Saint Petersburg Electrotechnical University "LETI"

The department MOEVM

The report on laboratory work №1

“Lexical analysis”

by discipline

“Development of linguistic processes”

Variant 19

Performed: A. Kondratev

Faculty CTI

Group 1381

Lecturer at the Department of MOEVM: Ivan Hodyrev

St. Petersburg

2015

In progress of the course work necessary to develop a programming language is a subset of the specified language, and the compiler from this language into an intermediate language, the type of which is determined by the option of individual tasks. Method of syntactic analysis (parsing) is also determined by the task.

The language must provide operations on variables and constants defined base types, as well as on the variables and the components of the derived type, which are defined for this assignment. The composition of operations should include both provided basic language, and listed in the job variant. In the language must be defined transaction type conversion when structural equivalence types or names. In the language has to be also possible to create user-defined types.

The language must allow the use of arithmetic expressions, in which may include constants and simple variables of basic types, the components of the structured type, parentheses, and signs of operations: addition, subtraction, multiplication, division. Priority of operations - an ordinary.

The language must allow the use of logical expressions of the relationship which may include parentheses and symbols of logic operations: AND, OR, NOT, and, in the case of a language in a Boolean constants and variables of this type. Priority of operations is usual.

Operations on variables of a structured type are defined for this job.

The composition of the operators of the language:

* assignment operator;
* operator input;
* output operator;
* composite operator;
* operator of unconditional transition;
* conditional statement, a condition in which a Boolean expression;
* cycle operator, a condition in which a Boolean expression.

The particular form of the operators is defined for this job.

The program in the input language can contain comments.

# Description of the source language.

## General information.

In the work describes the development of a compiler for the language is a subset of Pascal.

*Variant 19:*

* Basic language - Pascal.
* Basic types: integer, real, char.
* A structured type: character string.
* Operations on strings: the definition of the string length, string concatenation, replacing the substring, substring search, access to the substring.
* Additional requirements: functions, conditional assignment.
* The operator of the cycle - a precondition.
* The operator of overloading - allowed.
* Class of grammar - the grammar of operator precedence.
* The intermediate language - the triad.

## Lexis of the language.

## Terminal symbols of the language.

Terminal symbols of language:

* 26 letters of the alphabet, uppercase and lowercase: A B C ... Y Z a b c ... y z
* 33 letters of the Russian alphabet: А Б В ... Ю Я а б в ... ю я
* 10 numbers: 0 1 2 3 4 5 6 7 8 9
* 16 special characters: + - \* / = < > . , ( ) : { } \_ '
* space character
* ending character of line
* a tab character
  + 1. Tokens and an agreement on delimiters.

The program text is tokenized, which are:

* Identifiers
* Keywords
* Integers and real numbers
* Boolean values
* Special characters: + - \* / = < > . , ( ) : { } \_ ' - a single-character delimiters
* Combinations of special characters (a two-character delimiters):
  + <= Sign of the operator is less than or equal to.
  + >= Sign of greater than or equal operator.
  + := Mark assignment
  + <>Not equal sign operator.

## Comments.

## Comments - any characters enclosed in {}.

## Identifiers.

## Identifiers are used as names for variables, constants names and labels.

## Uppercase and lowercase letters are not distinguished. The restriction on the length of the length identifier can not be more than 127 characters. The identifier must be unique.

## Syntax:

|  |
| --- |
| **<identifier>**::=<letter> < sequence\_of\_letters\_and\_ digits>  **<letter>**::='\_'|'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|  |'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'  < **sequence\_of\_letters\_and\_ digits** >::={<letter>|<digit>}  **<digit>**::='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' |

## Constants.

## Constants are divided into integer, real, char, Boolean, and string. The constant is defined only once and can not be changed.

|  |
| --- |
| **<constant>**::=<decimal\_constant>|<real\_constant>|<char\_constant>|< Boolean\_constant>|< string\_constant >  **<** **decimal\_constant >**::=[<sign>]<sequence\_of\_digits>  < **sequence\_of\_digits** >::=<digit>{<digit>}  **<** **real\_constant >**::=< fractional\_constant >[< exponential\_part >]  < **fractional\_constant** >::=< sequence\_of\_digits >[.< sequence\_of\_digits >]  < **exponential\_part** >::=’e’[<sign>]< sequence\_of\_digits>|’E’[<sign>] < sequence\_of\_digits >  **<** **char\_constant >**::='''[<letter>|<digit>]'''  < **Boolean\_constant** >::=’true’|’false’  **<** **string\_constant >**::='''< sequence\_of\_letters\_and\_ digits >''' |

## Keywords

## Keywords are perceived as common symbols with fixed meaning, which is set in the definition of the language.

## The following keywords are reserved and can not be used as identifiers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| and | begin | boolean | char | const |
| do | else | end | false | function |
| goto | if | integer | label | not |
| operator | or | program | real | string |
| then | true | var | while |  |

## Strings

## A string is any sequence of characters enclosed in ‘apostrophes’.

## Descriptions and types.

## Types

## Integer signed type Integer: 2 bytes, range -32768..32767

## Real type Real: 4 bytes, range ±3.4\*10±38

## Boolean (logic) type Boolean: 1 bytes, range True-False

## Character type Char: 1 bytes

Operations:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Type of operator** | **Valid types of parameter** |
| + | Unary | integer, real |
| - | Unary | integer, real |
| + | Binary | integer, real |
| - | Binary | integer, real |
| \* | Binary | integer, real |
| / | Binary | real |
| And | Binary | boolean |
| Or | Binary | boolean |
| =, <>, <, >, <=, >= | Binary | integer, real, boolean, char |
| Not | Unary | boolean |

## Expressions

## Arithmetic expressions

|  |
| --- |
| **<arithmetic\_expression**>::=[<sign>]< therm >|< arithmetic\_expression ><sign><therm>  **< therm>::=<** multiplier>|< therm >’\*’< multiplier >|< therm >’/’< multiplier >  **<multiplier**>::='('<arithmetic\_expression>')'|<identifier>|<decimal\_constant>|< real\_constant>  **<sign>::='+'|'-'** |

## Logical expressions

|  |
| --- |
| **<** **logical\_expressions>**::=<logical\_therm>{’or ’< logical\_therm>}  **<** **logical\_therm >**::=< logical\_therm>’ and ’<logical\_multiplier>|<logical\_multiplier>  **<** **logical\_multiplier>**::=< identifier>|< arithmetic\_expression><ratio><arithmetic\_expression>|  <string\_constant><ratio><string\_constant>|’not’<logical\_multiplier>|’(‘<logical\_expressions >’)’  **<** **ratio >**::='='|'<'|'>'|'<='|'>=’ |

## Descriptions.

## Any description compares with notion of described by some identifier - variable name.

## Rules of the type conversions.

## This is a translation of "lower" types to "higher" in the interests of accuracy and consistency of the presentation data.

*Integer* --> *real*

## If in the operation are taking part operands of types of different levels (eg integer, and real), then the conversion is performed to the higher of the those present type (in this example to the real).

## More details about transformations:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Operand type1** | **Operand type2** | **Result Type** |
| + , -, \* | integer | integer | integer |
| +, -, \* | integer | real | real |
| +, -, \* | real | integer | real |
| +, -, \* | real | real | real |
| / | integer | integer | real |
| / | integer | real | real |
| / | real | integer | real |
| / | real | real | real |

## The transformation in the performance of the assignment operation:

|  |  |  |
| --- | --- | --- |
| **Variable type** | **Type assigned by the value** | **The variable type after executing operator** |
| Integer | Integer | Integer |
| Real | Real | Real |
| Real | Integer | Real |

## Converting types to perform of the operation relations.

|  |  |  |
| --- | --- | --- |
| **Operand1** | **Operand2** | **Conversion** |
| integer | integer | not |
| real | real | not |
| integer | real | Operand1--> real |
| real | integer | Operand2 --> real |

## Operators.

## The assignment operator.

## Syntax:

|  |
| --- |
| <**assignment\_operator**>::=<identifier>':='<arithmetic\_expression>|<logical\_expression>|<constant> |

## The operator of unconditional transition.

## Syntax:

|  |
| --- |
| **<** **operator\_of\_unconditional\_transition >**::='goto' <identifier> |

## The operator performs an unconditional transition to the labeled statement. The label must be unique within the program.

## The output operator.

## Syntax:

|  |
| --- |
| **<output\_operator**>::='write('<message> {','<message>} ')'  **<message**>::= <arithmetic\_expression>|<logical\_expression>|<string\_constant**>** |

## The operator outputs the standard output stream.

## The input operator.

## Syntax:

|  |
| --- |
| **<input\_operator>**::='read(' <identifier> {','<identifier>} ')' |

## The input operator reads from standard input (keyboard). The identifier cannot be a constant name.

## The composite operator.

## Syntax:

|  |
| --- |
| **<composite\_operator>**::= 'begin' {<operator> ';'} 'end' |

Operators that are part of a compound statement, is performed in order they were written.

## The conditional operator.

## Syntax:

|  |
| --- |
| **<** **conditional\_operator >**::='if '<logical\_expression>' then '<operator>|<composite\_operator> [' else   '<operator>|<composite\_operator>] |

## First calculated <logical\_expression>. If it is true, then the first < operator > is performed, else the second < operator >, if it there is.

## The operator of the while loop.

## Syntax:

|  |
| --- |
| **<while\_loop>**::='while '<logical\_expression> ' do ' <operator>|<composite\_operator> |

## Execute the operators <operator>, until the true is <logical\_expression>.

## Operator of determining the length of the string.

## Syntax:

|  |
| --- |
| **<operator\_length\_string>**::='length('<identifier>|<string\_constant>')' |

## The operator returns the length of the string.

## The operator of the string concatenation.

## Syntax:

|  |
| --- |
| **<operator\_string\_concatenation>**::='concat('<identifier>|<string\_constant>', '<identifier>|< string\_constant >')' |

The operator returns a string representing is passed the two series-connected the string.

## The operator of the replacement of the substring in a string.

## Syntax:

|  |
| --- |
| **<operator\_replacement\_substring>**::='replace('<identifier>|<string\_constant>', '<identifier>|<string\_constant>',   '<identifier >|< string\_constant >')' |

## The operator returns the modified string. If the substring is not found, the original string is returned.

## The search operator of substring in a string.

## Syntax:

|  |
| --- |
| **<search\_operator\_substring>**::='search('<identifier>|<string\_constant>', '<identifier>|<string\_constant>')' |

## The operator returns the index of element from which to start the substring, otherwise -1 is returned.

## The operator of access to the substring.

## Syntax:

|  |
| --- |
| **<operator\_access\_substring >**::='substr('<identifier>|<string\_constant>', '<identifier>|<sequence\_of\_digits>',   '<identifier>|<sequence\_of\_digits >')' |

## The operator returns the substring of a given number of elements starting at the specified.

## The operator of functions.

## Syntax:

|  |
| --- |
| <**operator\_function**>::=<function\_header> [<block\_definitions \_variables >] <composite\_operate> ‘;’  < **function\_header**>::=’function’ <identifier> ‘:’ <simply\_type> ‘;’ |

## Function allows splitting the logic of the program into smaller, more manageable fragments, and is similar to subprograms in other languages.

## The operator of conditional assignment.

## Syntax:

|  |
| --- |
| < **operator\_conditional\_assignment** >::= ‘if’ <logical\_expression> 'then' <assignment\_operator> ['else'   <assignment\_operator >] |

## As such, the ternary operator in Pascal is not. Use alternative is “if then else”.

## The operator of overloading.

## Syntax:

|  |
| --- |
| < **operator\_ overloading** >::= ‘operator’ < overloaded \_operations > ‘(‘<description\_overloaded\_variables > ’)’   <identifier> ‘:’ <simple\_type> ‘;’ <composite\_operator > ‘;’  < **description\_overloaded\_variables \_1**>::= < identifier> ‘:’ < simple\_type > ‘;’ < identifier> ‘:’ < simple\_type >  < **description\_overloaded\_variables \_2**>::= < identifier> ‘,’ < identifier> ‘:’ < simple\_type >  < **description\_overloaded\_variables** >::=< description\_overloaded\_variables \_1> |   <description\_overloaded\_variables \_2>  < **overloaded \_operations**> ::= <ratio>|’+’|’-’|’**\***’|’/’ |

# Description of lexical analysis stage.

The ratio between the tokens and lexemes for different of language constructions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Token** | **Lexemes** | **Language constractions** | **Token** | **Lexemes** | **Language constractions** |
| **\_ID** | str, count | Identifier | Sign of operations | | |
| **\_NUM** | 10, 1023 | An unsigned integer | **+** | + | Arithmetic |
| **\_STR** | 'some string' | Character string | **-** | - |
| Keywords | | | **\*** | \* |
| **\_PROG** | program | Relevant keywords | **/** | / |
| **\_CONST** | const | **\_NOT** | Not | Logical |
| **\_INT** | integer | **\_AND** | And |
| **\_REAL** | real | **\_OR** | Or |
| **\_STRING** | string |  |  |  |
| **\_GOTO** | goto | **\_REL** | <, <=, >=,>, <>, = | Ratio |
| **\_WHILE** | while | **\_EQ** | := | Assignment |
| **\_IF** | If | Special symbols | | |
| **\_THEN** | then |
| **\_ELSE** | else |
| **\_READ** | read | **.** | . | The relevant symbols |
| **\_WRITE** | write | **=** | = |
| **\_TRUE** | true | **(** | ( |
| **\_FALSE** | false | **)** | ) |
| **\_BOOL** | boolean | **;** | ; |
| **\_DO** | do | **,** | , |
| **\_CHAR** | char |  |  |
| **\_BEGIN** | begin | **:** | : |
| **\_END** | end |  |  |
| **\_VAR** | var |  |  |
| **\_LABEL** | label |  |  |
| **\_CONC** | concat |  |  |
| **\_LEN** | length |  |  |
| **\_REPL** | replace |  |  |
| **\_SUBS** | substr |  |  |
| **\_SRCH** | search |  |  |
| **\_FUNC** | function |  |  |
| **\_OPER** | operator |  |  |
| **\_L** | \_|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z |  | **\_RC** | 12.345e±67 | Real constant |
| **\_N** | 0|1|2|3|4|5|6|7|8|9 |  | **\_CC** | ‘a’, ‘b’, ‘2’, ' ' | Character constant |

## Description of the types lexemes.

## Considered the lexical analyzer lexemes can be one of the following types:

## Identifiers - the names of the user objects in the program;

## Constants - numeric or logical values ​​specified explicitly;

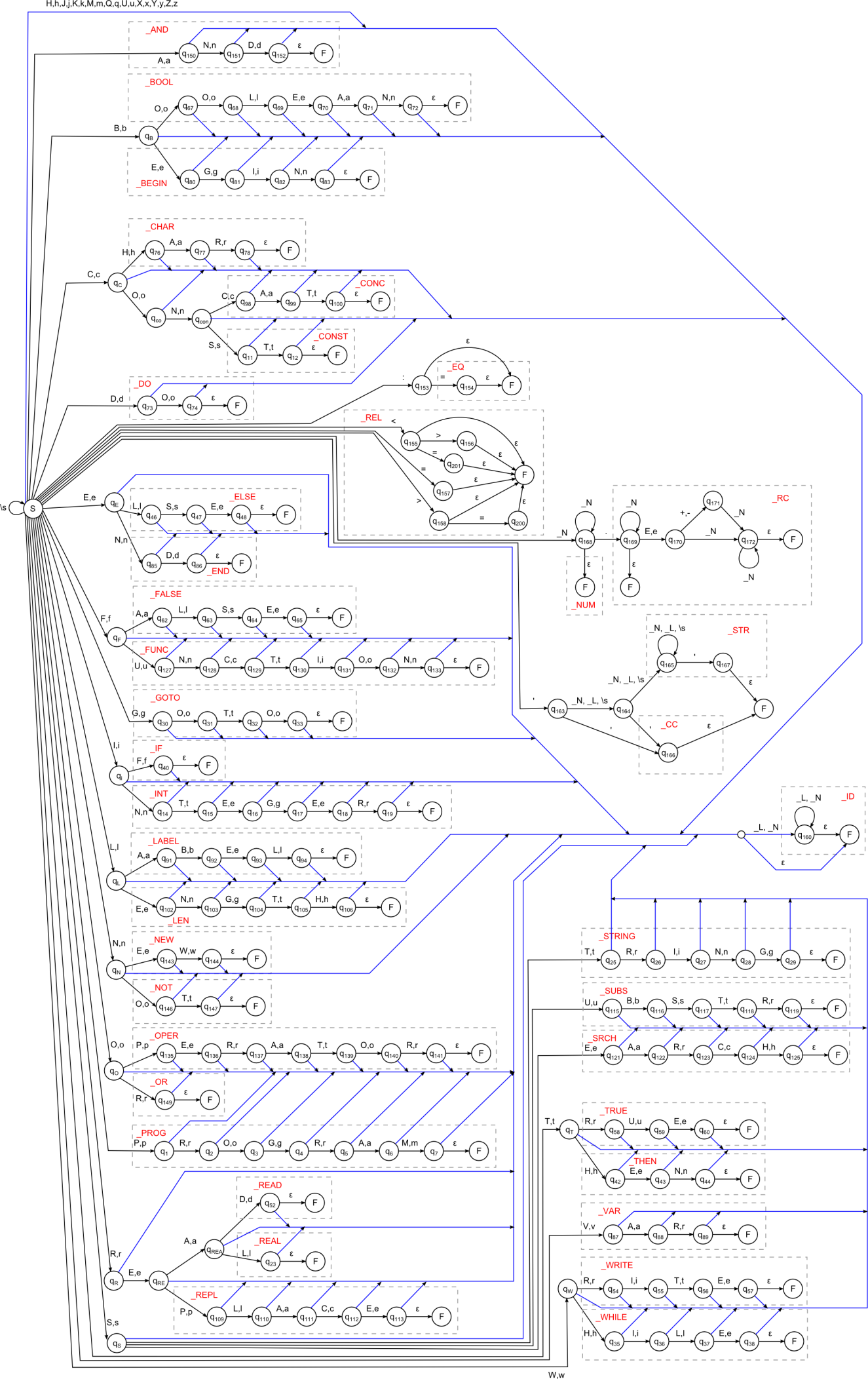
## Keywords - reserved identifiers, which have strictly defined meaning;

## Signs of operations - symbols that represent unary and binary operations;

## Special characters - the square brackets, parentheses, commas, etc;

## Separators - the gaps and newline characters. If the text includes the program in any place can be at least one delimiter, then the place can be any number of separators.

## Graph of lexical analysis.



## Modeling of the work of the lexical analyzer.

## For example, we have the following program at the input:

program HelloWorld;

var a:integer; b:boolean;

begin

a:=3;

b:=true;

if b and (a = 3) then

write(‘hello world’);

end.

Formed output stream of tokens:

\_PROG \_ID ; \_VAR \_ID : \_INT ; \_ID : \_BOOL ; \_BEGIN \_ID \_EQ \_NUM ; \_ID \_EQ \_TRUE ; \_IF \_ID \_AND ( \_ID = \_NUM ) \_THEN \_WRITE ( \_STR ) ; \_END .

As a result of the lexical analysis:

* The program parsed into separate lexemes.
* Lexemes are converted to tokens.

## Result.

As a result of laboratory work was a description of the syntax of the language, identified the concepts of construction of a lexical analyzer. It was implemented lexical analysis.