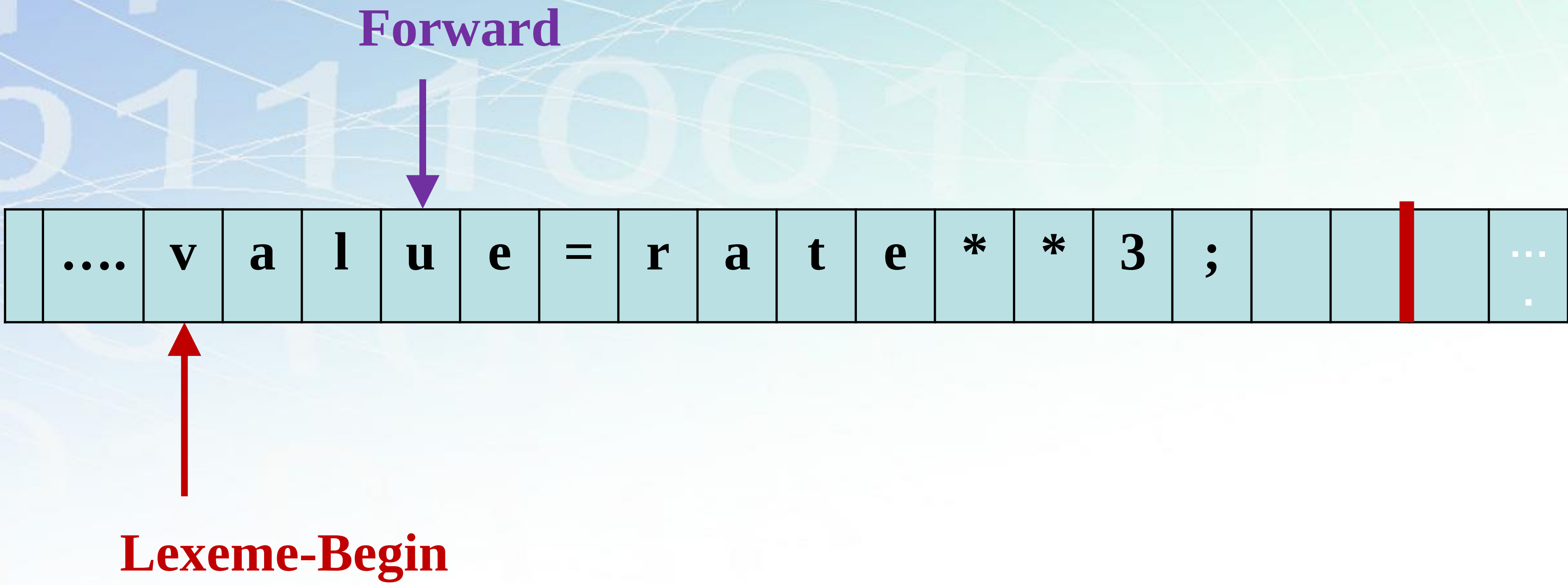
value=rate**3;



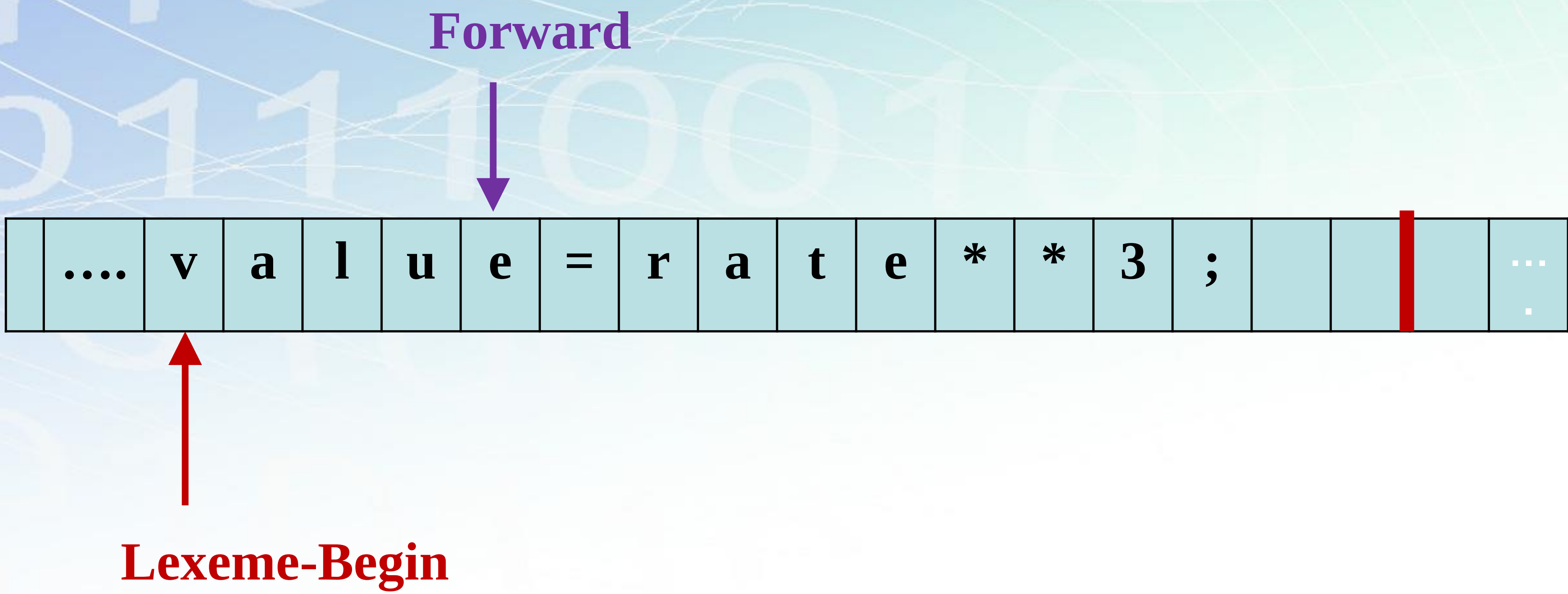
value=rate3;**



value=rate**3;



value=rate**3;



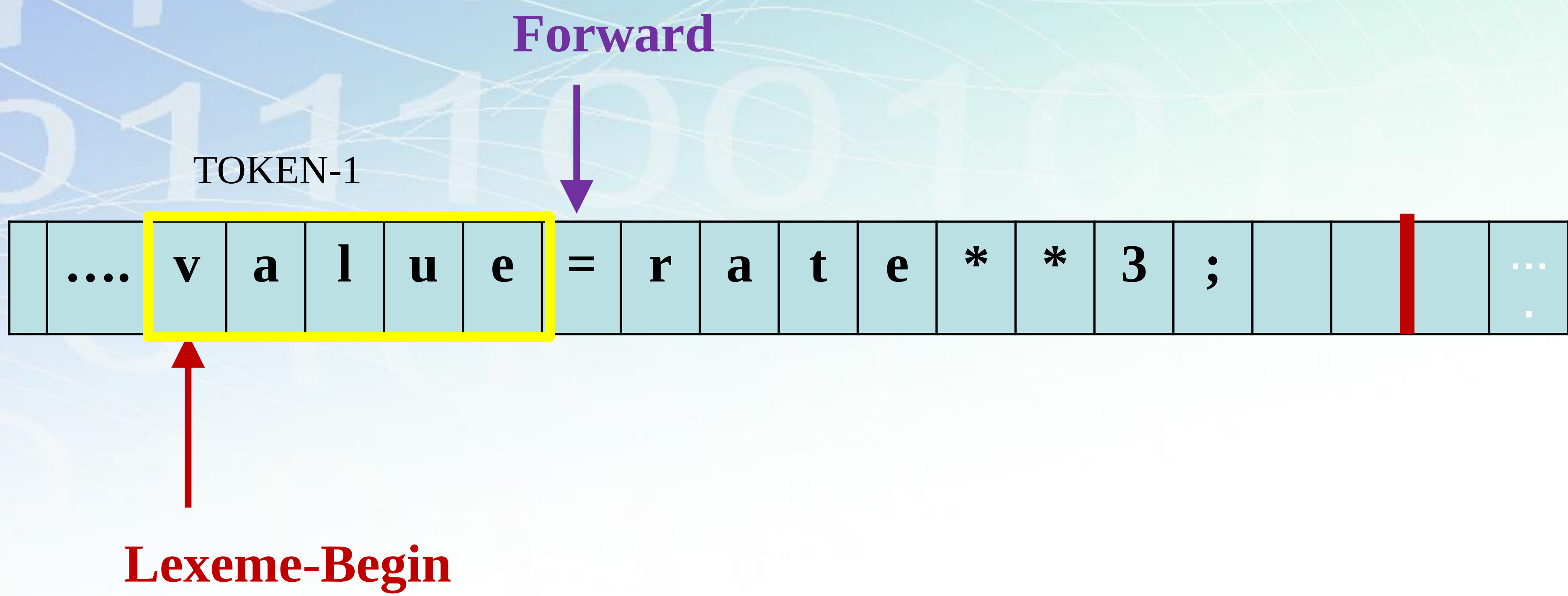
value=rate3;**

Forward

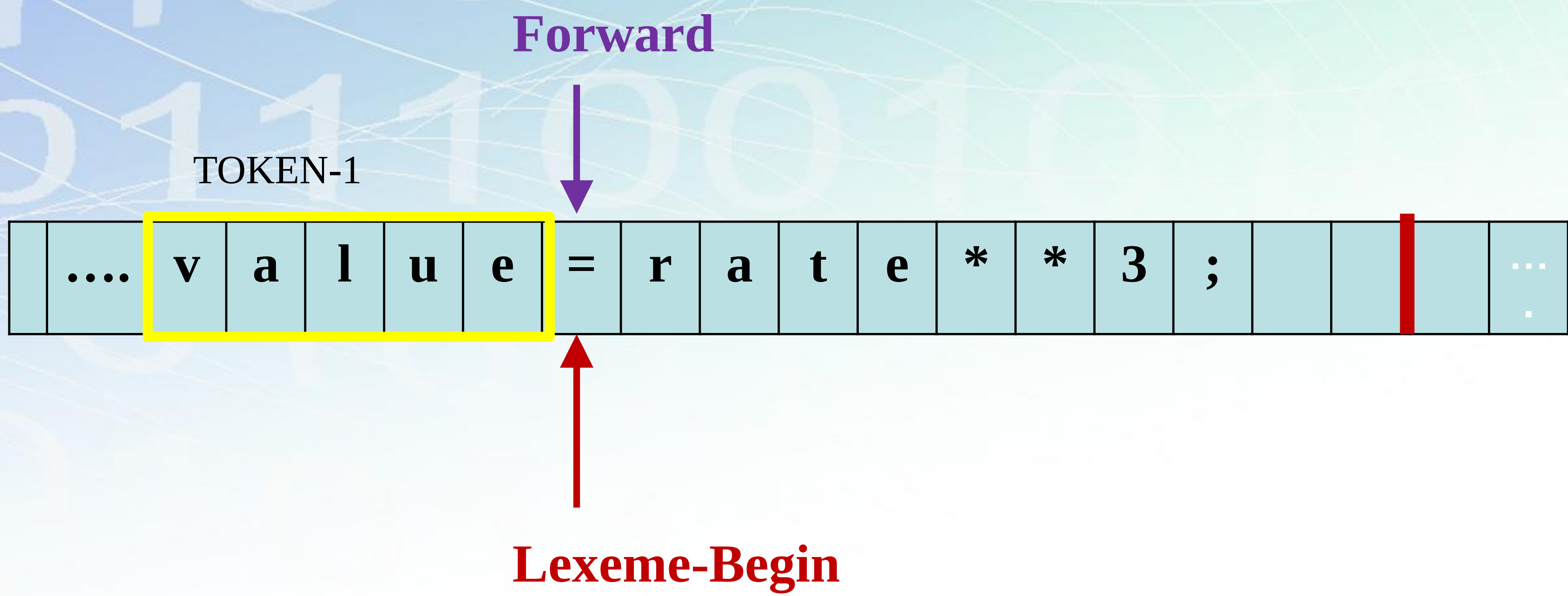
	….	v	a	l	u	e	=	r	a	t	e	*	*	3	;			….

Lexeme-Begin

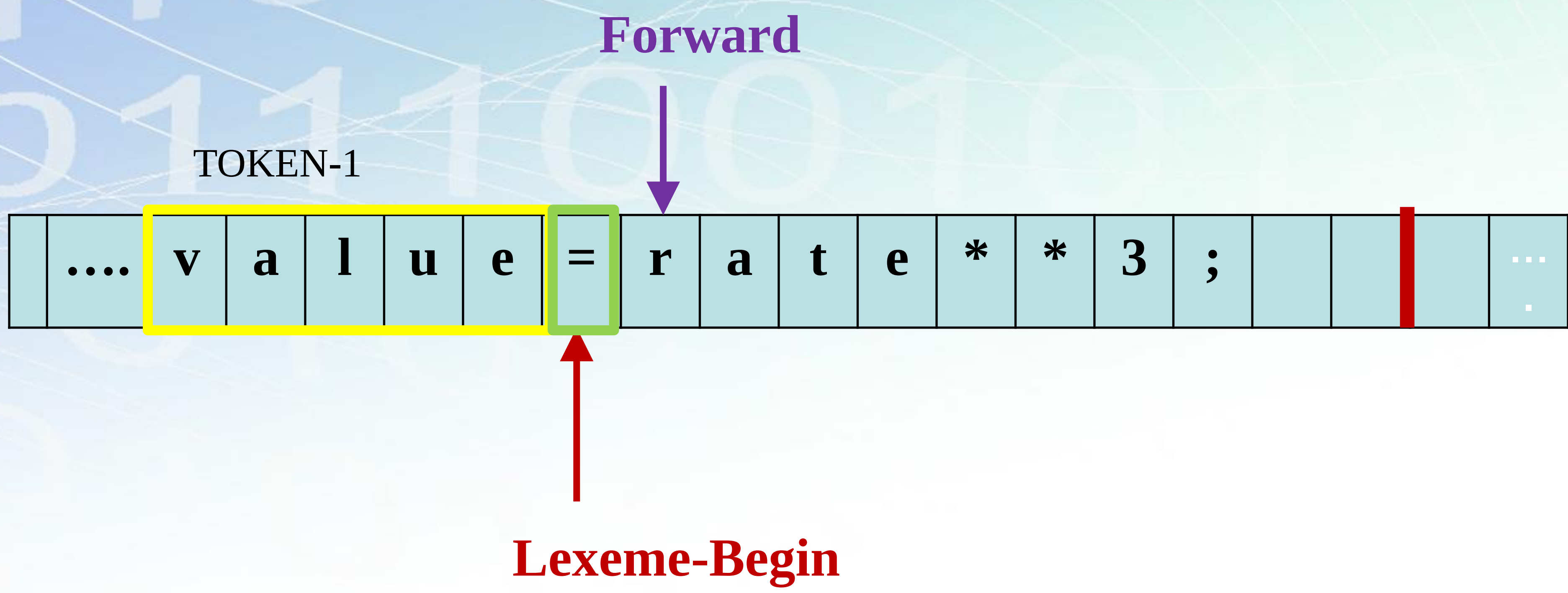
value=rate3;**



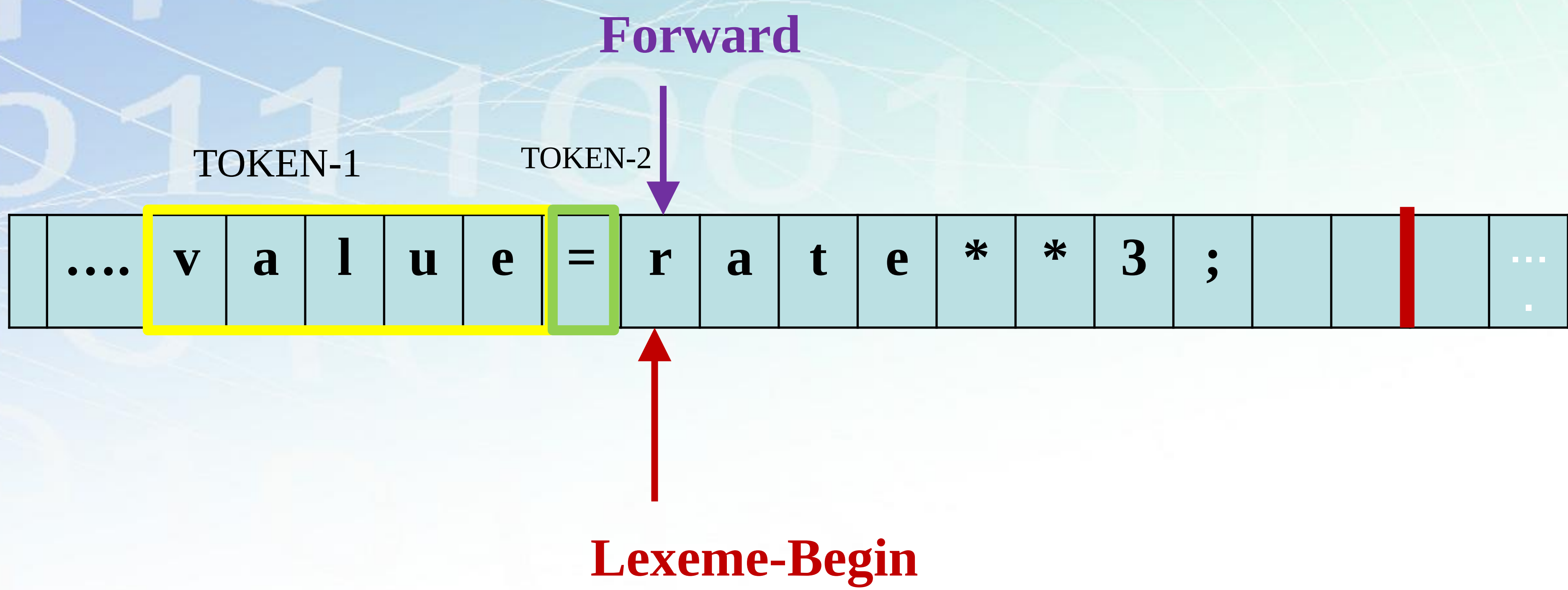
value=rate**3;



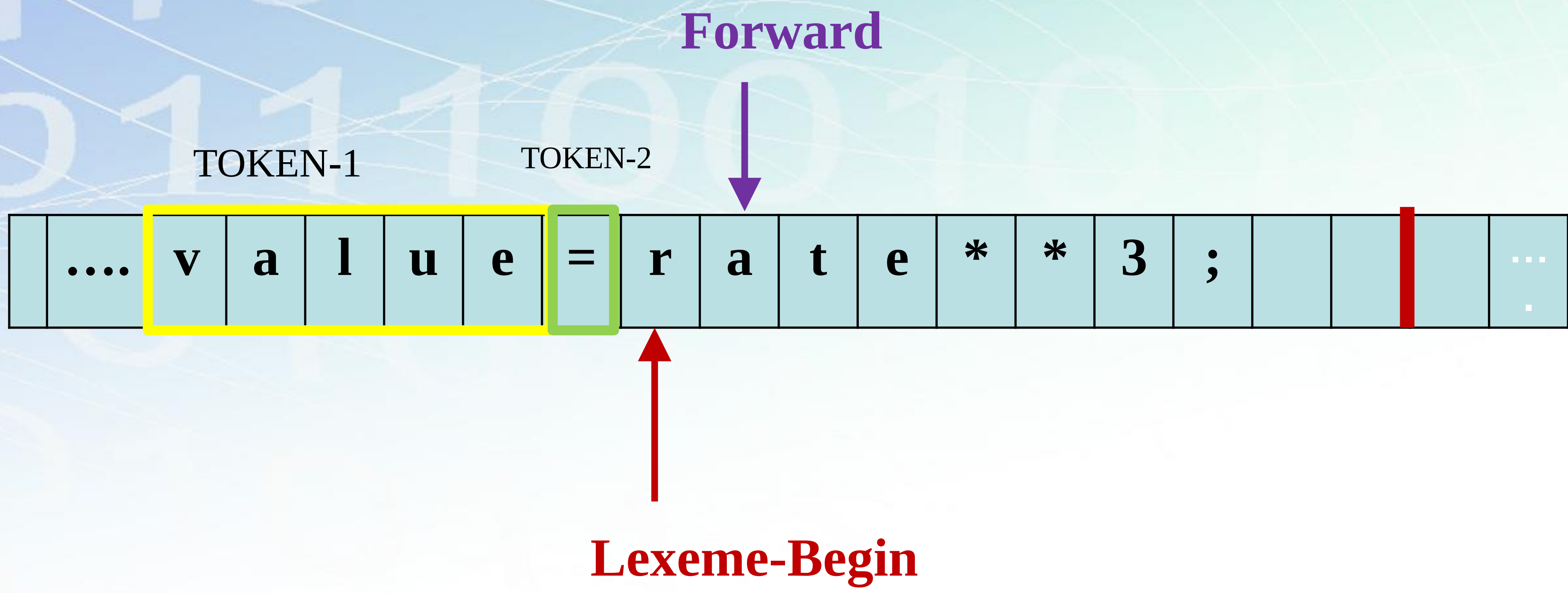
value=rate3;**



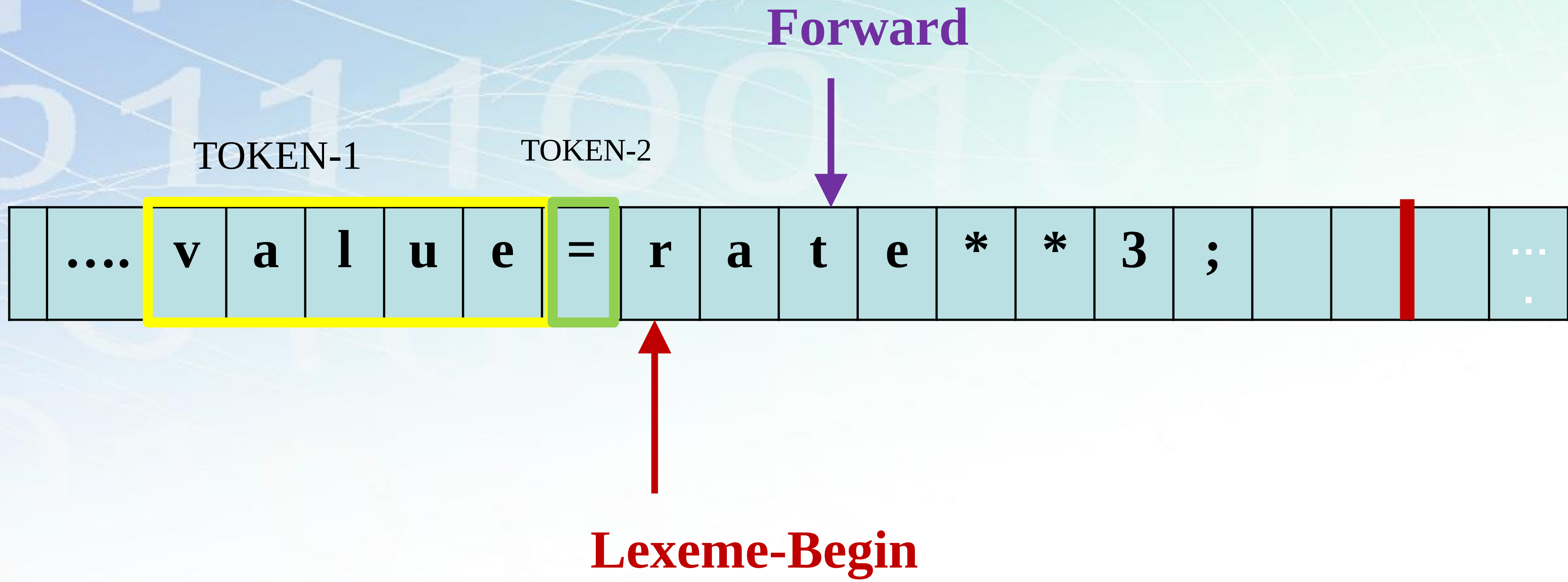
value=rate3;**



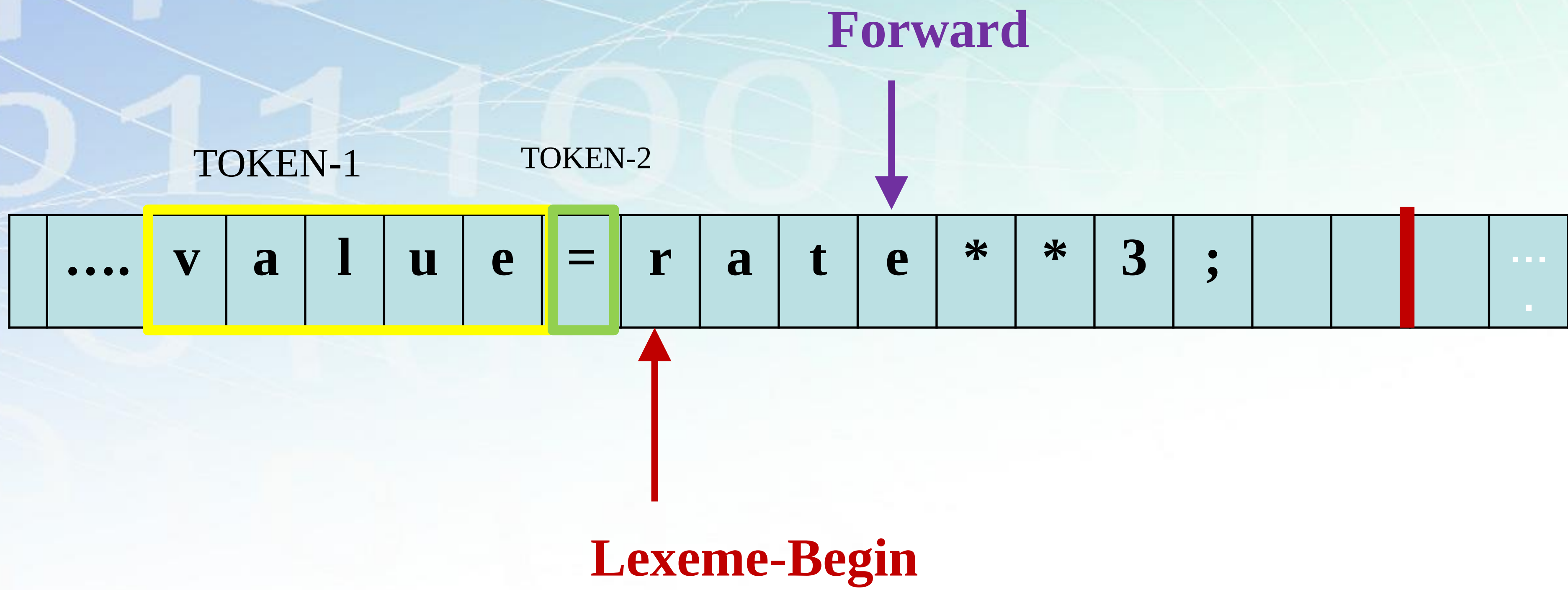
value=rate**3;



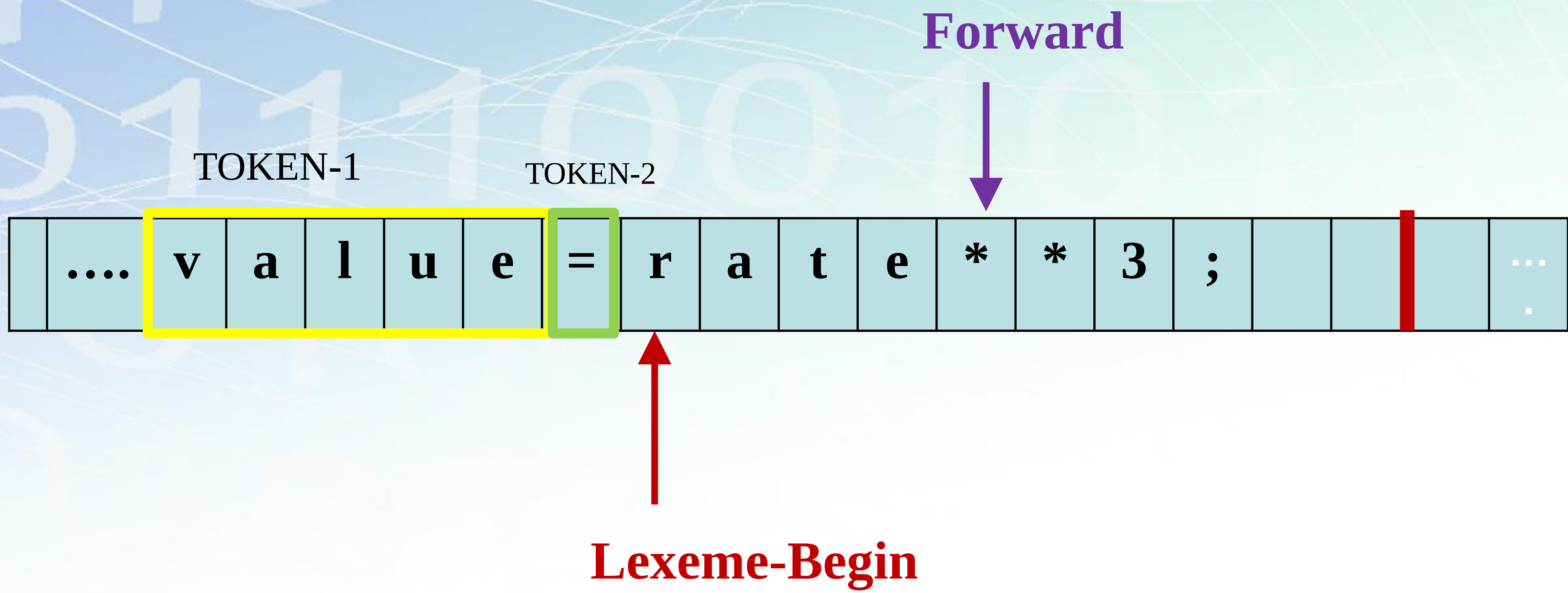
value=rate3;**



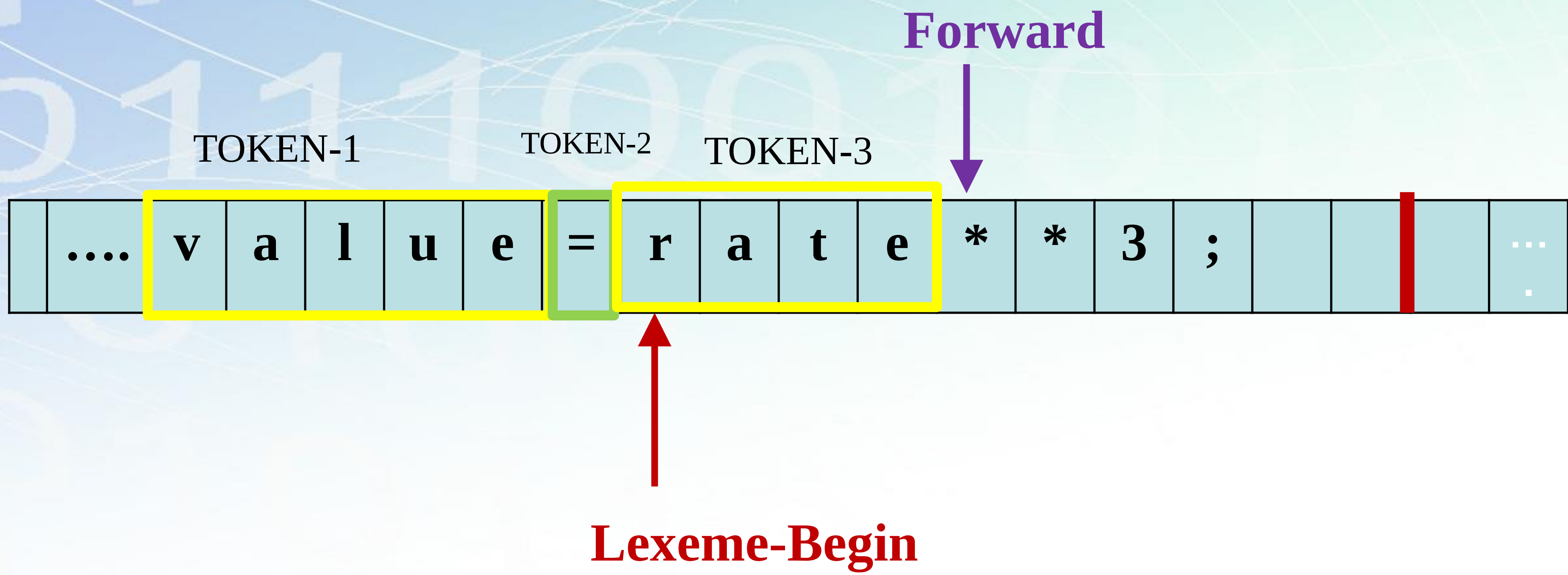
value=rate**3;



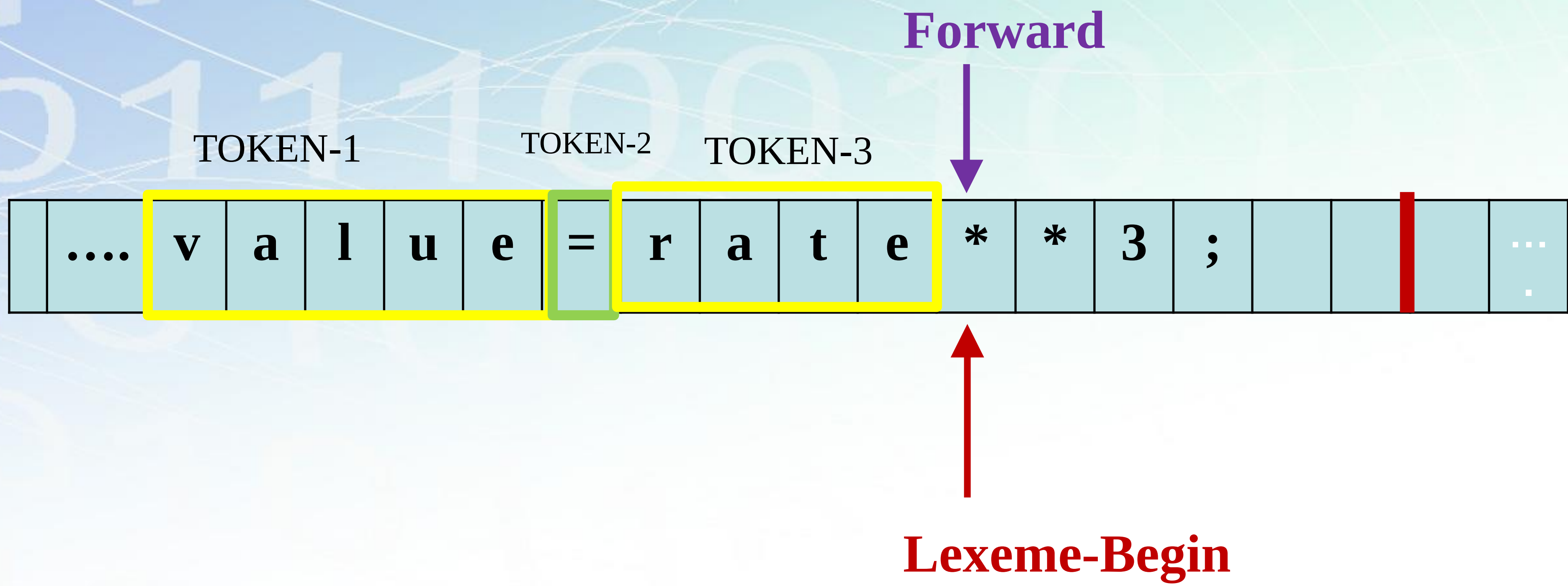
value=rate**3;



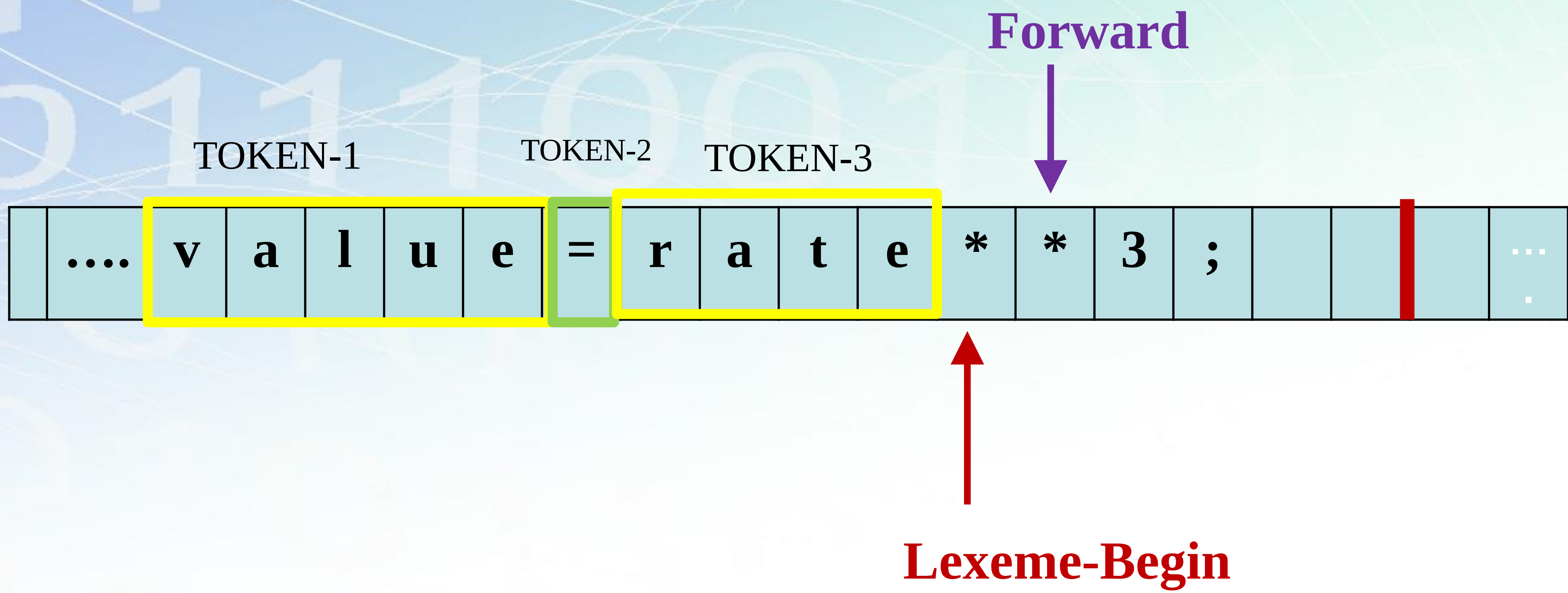
value=rate**3;



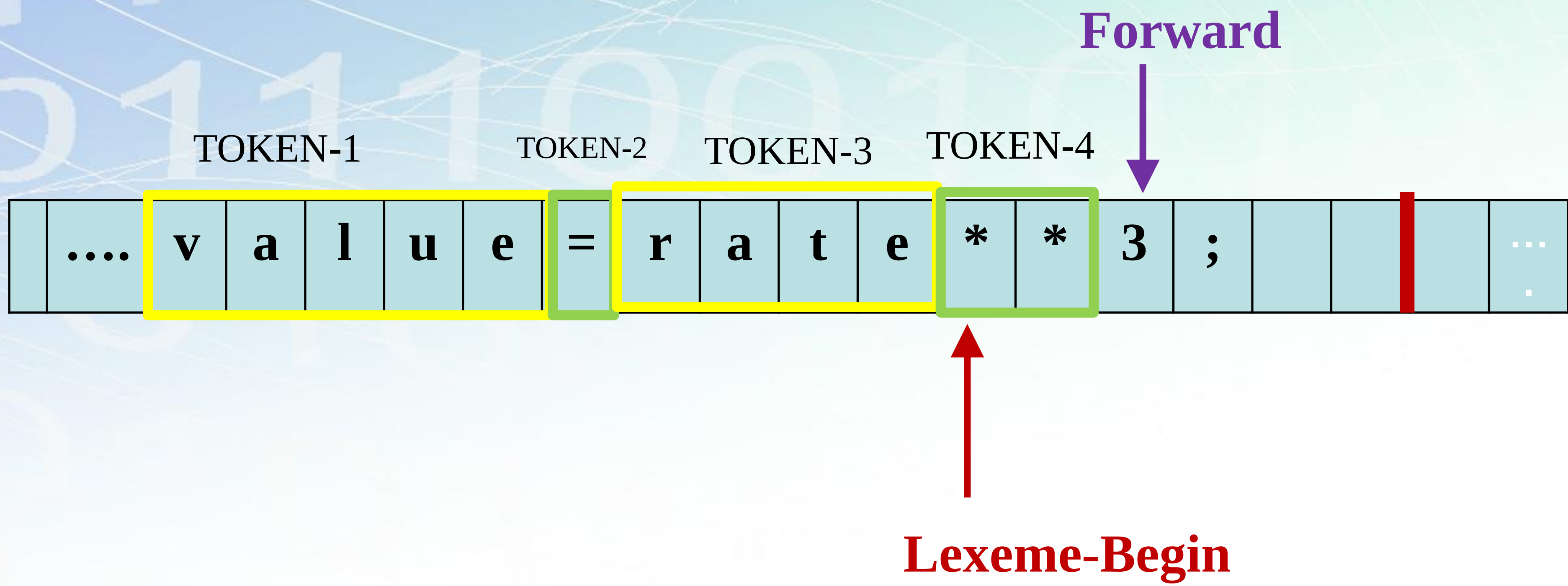
value=rate3;**



value=rate3;**

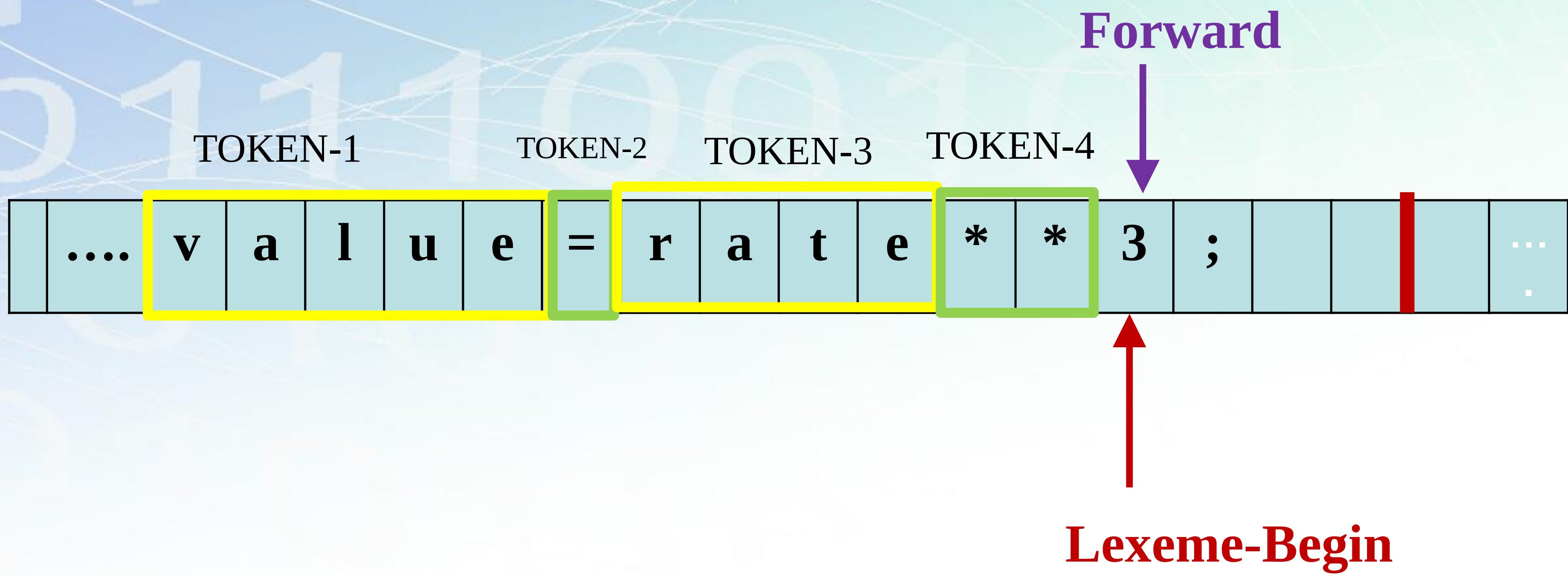


value=rate3;**



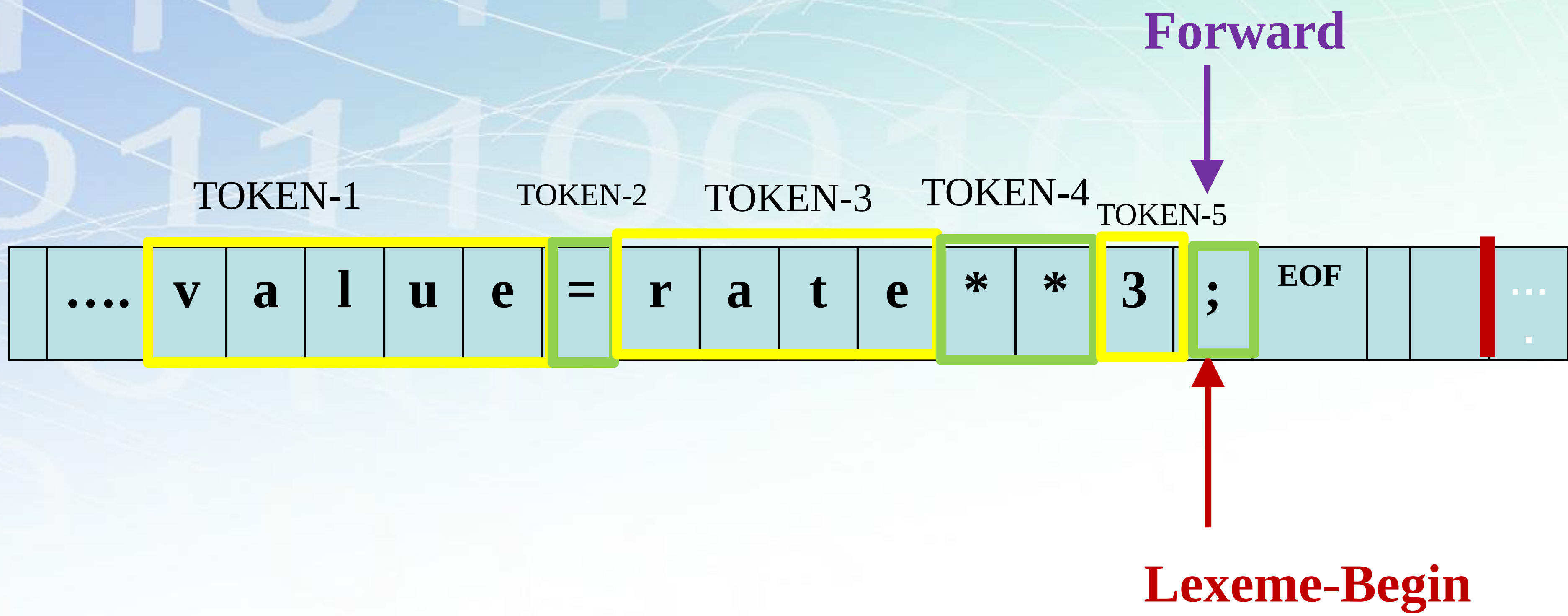
Forward
TOKEN-1
TOKEN-2
TOKEN-3
TOKEN-4
….
v
a
l
u
e
=
r
a
t
e
*
*
3
;
Lexeme-Begin

value=rate3;**

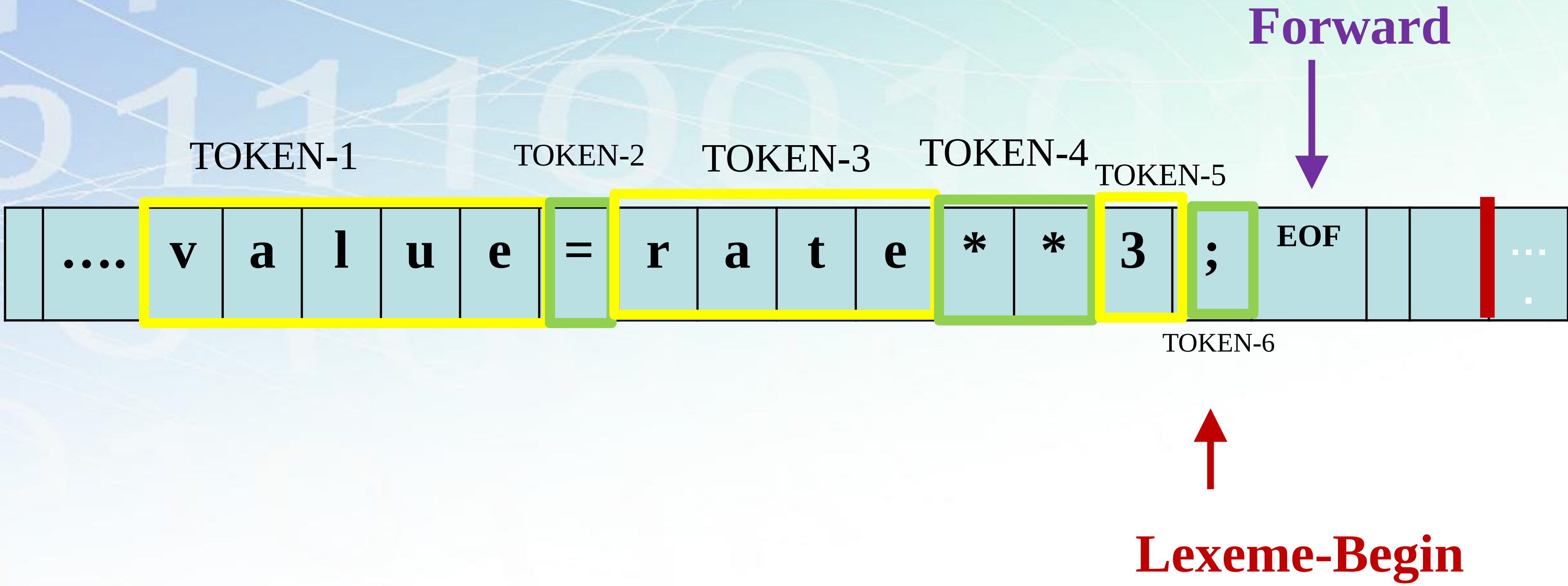


Forward
TOKEN-1
TOKEN-2
TOKEN-3
TOKEN-4
….
v
a
l
u
e
=
r
a
t
e
*
*
3
;
Lexeme-Begin

value=rate**3;


Forward
TOKEN-1
TOKEN-2
TOKEN-3
TOKEN-4
TOKEN-5
….
v
a
l
u
e
=
r
a
t
e
*
*
3
;
EOF
Lexeme-Begin

value=rate3;**

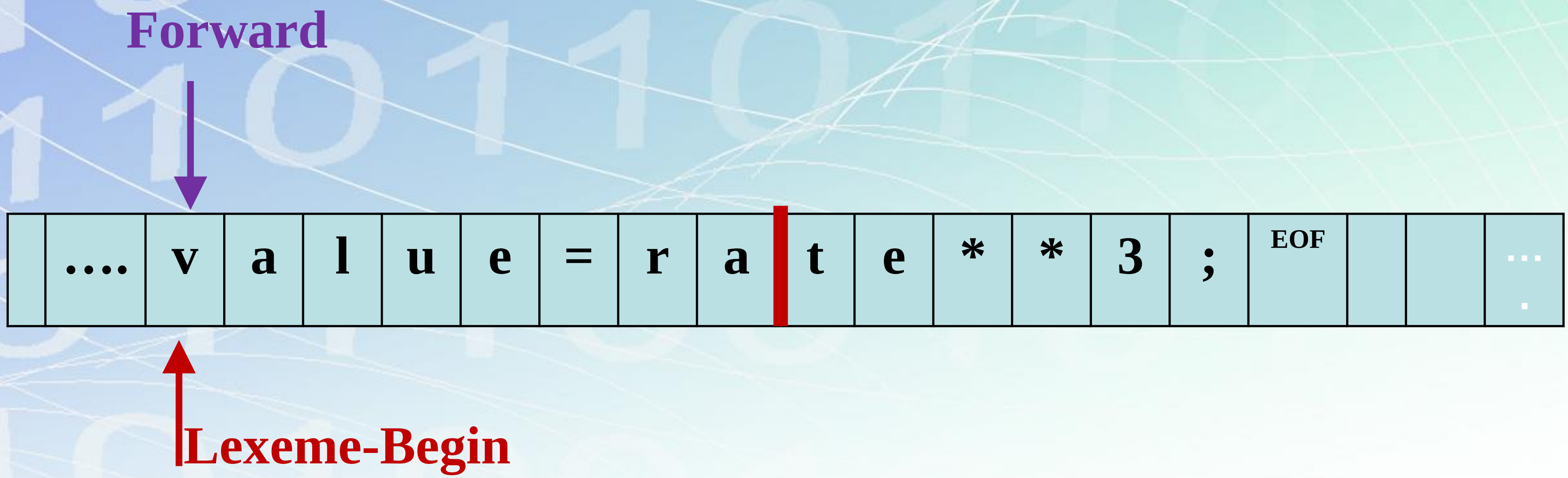


Note : EOF is inserted at the end if the number of characters is

less than N

Buffer Pairs

value=rate3;**



- Reads N bytes into one half of the buffer each time.
- If the input has less than N bytes , put a EOF marker in the buffer.
- when one buffer has been processed, it is reading N bytes into another half this continues.

if forward at end of first half **then begin**

TEST _1

reload second half;

forward := *forward* +1

end

TEST _2

else if forward at end of second half **then begin**

reload first half;

move *forward* to beginning of first half

end

else *forward* := *forward* + 1;

Forward ptr will take two test for each advance.
Therefore it causes overload

Buffer pair with Sentinels

- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a ***sentinel*** character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof.**
- **eof** can be used as a marker for the end of the entire input.

  Any **eof** that appears other than at the end of a buffer means that the input is at an end.

Forward

First half of Buffer 1 ends middle of the statement

	….	v	a	l	u	e	=	r	eof	a	t	e	*	*	3	;		…	eof

Lexeme-Begin

	..	v	a	l	u	e	=	r	eof	a	t	e	*	*	3	;	eof	…	eof

Statement ends at the Second half

```
forward : = forward + 1;
if forward = eof then begin
        if forward at end of first half then begin
                reload second half;
                forward := forward + 1
        end
        else if forward at end of second half then begin
                reload first half
                move forward to beginning of first half
        end
        else /* eof within a buffer signifying end of input */
                terminate lexical analysis
end
```

Advantages

- Most of the time, It performs only one test to see whether forward pointer points to an eof.
- Only when it reaches the end of the buffer half or eof, it performs more tests.
- Since N input characters are encountered between eofs, the average number of tests per input character is very close to 1.

Thank You!